

Objektno programiranje

1. Problemi kompatibilnosti, kontinuiteta i ponovnog koriscenja

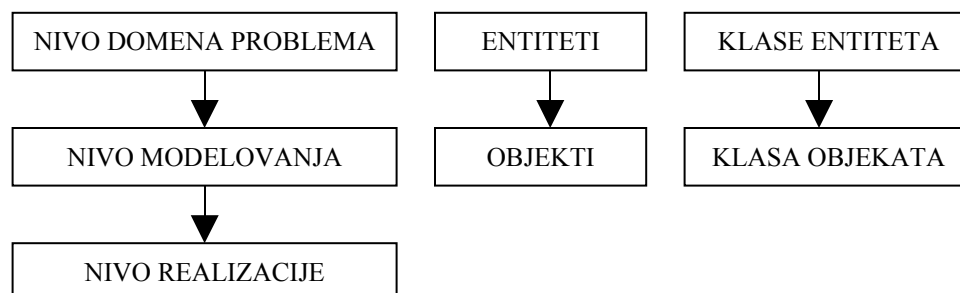
Problem KOMPATIBILNOSTI je direktna posledica algoritamske orjentisanosti izrade programa. Primera radi, ako razliciti timovi realizuju razlicite potprograme jednog programa, oni jednu vrstu podataka mogu definisati kao razlicite strukture i tada se svi potprogrami ne bi mogli implementirati u jedinstven program. Dakle, postoji potreba da se na pocetku definisu i naprave sve strukture podataka i tek onda pristupi izradi algoritama.

Problem KONTINUITETA zahteva da se moraju praviti programi za dugotrajnu upotrebu, dakle, da to bude jedan projekat koji ce se vremenom nadogradjivati, jer da bi projekat trajao na trzistu, u njega se moraju unositi izmene (tj. da se prave verzije). Ove izmene najcesce znace uvođenje novih funkcija, sto je izuzetno nezgodno raditi kod algoritamski orjentisanih programa jer je skoro nemoguće uvek menjati algoritam programa, dok je ove izmene dosta lako naciniti ukoliko se one unose u strukturu podataka, pri cemu algoritam programa ostaje skoro netaknut.

Problem MOGUCNOSTI VISESTRUKE UPOTREBE (Reusability) zahteva da se napisani softver moze koristiti ponovo vise puta, s tim da se dodaju neke izmene. Primer: problem izrade prozora – da se prozor napravi jednom i da se svi ostali prozori realizuju preko ovog postojećeg ubacivanjem nekih dodataka. Ovaj problem je NEMOGUCE ostvariti algoritamski.

2. Odnos algoritma i strukture podataka

Objektni programer se dugo zadrzava u DOMENU PROBLEMA (deo realnosti gde se nalazi ono sto zelimo resiti). Cilj mu je da IDENTIFIKUJE ENTITETE koji postoje u tom domenu. Primer: u problemu resavanja sistema jednačina $A \cdot x = b$ postoje entiteti matrice A i dva vektora x i b . Sledci korak je da se ENTITETI KLASIFIKUJU – u nasem primeru su to klasa entiteta matrica i klasa entiteta vektora. Zatim se ide na MODELOVANJE – izdvajanje za nas bitnih osobina klasa entiteta i samih entiteta. Tek posle toga ide REALIZACIJA tj. pisanje softvera. Potrebno je napomenuti da se u svaki model ukljucuje AKTIVNA KOMPONENTA tj. sta se sa modelom moze raditi.



Primer modela klase MATRICA:

- Naziv klase: Matrica

- Podaci: broj_vrsta, broj_kolona, vrenosti_elemenata
- Operacije: Upisati, Prikazati, ZadatiElement(i,j), OcitatiElement(i,j), Invertovati, Transponovati, ...

Svaka klasa ima PRIVATNU STRUKTURU PODATAKA i time je problem kompatibilnosti resen jer NEMA centralne strukture podataka. Takodje, svaka klasa ima PRIVATNE ALGORITME, pa je zato svaka klasa SAMODOVOLJNA tj. ona je struktura za sebe. Jednom napravljena klasa se vise ne treba menjati, vec se ona koristi uz dodavanje nekih novih osobina, sto se postize mehanizmom NASLEDJIVANJA.

Principi objektnog programiranja:

- a) Princip APSTRAKCIJE – odabiranje osobina koje su nam bitne za dato razmatranje.
- b) Princip SKRIVANJA INFORMACIJA – stavljanje u drugi plan detalja realizacije, koji su pri tome klijentu nedostupni.
- c) INKAPSULACIJA – mehanizam za ostvarivanje prethodna dva principa
- d) MODULARIZACIJA – (C++ koristi C-ove module)
- e) POLIMORFIZAM – oznacava kontekstno zavisno ponasanje
- f) Objektni program ima sve osobine sistema ciji su delovi povezani relacijama (glavna osobina je nasledjivanje).

3. Definicija klase i objekta

Nasa definicija za objekat ce biti na nivou modelovanja.

Def. Objekat je MODEL ENTITETA koji ima IDENTITET, STANJE i PONASANJE.

STANJE je deo proslosti i sadasnosti je neophodan za odredjivanje buduceg ponasanja objekta.

IDENTITET je slicno nazivu, nesto po cemu razlikujemo objekte. Moze se zadati kao naziv, preko adrese (pokazivacem) ili kljucem (koji bi trebao biti drugi po redu posle naziva).

PONASANJE je reakcija objekta na pobudu u vidu izvodjenja operacija.

Sinonim za objekat je INSTANCA KLASA (Instance).

Def. Klasa objekta predstavlja model klase entiteta koji obuhvata objekte sa istom strukturom i ponasanjem. Opisno receno, ono sto je objekat za klasu, to je promenljiva za tip podataka.

Struktura objekta:

Objekat u sebi može da sadrži i neki podobjekat. Strukturu objekta čine deskriptivni elementi i podobjekti datog objekta zajedno. Svi objekti iste klase moraju imati istu strukturu objekta.

Sadržaj objekta:

- PODACI-CLANOVI (nazivaju se još i atributi, mada je ovo neispravan naziv jer su atributi bitne osobine objekta, ali nisu podaci! Atributi su posledica stanja objekta, npr. Jabuka je crvena)
- OBJEKTI-CLANOVI
- FUNKCIJE-CLANICE (nazivaju se još i metode, mada je preciznije reći da su metode one funkcije članice koje su vidljive spolja tj. vidljive za klijenta, jer ne moraju sve funkcije objekta biti dostupne klijentu).

Aktiviranje metode se još naziva SLANJE PORUKE (Message).

KLIJENT KLASA je softverska komponenta (funkcija ili druga klasa) koja koristi ovu klasu (znači, NIJE KORISNIK!!!).

Ono što smo dosada nazivali funkcijama ćemo odsada nazivati SLOBODNIM FUNKCIJAMA, da bi se razlikovale od funkcija članica klase (metoda).

PROTOKOL KLASA je skup metoda zajedno sa pravilima njihovog korišćenja.

Principi:

- a) Sve je objekat.
- b) Program je skup objekata koji međusobno komuniciraju razmenom poruka (poruka = poziv tj. aktiviranje metode).
- c) Svaki objekat poseduje sopstvenu memoriju (Ovo je veoma lako narušiti! Primer: Slog koji ima pokazivac na neku strukturu. Ako slog kopiramo, dobijamo još jedan slog koji ima pokazivac na ISTU strukturu, dakle, ONI DELE MEMORIJU! Ovo ne vodi ka dobrom).
- d) Svaki objekat pripada jednoj klasi (ovo je analogija sa “svaka promenljiva pripada nekom tipu”).
- e) Svi objekti iz iste klase mogu da primaju iste poruke.

4. Deklarisanje klase u C++

Pretvaranje C-a u objektni jezik, tj. tvorac C++ je Bjarne Stroustrup. Osnovni pojam u objektnom programiranju je KLASA. Deklarisanje klase:

```
class Ime_Klase{
    podaci_clanovi
    objekti_clanovi
    funkcije_clanice
};
```

Primer: Deklarisacemo klasu tacke u ravni (Point)

//Naziv datoteke: POINT.H

```

class Point{
    private:
        double x,y;
    public:
        void SetPoint(double xx, double yy) {x=xx; y=yy;}
        double GetX() {return x;}
        double GetY() {return y;}
        double Distance();
};

```

//Naziv datoteke: POINT.CPP

```
#include "point.h"
```

```

double Point::Distance()
{
    return sqrt(x*x+y*y);
}

```

Rec "private:" oznacava deo klase koji je zatvoren za klijenta, tj. nije mu dostupan direktno. Rec "public:" oznacava deo klase koji je otvoren za klijenta, tj. direktno mu je dostupan. Postoji jos i naredba "protected:", koja oznacava segment dostupan samo za privilegovane klijente.

Napomena: ako se u klasi ove labela ne navedu, onda se podrazumeva da vazi PRIVATE, sto znaci da bi u klasi sve bilo zatvoreno za klijenta i klasa bi bila neupotrebljiva – tako nas programski jezik navodi da razmisljamo o zastiti podataka. Zato se "private:" cesto i ne pise jer ono vazi do dela sa labelom "public:".

Za funkcije clanice su podaci clanovi uvek OTVORENI (sto je prirodno, jer inace ne bi imali smisla), bez obzira sto se nalaze pod naredbom "private:" i time je klijentu omogucen pristup podacima clanovima samo preko funkcija clanica – tako se vrsi kontrola pristupa podacima!

Funkcije clanice (metode) se mogu zadati na dva nacina:

- a) **INLINE** funkcije – one se cele pisu u okviru klase i u celosti su zamene za makrodirektive (makrodirektive su uvedene jer su brze od potprograma, one se u celosti prevode i kao takve ugradjuju u program, dakle, kod njih nema ni skokova, ni smestanja podataka na stek). Ovo znaci da se inline funkcije prevode i direktno ugradjuju u program (dakle, nisu funkcije vec slozenije naredbe), ali posto kod njih NE SME BITI PROGRAMSKOG SKOKA, KAO NI STAVLJANJE NA STEK, ne smeju se koristiti slozenije naredbe, kao ni naredbe skoka (while, if, switch,...). U nasem primeru, inline funkcije su:


```
void SetPoint(double xx, double yy) {x=xx; y=yy;}
```

```
double GetX() {return x;}
double GetY() {return y;}
```

- b) OUTLINE funkcije – one koje nisu inline, a priradaju klasi, za njih se unutar klase navodi samo prototip. U našem primeru takva funkcija je :

```
double Distance();
```

Ova funkcija se mora cela napisati izvan klase, a da bi se oznacilo da ona pripada datoj klasi stavlja se kvalifikator klase:

```
Ime_Klase::Naziv_Funkcije(...) {...}
```

U našem primeru je to:

```
double Point::Distance()
{
    return sqrt(x*x+y*y);
}
```

Napomena: da smo ovde ispred double stavili naredbu "inline" (inline double Point::Distance()) , funkcija bi postala INLINE iako se fizicki ne nalazi unutar klase!

Ove funkcije članice klase su potpuno ravnopravne sa inline funkcijama.

Primer: instanciranje klase (kreiranje objekta):

```
//Naziv datoteke: "PROBA.CPP"
```

```
#include "point.h"
```

```
void main(void)
{
    Point a,b;
    double r;

    a.SetPoint(1,2);
    b=a;
    r=b.Distance()+3;
}
```

Postavice vrednosti podataka članova za tacku (objekat) a na x=1 i y=2

Neisparavno bi bilo napisati:

```
r=a.y;
```

Ovo ne bi radilo, jer je y u delu pod "private:", pa je zatvoren za klijenta!

Koji delovi klase se zatvaraju, a koji otvaraju za klijenta?

- Podaci članovi su u najvećem broju slučajeva zatvoreni, da bi se njihova izmena vrsila pod kontrolom. Dakle, kontrola promene sadržaja podataka članova je glavni razlog zatvaranja podataka članova.
- Objekti članovi se mogu ostaviti otvorenim ili se mogu zatvoriti, sto zavisi od opredeljenja. Ako zatvorimo objekat, onda se za njega moraju napraviti

funkcije koje njim rukuju u klasi. Ako ga otvorimo, onda njegovim internim metodama mozemo direktno pristupati, sto moze biti zbunjujuce.

- Funkcije clanice (metode) se uvek otvaraju.

Primer: klasa kompleksnog broja (ovu klasu cemo jos naknadno menjati)

//Naziv datoteke: "COMPLEX.H"

```
class Complex{
private:
    double r,i;
public:
    double Re() {return r;}
    double Im() {return i;}
    void Create(double rl, double img) {r=rl; i=img;}
    void Conjugate () {i=-i;}
    void ModArg(double &, double &);
};
```

Metode koje nemaju formalne parametre rade uvek nad podacima clanovima – u jeziku C++ je dozvoljen ovakav skracen zapis, pa se ne moraju pisati podaci clanovi

Referenca (upucivac ili alias)

Referenca: ona je po svojoj prirodi adresa, ali vrednost reference je vrednost podatka na koji ukazuje (dakle, nije isto sto i pokazivac, jer je vrednost pokazivaca adresa podatka na koji pokazuje). Koristi se kod prenosa po adresi, tj. kada se zeli izbeci pravljenje kopije podatka na steku. Jednom kada se definise referenca na podatak na nekoj adresi, ona uvek ostaje vezana za tu adresu – ovo znaci da se adresa na koju upucuje referenca ne moze menjati!

Dodatno objasnjenje reference: Ako definisemo celobrojnu promenljivu ceo_broj:

```
int ceo_broj;
```

i ako definisemo pokazivac koji pokazuje na ceo_broj i referencu koja ukazuje na ceo_broj:

```
int* p = &ceo_broj; //vrednost pokazivaca je adresa od ceo_broj
```

```
int& r = ceo_broj; //referenca r ukazuje na ceo_broj
```

tada je *p isto sto r jer oba oznacavaju vrednost na istoj adresi.

Globalno govoreci vazi: $\text{ref} \Leftrightarrow \text{*pok}$ ("ref" je neka referenca, a "pok" neki pokazivac na isti tip).

//Naziv datoteke: "COMPLEX.CPP"

```
#include "complex.h"
```

```
void Complex::ModArg(double &mod, double &arg)
{
    mod = sqrt(r*r+i*i);
    arg = (r==0 && i==0)? 0 : atan (i/r);
}
```

Iako je ovde arg ADRESA, ne pisemo *arg, vec samo arg, jer je u pitanju referenca, a ne pokazivac! Ovo isto vazi i za "mod".

Primer koriscenja ove klase u slobodnoj funkciji:

```
Complex Add(Complex z1, Complex z2)
{
    Complex w;
    w.Create(z1.Re()+z2.Re(), z1.Im()+z2.Im());
    return w;
}
```

Vidimo da se pristup podatku clanu "r" vrsi preko metode Re(). Da smo napisali **z1.r** ne bi radilo, jer je ovo slobodna funkcija (ona nije clanica klase i za nju su podaci clanovi zatvoreni!)

5. Definicija i vrste apstrakcije. Princip skrivanja informacija

Apstrakcija je klucni mehanizam naseg razmisljanja (lat. *apstrahere* – izdvojiti bitno). Primene apstrakcije:

- izdvajanje bitnih informacija
- detalji realizacije se proglašavaju nebitnim

Primer: $y = \sin(x)$; mi ovom naredbom dobijamo rezultat ttrazene funkcije, ali kako je to realizovano nas ne interesuje.

U praksi postoje tri vrste apstrakcije:

- apstrakcija entiteta (materijalna tacka)
- aps. akcije (u klasu se smestaju logicki srodne funkcije)
- aps. tipa virtuelne masine (postoje nivoi operacija – jedna operacija na visem nivou zamenjuje se skupom operacija na nizem).

Princip skrivanja informacija: (information hiding principle – postavio ga je Parnas) Detalji realizacije su razdvojeni od interfejsa i moraju biti nedostupni klijentu (tj. skriveni od njega). Interfejs klase je ono sto se nalazi u delu pod "public:".

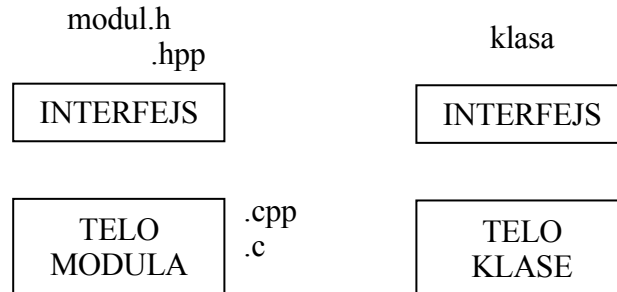
6. Pojam inkapsulacije i realizacija u C++. Pravilo inkapsulacije

Inkapsulacija i modularnost su sredstva za realizaciju klase tako da budu postavane sledece stavke:

- inkapsulacija je objedinjavanje strukture i ponasanja u softversku celinu uz ostvarivanje strukture ponasaanja
- naredba **class** služi za inkapsuliranje tj. objedinjavanje
- PRAVILO INKAPSULACIJE: objekat treba koristiti disciplinovano u skladu sa dokumentacijom proizvođača.

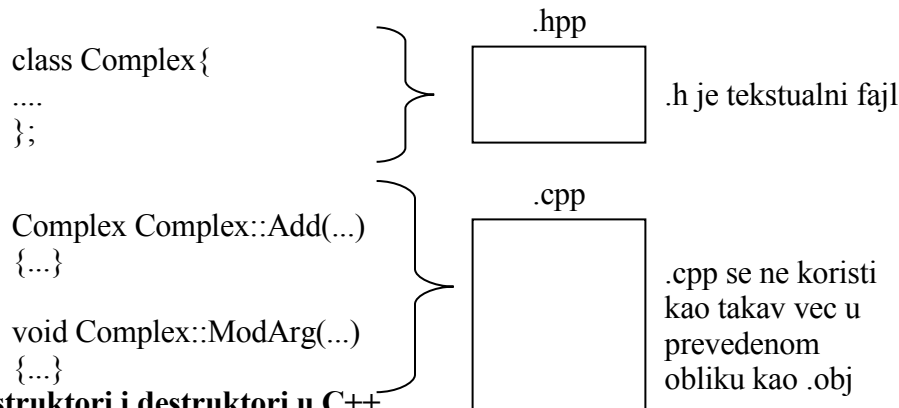
7. Pojam modula i primena u objektnom programiranju

Da bi klasa mogla više puta da se upotrebi (visekratna upotreba), klasa se stavlja u modul čija struktura je dvodelna:



Majerova jednakost: modul = klasa. Svaka klasa se nalazi u svom modulu ili može više logički srodnih klasa da se nađe u istom modulu: klasa ≤ modul.

Smestanje klase u modul:



8. Konstruktori i destruktori u C++

Videli smo da se definisanje objekta vrši naizgled isto kao definisanje promenljive, npr. `Complex z;`

Posto ovo ipak nije isto, kazemo da je objekat konstruisan i to se radi specijalnim metodama – konstruktorima, koji imaju sledeće sintaksne osobine:

- Nemaju nikakav tip (čak ni `VOID`)!
- Po imenu se moraju poklapati sa imenom klase
- Način poziva je različit od poziva ostalih metoda (nije `z.Complex;`, već `Complex z;`)

Svaka klasa ima barem jedan konstruktor (čak i ako nije naveden), ali ih može imati i više. Svi konstruktori će imati isto ime, pa će se njihovo prepoznavanje vršiti po ulaznim parametrima (Ovo znači da ne smeju postojati dva konstruktora sa jednakim ulaznim parametrima!)

Izdvajaju se određene podgrupe konstruktora:

- konstruktor kopije – ima specijalne osobine (više o tome kasnije)
- podrazumevani konstruktor – to je konstruktor bez parametara
- ugradjeni konstruktor – on spada u podrazumevane konstruktore, aktivira se kada za klasu nije naveden ni jedan konstruktor. Razlika podrazumevanog i ugradjenog konstruktora je u tome sto podrazumevani mi mozemo napisati. Ako klasu snabdemo barem jednim konstruktorom, tada ugradjeni konstruktor vise ne vazí.

Nekad se u izrazima u kodu mora pozvati konstruktor i kada se to implicitno radi, bice pozvan PODRAZUMEVANI konstruktor. Zato klasu cesto tako pisemo da napisemo i podrazumevani konstruktor, da bi u pomenutom slucaju on bio pozvan. Ponekad se podrazumevani konstruktor napise tako da ne radi nista i tada je on samo zamena za ugradjeni konstruktor i tada se u pomenutom slucaju on aktivira.

Primer: konstruktori za nasu klasu Complex:

//Naziv datoteke: COMPLEX.H

```
class Complex{
    private:
        double r,i;
    public:
        Complex() {r=0; i=0;}
        Complex(Complex *);
        Complex(Complex &);
        Complex(double rl, double img) {r=rl; i=img;}
};
```

Ovo je podrazumevani konstruktor. Da smo ga napisali kao:
Complex () {}
bio bi zamena za ugradjeni konstruktor.

Ulazni parametar konstruktora klase ne moze biti ta ista klasa, tj. ne moze se napisati:
Complex(Complex z) {...}
Ovo se resava tako sto se klasa prosledi preko adrese pomocu pokazivaca ili reference.

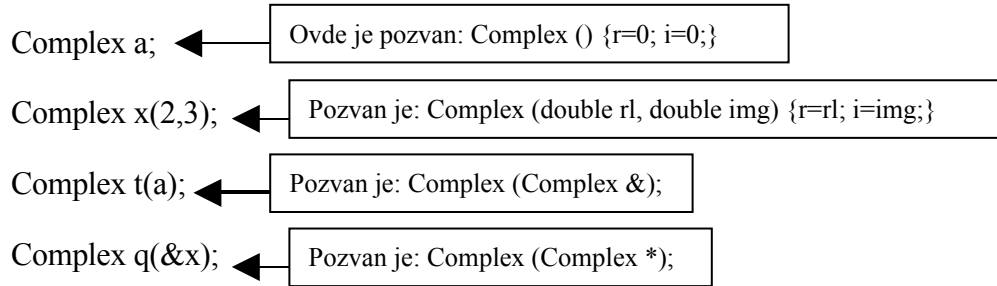
//Naziv datoteke: COMPLEX.CPP

#include "complex.h"

```
Complex::Complex(Complex *z)
{
    r=z->r;
    i=z->i;
}
```

```
Complex::Complex(Complex &z)
{
    r=z.r;
    i=z.i;
}
```

Pozivi ovih navedenih konstruktora ce biti:



U C++ je moguće zadati funkcije tako da postoje takozvane PODRAZUMEVANE VREDNOSTI i ovo je moguće uraditi za sve funkcije. Primer:
 double f(double xx=1, double yy=0, double zz=3) {x=xx; y=yy; z=zz}

Funkcije koje imaju ovakve parametre mogu biti pozvane i bez navodjenja ulaznih vrednosti jer se tada koriste podrazumevane vrednosti. Postoji jedino restrikcija u tom pogledu da cim jedan parametar dobije podrazumevanu vrednost, svi ulazni parametri funkcije iza njega takodje dobijaju podrazumevane vrednosti. Primera radi, ako gore definisanu funkciju pozovemo sa:

f(4);
 vrednosti za x,y,z ce biti sledece: x=4, y=0, z=3.
 Jasno je da i konstruktori mogu imati podrazumevane vrednosti:
 Complex (double rl=0, double img=0) {r= rl; i=img;}
 Tada ako konstruktor pozovemo na sledeci nacin:
 Complex w(5);
 podaci clanovi ce dobiti vrednosti r=5, i=0.

U tipicnom slucaju konstruktor vrsi i inicijalizaciju, pa uvodimo konstruktor inicijalizator, koji je sintaksno drugaciji od ostalih konstruktora:

Complex (double rl=0, double img=0) : r(rl), i(img) {...}

Ovo je isto kao: r=rl , i=img

Napomena: Veoma paziti kako se konstruktorima dodeljuju podrazumevane vrednosti! Primera radi navescemo dva konstruktora sa razlicitim ulaznim parametrima koji imaju podrazumevane vrednosti:

Complex (int x, double y=2) {...}
 Complex (int a, int b=0) {...}

Tada pri pozivu:

Complex z(1);

ce doci do greske, jer prevodilac nece znati koji konstruktor da pozove!

Konstruktor kopije

Prepoznaje se po tome sto ima jedan parametar koji je referenca:

Complex (Complex &z) {...}

Pojavljuje se u dvostrukoj ulozi:

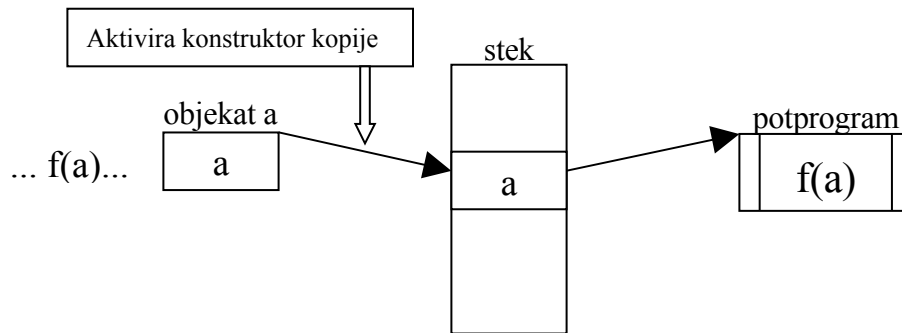
- kao normalan konstruktor

- sa specifinom ulogom

Njegova specifina namena je kod prenosa objekta kao parametra (ali ne kod prenosa po adresi, vec po vrednosti, kada se ceo objekat smesta na stek). Konstruktor kopije je zaduzen da ceo objekat kopira na stek. Posto je ovo cesto neizbežno, svaka klasa u sebi sadrzi ugradjeni konstruktor kopije (to je u stvari obican ugradjeni konstruktor). I kada se prosledjuje i kada se vraca objekat kao parametar, poziva se konstruktor kopije kao posrednik. Ponekad se on moze izbeci i to se cini prenosom objekta po referenci:

```
double f(Complex &q) {...}
```

Prenos objekta po vrednosti:



Vazno:

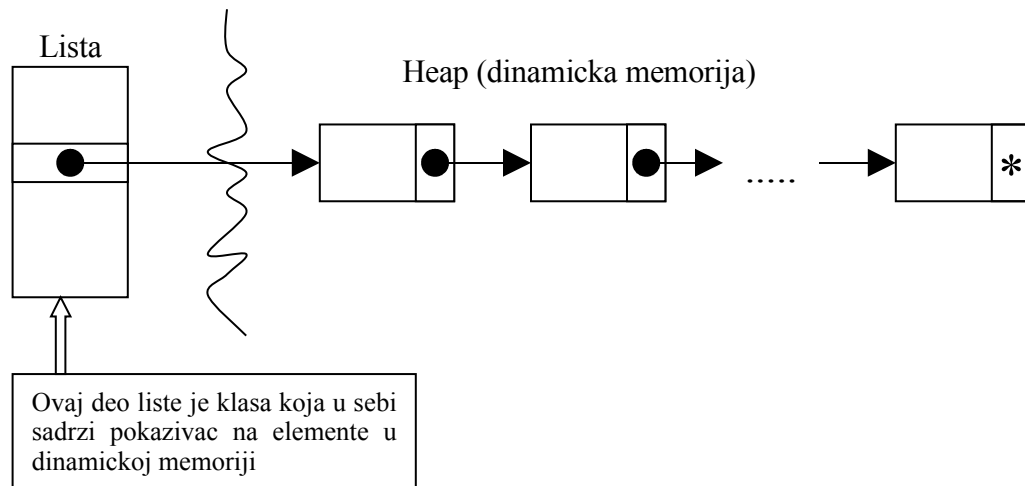
Konstruktor kopije se mora programirati kada klasa sadrzi elemente u dinamickoj memoriji (HEAP)!

Objasnjenje: Pri pozivu funkcije koja kao parametar prima objekat po vrednosti (a ne po adresi), prvo se poziva ugradjeni konstruktor koji ce konstruisati osnovni objekat bez dinamickih elemenata, a zatim se poziva konstruktor kopije koji smo mi isprogramirali da konstruisanom osnovnom objektu dodeli dinamicke elemente! Zato je jako vazno paziti na sledecu napomenu.

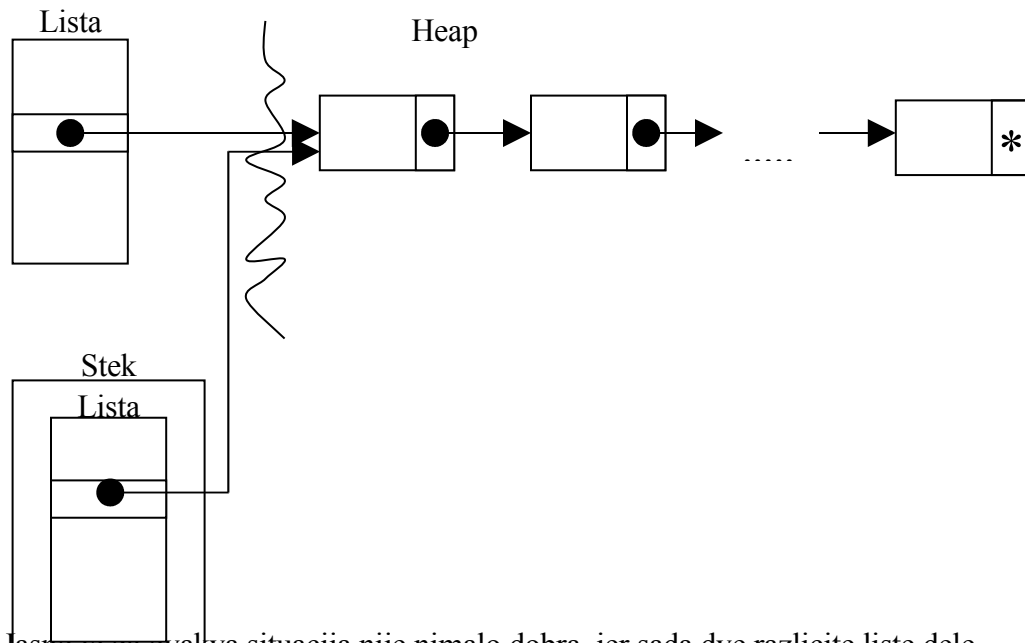
Napomena: Ako smo mi napisali konstruktor, tada ugradjeni konstruktor vise NE VAZI, pa se nece moci pozvati u funkciji koja kao parametar prima objekat po vrednosti (ili bilo kom drugom slucaju kopiranja na stek) i doci ce do greske! Zato je preporucljivo da ako smo napisali neki konstruktor, napisemo i ugradjeni konstruktor (to je onaj bez parametara), da bi se mogao pozvati u pomenutom slucaju.

Sledeci primer pokazuje zasto je u slucaju postojanja dinamickih elemenata potrebno programirati konstruktor kopije:

Primer: Uzmimo za primer da je lista realizovana preko klase:



Problem nastaje kada konstruktor kopije treba ovakvu listu da premesti na stek. Ugradjeni konstruktor kopije pravi plitku kopiju, sto znaci da ce on kopirati samo objekat bez dinamicke memorije i stvorice sledecu situaciju:



Jasno je da ovakva situacija nije nimalo dobra, jer sada dve razlicite liste dele elemente u dinamickoj memoriji! U ovom slucaju ako mi izmenimo elemente jedne liste, automatski ce se izmeniti i elementi druge liste! Ovo je veoma nepozeljno i resenje je da se konstruktor kopije isprogramira da pravi duboke kopije, sto znaci da na stek stavlja i klasu i njenu dinamicku memoriju (duboka kopija = plitka kopija + Heap).

Destruktor

To su specijalne metode ciji je zadatak da poniste objekte. Nemaju tip. Destruktor pocinje tildom “~” i ima naziv klase:

```
~Complex() {} //ovo je ugradjeni destruktor
```

Moze se eksplicitno pozvati, npr.

```
z.~Complex();
```

ali se to najcesce ne radi. Obicno se radi automatsko ukljucenje destruktora, koje se vrsi cim objekat izađe izvan opsega. Brisanje objekta sa steka se takodje vrsi destruktorom.

Vazno:

Destruktor se mora programirati kada klasa sadrzi elemente u dinamičkoj memoriji (HEAP)!

Vratimo se na nas primer liste realizovane preko klase. Ako se za takvu klasu bude pokrenuo ugradjeni destruktor, on ce obrisati samo objekat bez elemenata u dinamičkoj memoriji! Dakle, memorija ostaje zauzeta, a mi nemamo mogucnost da joj pristupimo, jer smo pokazivac na prvi element u heap-u obrisali. Zato je potrebno isprogramirati destruktor da prvo obrisuje elemente iz heap-a! Pri zavrsetku programa automatski se poziva prvo programirani destruktor (koji smo napisali da obrisuje dinamicke elemente u slucaju da oni postoje), a zatim se, takodje automatski, poziva ugradjeni destruktor, koji konacno “unisti” objekat.

Destruktor mozemo pozvati i unutar metode date klase, da bi se po potrebi obrisali dinamički elementi objekta i objekat postavio u inicijalno stanje (koje smo mi odredili u destruktoru). To se radi preko pokazivaca na dati objekat THIS:

```
(*this).~Complex();
```

```
ili sa: this->~Complex();
```

Napomena: Veoma paziti na potrebu za programiranjem konstruktora i destruktora, jer je kasnije izuzetno tesko pronaci gde je greska u programu, posto se program ponasa nepredvidljivo i greske su stalno razlicite.

9. Kooperativne funkcije u C++

Nazivaju se jos i prijateljske funkcije. One su slobodne funkcije (dakle, nisu clanovi klase). Funkcije se proglašavaju za prijatelje date klase da bi imali pristup zasticenom (private) delu klase.

```
class K{  
    private:  
        double a;  
    public:  
        friend double h(K);  
};
```

Recju “friend” se daje do znanja da je slobodna funkcija prijatelj klase

U programu se definiše funkcija:

```
double h(K x)
{
    return 2*x.a;
}
```

Da funkciju nismo proglasili za prijateljsku, ovo ne bi radilo jer je podatak član a zaštićen (u delu pod "private:")

Metode neke druge klase se takođe mogu proglasiti za prijateljske. Prijateljskom se čak može proglasiti i klasa, ali su to specijalni slučajevi.

Namena: Svrha prijateljskih funkcija je da rade brže nego da smo u funkciji zvali metodu koja ima pristup podacima članovima (tj. "private" delu) date klase.

10. Pojam i vrste polimorfizama

Definicija: Polimorfizam je kontekstno zavisno ponašanje (odredjene kategorije se u zavisnosti od okolnosti ponašaju drugačije). Polimorfno se ponašaju objekti i funkcije.

Primeri polimorfnog ponašanja:

Promenljive koje se ponašaju polimorfno u C-u:

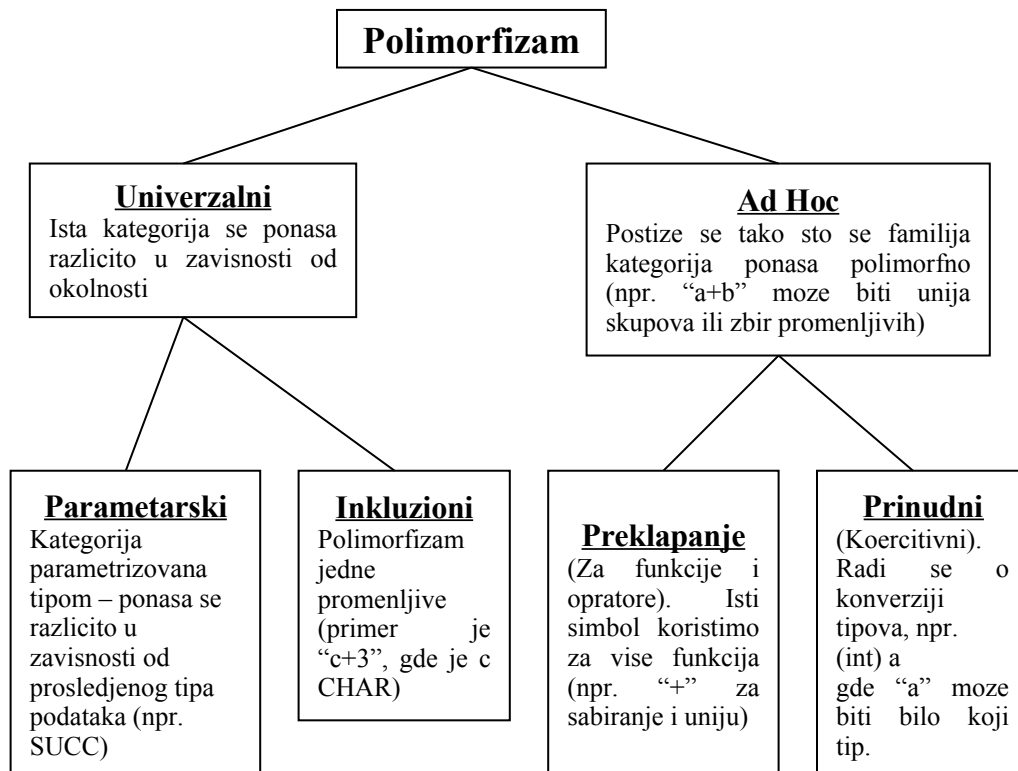
```
char c;
```

```
...
```

```
x = c+3
```

Ovde se "c" ponaša kao int, a može da se ponaša i kao char. Ovo je moguće jer je znakovni tip char član familije celobrojnih tipova int.

Polimorfizam postoji i u potprogramima, npr. u Pascal-u funkcija SUCC (sledebnik) za prosledjen broj vraća broj za jedan veći od prosledjenog, a za prosledjen karakter (npr. "a") vraća sledeći karakter po redu ("b").



U C++ su podržani svi polimorfizmi (jedino u C++ možemo programirati preklapanje!).

11. Preklapanje operatora u C++

Preklapanje funkcija

Preklapanje funkcija u C++ dozvoljava:

- da se u istom programu pojave dve ili više funkcija sa istim nazivom
- da se u okviru klase mogu pojaviti metode sa istim nazivom
- da se u dve različite klase mogu pojaviti metode sa istim nazivom

Jedino ograničenje je u tome da se preklopljene funkcije moraju razlikovati po ulaznim parametrima! (Prevodilac je u prepoznavanje funkcija pored naziva funkcije uključio i njene ulazne parametre). Upozorenje: prevodilac neće moći razlikovati funkcije ako se one razlikuju po povratnim vrednostima! Primera radi, prevodilac ne bi mogao da razlikuje sledeće dve funkcije:

<code>double func(int a);</code>	←	<div style="border: 1px solid black; padding: 5px; display: inline-block;">Ove dve funkcije prevodilac ne bi mogao da razlikuje!</div>
<code>int func(int x);</code>	←	

Napomena: Veoma paziti kako se preklopljenim funkcijama dodeljuju podrazumevane vrednosti! Primera radi, navesćemo dve preklopljene funkcije sa različitim ulaznim parametrima koji imaju podrazumevane vrednosti:

```
int Abs(int i, double x=3.5);  
double Abs(int a);
```

Tada pri pozivu:

```
Abs(1, 5.67);
```

prevodilac poziva prvu funkciju. Ali, pri pozivu:

```
Abs(1);
```

će doći do greške, jer prevodilac neće znati koju funkciju da pozove! Zato bi možda ovde trebalo izbegavati podrazumevane vrednosti.

Pravilo ocuvanja semantike: Funkcije i operatori sa istim nazivom treba da rade isti (“slican”) posao, tj. da imaju istu semantiku. Ovo se u realnosti odnosi na “vrlo približno isti posao”. Na primer, funkcije od kojih jedna dodaje element u skup elemenata nekog tipa, a druga dodaje karakter u string, obe mogu nositi naziv “Dodaj” (ovo nije baš isti posao, ali recimo da je slican).

Napomena: Kada metode u različitim klasama rade isti (slican) posao, one moraju imati isti naziv!

Problem sa objektnim programiranjem je u tome što danas postoji jako puno izradjenih klasa i u njima mnogo metoda i jasno je da ih nije moguće sve pamti,

pa se resursi troše da bi se programeru pojasnilo šta koja metoda u klasi radi. Zato se usvajaju neke opšte metode čiji su nazivi uvek isti! Npr. u Delphi-u sve metode za crtanje nose naziv "Draw".

Preklapanje operatora

Preklapanje operatora je uvedeno da bi se pojednostavili odrađeni izrazi i smanjila mogućnost greske. Na primer, izraz $z=a+b+c+d+e$ gde su promenljive kompleksni brojevi, bismo u našoj klasi Complex napisali ovako:

$z = \text{Add}(a, \text{Add}(b, \text{Add}(c, \text{Add}(d, e))));$

Da je u pitanju neki složeniji izraz (npr. još sa množenjem, oduzimanjem...) pisanje bi se još više zakomplikovalo i zato je dobro u ovakvom slučaju uvesti operator za sabiranje kompleksnih brojeva, a to radimo preklapanjem operatora. Mogli bismo funkciju da nazovemo sa "+" i da je pozivamo sa $z=+(a,b)$; (ovo je operatorska funkcija). Ipak bolje rešenje je:

Complex operator $+(Complex, Complex);$

Pozivanje se tada vrsi sa:

$z = \text{operator } +(a,b);$
 $z = a + b;$

Ovo je isto! Kada se napise $z=a+b$, prevodilac to prevede u $z=\text{operator } +(a,b)$ i pozove potrebnu funkciju

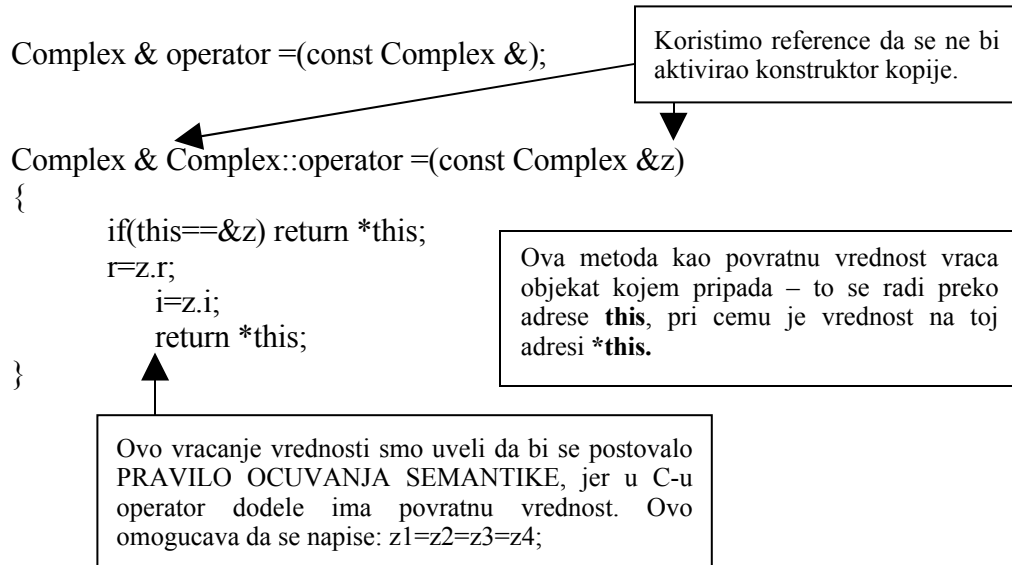
Ogranicenja:

- Ne mogu se preklopiti operatori za standardne tipove.
- Operator zadržava prioritet već postojećeg operatora i tako se grupise (npr. operator "+" koji smo napravili za kompleksne brojeve zadržava prioritete operatora "+" za obično sabiranje standardnih tipova! To recimo znači da će množenje i kod kompleksnih brojeva biti višeg prioriteta od množenja).
- Ne mogu se uvoditi novi simboli, tj. ne možemo uraditi da nadjemo simbol kojem već nije pridružen operator i da mu onda pridruzimo operator (ovo znači da nema uvođenja novih operatora – mogu se samo preklopiti ugrađeni operatori). Razlog je u tome što ako uvedemo novi operator, onda mu moramo odrediti prioritet u odnosu na ostale postojeće operatore, što bi lako moglo izazvati haos i zato je zabranjeno!
- Postoje neki operatori koji se ne mogu preklopiti: ":", ".*", "::", "?:"
- Postoje operatori za koje preklapanje sigurno vazi: "=", "&", ",",
- Operator dodele "=" je glavni kandidat za preklapanje! Ako u klasi ima dinamičkih članova, tada se operator dodele mora preklopiti! (Ovo je neophodno uraditi jer bi on inače samo izjednačio dve klase, ali dinamičke elemente ne bi kopirao, pa bismo imali situaciju da dve klase dele dinamičku memoriju, što je katastrofalno!).

Preklapanje operatora možemo izvršiti na dva načina:

- Preklapanjem operatora kao metode – ovo se radi ako dolazi do izmene podataka clanova klase (iz definicije je jasno da se operator dodele mora preklopiti metodom, jer se tu podacima clanovima objekta dodeljuju nove vrednosti).
- Preklapanjem operatora kao prijateljske funkcije – ovako se preklapa ako ne dolazi do izmene podataka clanova klase.

Primer:



Napomena: Mogli smo napisati i ovako:

Complex operator =(const Complex &);

```
Complex Complex::operator =(const Complex &z)
{
    if(this==&z) return *this;
    r=z.r;
    i=z.i;
    return *this;
}
```

Jedina razlika u odnosu na prethodni primer je u tome sto je ovde povratna vrednost objekat klase Complex, a ne referenca na objekat date klase (u prethodnom primeru je povratna vrednost bila: Complex &). Ovo ce takodje raditi s tom razlikom da se ovde pri prenosu povratne vrednosti aktivira konstruktor kopije (jer se prenosi po vrednosti), dok se u prethodnom primeru pri prenosu po referenci konstruktor kopije nije koristio (jer je referenca po svojoj prirodi adresa). Dakle, koriscenjem reference u prethodnom primeru smo izbegli konstruktor kopije.

Kod operatora dodele potrebno je izvršiti proveru da li je prosledjeni objekat upravo objekat kojem dodeljujemo vrednost. Npr. ako je konstruisan objekat M, u programu smo mogli napisati:

M=M;

U ovom slučaju nema potrebe da se izvršava operator dodele jer se objekat dodeljuje sam sebi! Da li je u pitanju isti objekat, možemo znati proverom adrese prosledjenog i objekta kojem dodeljujemo vrednost – ako su one iste prekida se funkcija i to je sve uradjeno uslovom:

```
if(this==&z) return *this;
```

Objasnjeno u vezi sa pokazivacem na objekat “this”: “this” je podatak član koji postoji u svakom objektu i on se ne zadaje. U okviru objekta “this” ima tip pokazivaca na tu vrstu objekta. Na primer, u našoj klasi Complex, njegova definicija bi bila: Complex *this; (dakle, bio bi pokazivac na objekat klase Complex). Tada bi za konstruisan objekat :

```
Complex z();  
njegova adresa bila:  
z.this
```

Preklapanje relacionih operatora

Oni su dobri kandidati za preklapanje (“==”, “!=”, “<”, “>”, ...). Relacioni operatori se takodje mogu preklopiti metodom ili prijateljskom funkcijom.

Preklapanje operatora “==”: Ako operator “==” preklopimo metodom i ako su x i y objekti, tada je izraz x==y ekvivalentan sa: x.operator==(y); Vidimo da ovde nije sve jedno koji operand pisemo prvo jer metoda ima parametar, a to je ovde objekat y, dok bi matematički trebalo da je sve jedno! Iz ovog razloga i zato što nema dodele vrednosti, ovde se preklapanje vrši prijateljskom funkcijom:

```
friend int operator==(const Complex &, const Complex &);
```

```
int operator==(const Complex &z1, const Complex &z2)  
{  
    return ((z1.r==z2.r) && (z1.i==z2.i));  
}
```

Za klasu Complex nećemo uvesti relacije “>” i “<” jer one nemaju smisla. Svakako ćemo preklopiti aritmetičke operatore.

Preklapanje aritmetičkih operatora

Primer sabiranja “+”: sabiranje ćemo preklopiti prijateljskom funkcijom jer se pri sabiranju ne menjaju postojeće vrednosti, već se generise rezultat. Postoje dva načina:

a) Koriscenje privremenog objekta:

```
friend Complex operator+(const Complex &, const Complex &);
```

```
Complex operator+(const complex &z1, const Complex z2)  
{
```

```

Complex w;
w.r = z1.r + z2.r;
w.i = z1.i + z2.i;
return w;
}

```

Privremeni objekat – on postoji samo unutar funkcije, pa se mora preneti po vrednosti, ne moze po adresi! Pogledaj sledeci primer!

Napomena: Ovo je primer ceste greske! Da smo prethodnu funkciju napisali na sledeci nacin, ona ne bi radila ispravno (iako bi prosla kompajliranje!) :

```

friend Complex & operator +(const Complex &, const Complex &);

Complex & operator +(const complex &z1, const Complex z2)
{
    Complex w;
    w.r = z1.r + z2.r;
    w.i = z1.i + z2.i;
    return w;
}

```

Razlika u odnosu na prethodno je sto je povratna vrednost referenca tj. adresa!

Ovo neće raditi ispravno, jer je povratna vrednost referenca, tj. adresa, a u ovom slucaju to je adresa objekta w koji je kreiran na steku (jer se nalazi u potprogramu) – dakle, ova memorija nije zauzeta i u toku programa se moze menjati, pa su posledice nepredvidive! Zato povratna vrednost mora biti objekat, a ne adresa! Velika nezgoda je u tome sto prevodilac neće prijaviti gresku, tj. proslo bi kompajliranje!

- b) Vracanje bezimenog objekta: mozda bolje resenje jer nema privremenog objekta:

```

friend Complex operator +(const Complex &, const Complex &);

Complex operator +(const complex &z1, const Complex z2)
{
    return Complex(z1.r+z2.r, z1.i+z2.i);
}

```

Preklapanje operatora “++” i “--”

Ovo su unarni operatori, pa ce biti metode. Obe se mogu realizovati kao inline ili outline bez ikakvog ogranicenja. Primer realizacije:

```

const Complex & operator ++()
{

```

Ova funkcija je inline. Ona se poziva sa:
++z;

```

    ++r;
    ++i;
    return *this;
}

```

```

const Complex::operator ++(int k)
{
    Complex w(r,i);
    r++;
    i++;
    return w;
}

```

Ova funkcija je outline. Parametar "int k", ovde služi samo tome da prevodilac može da razlikuje ovu funkciju od prethodne. Ova funkcija se poziva sa:
z++;

Preklapanje operatora "()" i "[]"

Operator "()" omogućuje da neki objekat bude pozvan sa parametrom. Na primer, ako napravimo klasu Polinom i uvedemo operator "()":

```

double Polinom::operator()(double x)
{
    //racunanje vrednosti polinoma za x
}

```

Glavni program:

```

void main(void)
{
    Polinom p;
    double r,y;
    r = p(2*y-1);
}

```

Ovde je "p" objekat, su zagrade METODA (iako se između otvorene i zatvorene zagrade nalazi izraz sa promenljivama). Naizgled je ovde objekat p pozvan sa parametrima, a u stvari je samo pozvana njegova metoda "()".

Slično je i za operator "[]", samo se koristi referenca:

```

Tip& T::operator [](argument) {...}

```

Poziv:

```

...=S[i];
S[i]=...;

```

Operator "->" se može preklapati, ali se najčešće ne koristi.

Preporuke o preklapanju operatora:

a) Pravilo o održanju semantike je obavezno!

- b) Zgodno je preklopiti operatore kada se ocekiju slozeniji izrazi koji koriste metode date klase
- c) Klasa se treba orjentisati ili metodski ili operatorski, tj. da sadrzi ili samo metode ili samo operatore, a nikako oba jer to zbunjuje i usporava
- d) Preklapanje ne raditi sem ako nije bas zgodno i povoljno!

12. Konverzija tipova u C++

Konverzija u klasu

Moze se vrsiti iz tipa u klasu ili iz klase u klasu.

```
class NazivKlase{
public:
    NazivKlase (tip x) {...}
    NazivKlase (klasa &u) {...}
};
```

Konstruktor za konverziju tipa u klasu

Konstruktor za konverziju klase u klasu

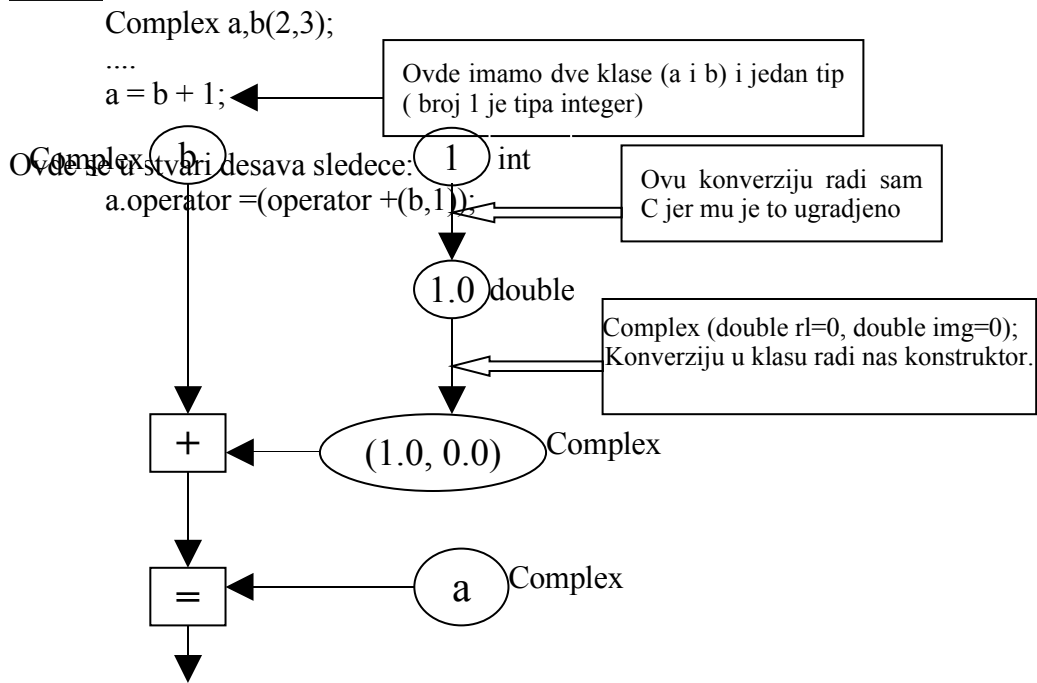
Ako se funkcija koja kao ulazni parametar ima klasu:

```
tip f(klasa a) {...}
pozove na sledeci nacin:
tip z; //npr. double z;
...
f(z); //poziv funkcije "f";
```

doci ce do konverzije tipa u istu klasu kao sto je klasa koja je ulazni parametar funkcije f. Ovo se sve desava preko konstruktora i zato se on mora napisati.

Napomena: ako su klase u vezi nasledjivanja, onda na scenu stupaju sasvim drugacije metode. Ako pozovemo funkciju sa nekom drugom klasom kao parametrom, treba se izvrsiti konverzija iz klase u klasu, sto se takodje radi preko konstruktora s tom razlikom sto se ovde uvek koristi referenca! (pogledaj kako smo napisali konstruktor za konverziju klase u klasu u gornjem primeru).

Primer:



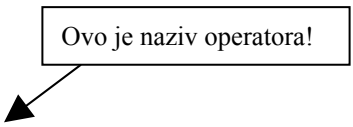
Konverzija iz klase

Pod ovim se podrazumeva konverzija iz klase u tip i vrsi se preklapanjem operatora konverzije.

Primer:

```
class NekaKlasa{  
    public:  
        operator int() {....}  
};
```

Ovo je naziv operatora!



Pozivanje bi glasilo:

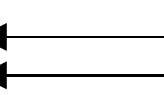
```
NekaKlasa a;
```

```
....
```

```
... (int)a ...
```

```
... int(a) ...
```

Ovako se moze pozvati konverzija iz klase u tip (u ovom slucaju tip "int")



Napomena: Ako radimo konverziju, tada ili raditi konverziju iz tipa u klasu ili konverziju iz klase u tip, ali nikako obe! Ovo je zato sto, ako postoje obe konverzije, moze doci do slucaja kada prevodilac ne zna koju konverziju da koristi, npr. u izrazu: "c + 1.0" on ne bi znao da li da konvertuje "c" ili "1.0".

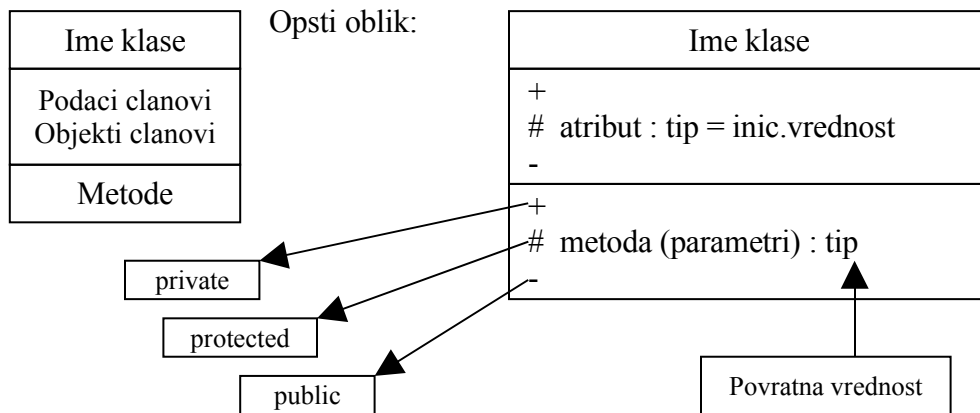
13. Tipovi veza izmedju klasa

Odnosi entiteta se modeluju vezama objekata klasa. Izdvajamo sledece karakteristicne veze:

- Kompozicija – najcvrsca veza, npr. "fakultet se sastoji iz odseka", "fabrika se sastoji iz sektora" – dakle, ova veza oznacava konstruktivne elemente, s tim da se ukidanjem objekta sadrzaoca ukidaju i komponentni objekti (ako se ukine fakultet, automatski nestaju svi odseci, a tako i sektori u fabrici ne mogu postojati nezavisno od fabrike, vec i oni nestaju njenim ukidanjem)
- Agregacija – ova veza nije tako cvrsta kao kompozicija. Na primer, to je veza: "nastavna grupa i student", ili "voz i vagon". Ova veza je labavija jer se ukidanjem objekta sadrzaoca ne ukidaju komponentni objekti (ako se ukine nastavna grupa koja se sastoji od studenata, to ne znaci da studenti vise ne postoje ili ako se rasformira voz, vagoni i dalje postoje).

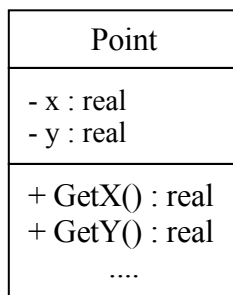
- Nasledjivanje – ono modeluje GENERALIZACIJU – SPECIJALIZACIJU tj. odnos opste – pojedinačno. Na primer: “funkcija je relacija”, “lav je macka” (uociti da obrnuto ne mora da vazi!).
- Asocijacija – njom se formalno nezavisne klase dovode u vezu po znacanju. Na primer, za klase “Nastavnik” i “Predmet” bi veza bila “predaje”. Ove veze je cesto teze uociti od prethodnih.
- Veze zavisnosti – ostale veze, npr. kao veza “koriscenja” – ako u nekoj metodi klase A postoji parametar klase B, onda kazemo da “A koristi B”.

UML (Unified Modeling Language) je sredstvo za modelovanje objektnog softvera i zasnova se na dijagramima, tj. ima graficki nacin prikazivanja (pored uvoda koji je tekstualan). Kljucni su dijagrami klasa:



Napomena: za podatke clanove i objekte clanove se u UML-u koristi naziv “atributi”, sto i nije bas ispravno. Takodje, UML umesto tipa “double”, koristi naziv “real”.

Primer: prikaz klase Point:



Problem nastaje sa prikazivanjem konstruktora jer mu naziv zavisi od programskog jezika koji koristimo, a drugi problem je sto ovde nisu navedene celokupne klase, vec samo delovi, pa se za klase koje su veoma vazne pise potpun sadrzaj van dijagrama klasa, dok se klase u dijagramu cesto predstavljaju samo svojim nazivom.

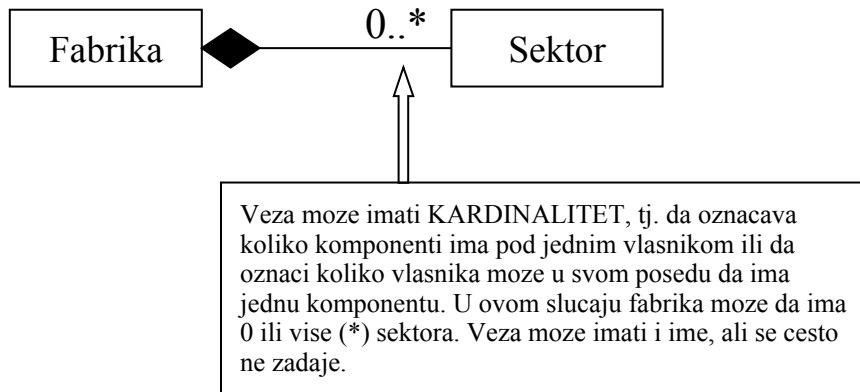
14. Agregacija

Kompozicija – ukidanjem celine se ukidaju delovi

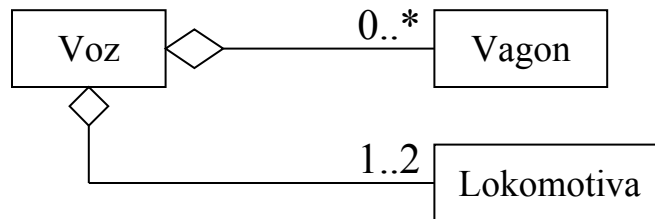
Agregacija – ukidanjem celine se ne moraju ukidati delovi

Iz ovog razloga se kaže da je kompozicija deo agregacije, pa se agregacija deli na kompoziciju i “cistu agregaciju”.

Kompozicija je najcvrsca veza klasa i prepoznaje se po zivotnom veku objekta. Ovde se za deo koristi naziv KOMPONENTA, a za celinu naziv VLASNIK. Dakle, ovde zivotni vek komponente zavisi od zivotnog veka vlasnika. Kompozicija se oznacava na sledeci nacin:



Cista agregacija je veza u kojoj deo može postojati bez celine, tj. zivotni vek dela ne zavisi od celine. Često se realizuje preko liste objekata koji pripadaju vlasniku. Graficki se predstavlja:

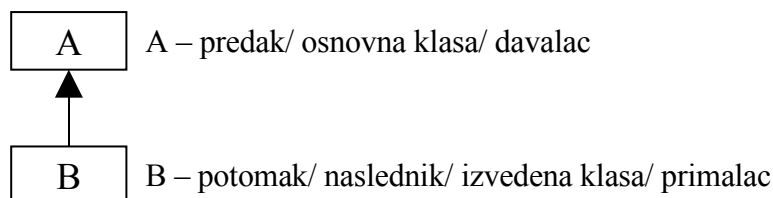


15. Definicija i osobine nasledjivanja

Definicija:

- Nasledjivanje modeluje odnos generalizacija- specijalizacija.
- Nasledjivanje u zajednici sa modularizacijom omogućava višekratnu upotrebu
- Nasledjivanje (Inheritance) je preuzimanje kompletnog sadržaja druge klase uz mogućnost dodavanja članova i modifikaciju metoda.

U UML-u se predstavlja na sledeci nacin:



Napomena: Prilikom preuzimanja klase tj. nasleđivanja, izvorni kod pretka ne postoji, tj. ne sme se modifikovati jer mi imamo samo prevedeni (masinski) kod. Predak se nalazi u modulu.

Osobine:

- Mogućnost modifikacije metoda (redefinisanje) – ako neka metoda iz klase A ne odgovara klasi B, metoda se u klasi B može modifikovati, tj. ponovo definisati.
- Proširivanje – u klasi potomka se mogu dodati metode i podaci.
- Tranzitivna osobina – ako klasa C nasledjuje klasu B, a klasa B nasledjuje klasu A, tada klasa C nasledjuje klasu A. Ova osobina omogućava hijerarhiju klasa.
- Visestruko nasledjivanje – jedna klasa može imati više od jednog pretka.

Napomena: naslednik (klasa potomka) nije klijent, klijent samo koristi uslugu neke klase.

16. Demetrin zakon i zakon supstitucije

Demetrin zakon: Metode date klase ne smeju ni na koji nacin da zavise od strukture bilo koje druge klase osim neposrednog pretka.

Zakon supstitucije: Ako za svaki objekat “s” iz klase “S” postoji objekat “t” iz klase “T”, takav da se proizvoljni program nad “T” jednako ponasa kada se “t” zameni sa “s”, tada je “S” potomak “T”.

Drugacije receno, ako se predak zameni potomkom, u programu nista ne sme da se promeni. U ovom slucaju se potomak ponasa polimorfno, jer se ponasa kao predak.

17. Realizacija nasleđivanja u C++

```
class A{           //klasa A je predak i na nju nemamo uticaja
    private://ovaj deo je zatvoren za klijente, naslednike, funkcije...

        ....
    protected:    //nivo zastite namenjen naslednicima. Ovaj deo je zatvoren
        ....      // za klijente, ali je otvoren za naslednike
    public:

};
```

Klasa B nasledjuje klasu A:

```
class B : public A{    //sve sto postoji u A, nalazi se i u B
```

```

....
//redefinisani deo
....
};

```

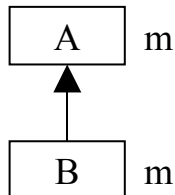
Ovaj deo govori kako je klasa nasledjena (da li je pod public, protected ili private)

Nasledjivanje klase	Clan osnovne klase	
	public	protected
public	public	protected
protected	protected	protected
private	private	private

Na primer, iz tabele se vidi da ako neku klasu nasledimo kao public, tada podaci nasledjene klase pod public ce u klasi naslednici takodje biti public, dok ce oni pod protected u klasi naslednici takodje biti protected.

Sta se ne nasledjuje: Konstruktor, destruktor, operator dodele i prijateljske funkcije se ne nasledjuju.

Redefinisanje metode: izvodi se tako sto sto se metoda ponovo napise u klasi potomka. Nova metoda ne mora da ima iste parametre, niti tip parametara – sve moze da se razlikuje sem imena.



Ako klasa B nasledjuje klasu A u kojoj postoji metoda “m”, a u klasi B je ona redefinisana, tada se metodi “m” u klasi A moze pristupiti preko nasledjivanja iz klase B sa:

A::m(...);

Primer: realizacija klase kvadrat preko nasledjivanja od klase pravougaonik.

//Naziv datoteke: “PRAVOUG.H”

```

class Pravougaonik{
    protected:
        double s1,s2;
    public:
        Pravougaonik(double a, double b) : s1(a), s2(b) {}

```

```

        double A() const {return s1;}
        double B() const {return s2;}
        double Obim() const {return 2*(s1+s2);}
        double Povrsina() const {return s1*s2;}
};

//Naziv datoteke: "KVADRAT.H"

#include "pravoug.h"

class Kvadrat::public Pravougaonik{
public:
    Kvadrat(double a) : Pravougaonik(a,a) {}
    double Stranica() const {return s1;}
    double Dijagonala() const {return s1*sqrt(2);}
};

```

U ovom primeru klasa Kvadrat sve nasledjuje od klase pravougaonik. Podaci clanovi klase Kvadrat su :

double s1,s2;

Metode klase Kvadrat su:

```

double A() const {return s1;}
double B() const {return s2;}
double Obim() const {return 2*(s1+s2);}
double Povrsina() const {return s1*s2;}
Kvadrat(double a) : Pravougaonik(a,a) {} //konstruktor
double Stranica() const {return s1;}
double Dijagonala() const {return s1*sqrt(2);}

```

} Nasledjene od klase Pravougaonik

Metodu "Stranica" smo uveli da ne bi doslo do zabune kod trazenja velicine stranice kvadrata, jer postoje vec dve metode (A i B) koje ce vratiti isti rezultat. Primeri pozivanja ovih metoda:

//Naziv datoteke: "PROBA_KV.CPP"

```

#include "kvadrat.h"

void main(void)
{
    Kvadrat k(2);

    cout<<"Obim "<<k.Obim()<<"Povrsina "<<k.Povrsina()
    <<"Stranica "<<k.Stranica()<<"Dijagonala "<<k.Dijagonala()
    <<endl;
}

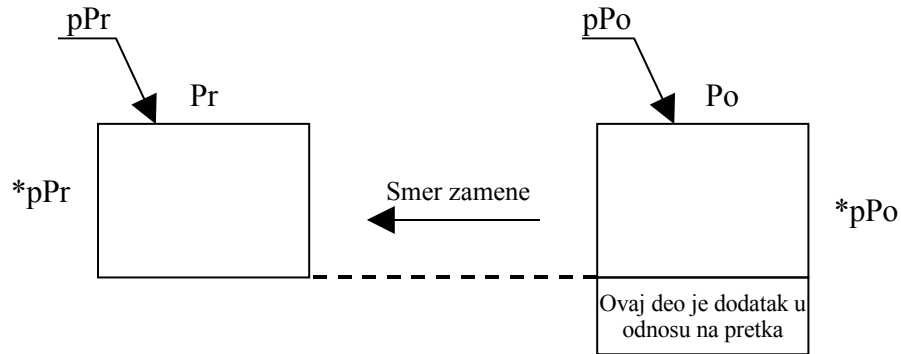
```

18. Inkluzivni polimorfizam u C++

Recimo da imamo klase i objekte:

Predak Pr; //nalazi se na adresi pPr i proziva sa *pPr
Potomak Po; //nalazi se na adresi pPo i proziva sa *pPo

Graficki predstavljeno:

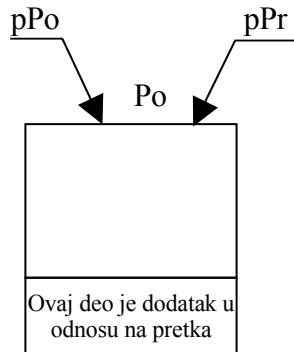


$Pr = Po$; //ovo je dozvoljeno – predak zamenjujemo potomkom

U zameni jedino dolazi u obzir da se predak zameni potomkom! Ova zamena se mogla izvršiti i preko pokazivaca:

$*pPr = *pPo$; //isto sto i $Pr = Po$;

$pPr = pPo$; //ovo je takodje dozvoljeno (ovo je polimorfizam) i desava se sledece:



Ovde imamo situaciju da dva pokazivaca pokazuju na isti objekat, tj. isti memorijski prostor. Postavlja se pitanje kako ovo funkcioniše, jer je pPr pokazivac na klasu pretka, a sada pokazuje na objekat iz klase potomka. Ovo funkcioniše tako sto kada se napise:

$pPr = pPo$;

izvrsava se TYPECAST:

$pPr = (\text{Predak } *) pPo$;

Dakle, zbog izvršenog typecast-a se pPr i dalje ponasa kao pokazivac na predak! Zato ako pokusamo preko pPr pristupiti necemu sto postoji u potomku, a cega nema u pretku, bice prijavljena greska!

Zakon supstitucije: Ako za svaki objekat “s” iz klase “S” postoji objekat “t” iz klase “T”, takav da se proizvoljni program nad “T” jednako ponasa kada se “t” zameni sa “s”, tada je “S” potomak “T”.

Drugacije receno, ako se predak zameni potomkom, u programu nista ne sme da se promeni. U ovom slucaju se potomak ponasa polimorfno, jer se ponasa kao predak. Ista situacija je i kod zamene formalnog parametra stvarnim:

```
tip f(Predak &pr) {...}
```

Sada se funkcija moze pozvati sa potomkom:
Potomak z;

```
....  
f(z);
```

Da bi se funkcija mogla pozvati sa potomkom, ovde je neophodno staviti referencu.

Primer:

```
class Predak {  
    protected:  
        int podatak;  
    public:  
        Predak() {podatak=0;}  
        void Set(int i) {podatak=i;}  
        int Get() {return podatak;}  
};
```

Metodu Set cemo proglasiti za VIRTUELNU (pogledaj sledeci naslov)

```
class Potomak : public Predak {  
    protected:  
        int backup;  
    public:  
        void Set(int i) {backup=podatak; podatak=i;}  
        void Restore() {podatak=backup;}  
};
```

Ova klasa ima dva podatka clana (“podatak” i “backup”)

Redefinisemo metodu Set

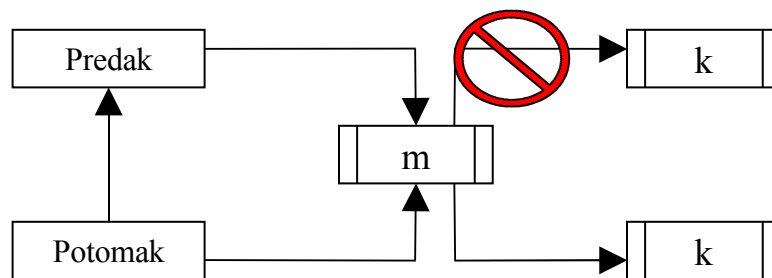
U ovom primeru u klasi Potomak postoje sledeci podaci clanovi: podatak i backup i sledece metode: Set (koja nije preuzeta vec je redefinisana), Get i Restore. Razmotrimo sledeci slucaj:

```
Predak *pPr;  
Potomak *pPo;  
....  
pPo->Set(1);  
pPr=pPo; //sada i pPr i pPo pokazuju na isti objekat iz klase Potomak  
pPr->Set(2); //ovde ce se aktivirati metoda iz pretka!!!
```

Vazno: Vidimo da iako smo namestili da pPr pokazuje na objekat klase Potomak, pri izvršenju naredbe: pPr->Set(2);, pozvace se metoda Set iz klase Predak, a to je zato sto je pPr pokazivac na klasu Predak! Nas cilj je bio da se pokrene metoda Set iz klase Potomak (jer se primenom metode iz klase Predak na objekat klase

Potomak u kojoj postoji redefinisana ova metoda, krši princip nasleđivanja). To se nije dogodilo iako pPr u sebi sadrži adresu objekta iz klase Potomak! (Kao što smo već nagovestili, za ovo je kriv typecast koji se izvršava naredbom: pPr=pPo;). Da bi se u ovom slučaju ipak pozvala metoda iz klase Potomak, uvodimo virtualne funkcije.

19. Virtualne funkcije u C++



Neka u klasi Predak postoje metoda “k” i metoda “m” koja koristi metodu “k” i neka je u klasi Potomak, metoda “k” redefinisana. Ako se iz objekta klase Potomak pozove metoda “m”, mi očekujemo da se u “m” pozove redefinisana metoda “k”, ali se ovo neće dogoditi, već će se pozvati metoda iz klase Predak! Ovo se desava jer je u metodi “m” ugrađen skok na metodu “k” i ovaj problem se mora ispraviti u samom prevodiocu! To se radi uvođenjem virtualnih metoda. Uz svaki objekat se vezuje jedna tabela (V-TABELA), koja sadrži spisak adresa svih virtualnih metoda u tom objektu (u našem primeru će virtualna metoda biti “k”). Kada se pozove virtualna metoda, prevodilac pristupa V-tabeli datog objekta, očitava adresu metode i tek zatim pristupa pročitanoj adresi. Jasno je da su zbog ovog citanja tabele, virtualne funkcije sporije od običnih. Tabele adresa u objekat upisuje konstruktor.

Definicija:

virtual tip Ime_Metode(parametri) {...}

Ova definicija se upisuje u Predak, dok se u Potomku više ne piše “virtual” – ako je metoda virtualna u pretku, ona ostaje virtualna u svim potomcima!

Napomena: Ako se za metodu očekuje da redefinisana u potomku, ona se proglašava virtualnom. Koje će se metode kasnije redefinisati, nije tesko predvideti.

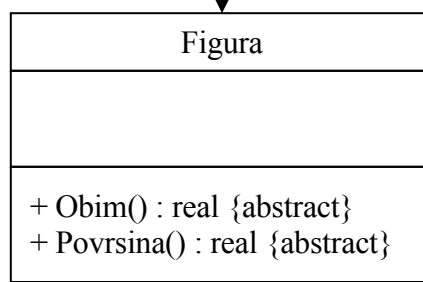
Napomena: Kada se redefinisuje virtualne metode, one moraju imati isti prototip!

Rezime: Kada se poziva redefinisana metoda koja je virtualna, biće pozvana ona iz potomka, dok ako je obična metoda, biće pozvana iz pretka.

Apstrakcije (nepotpune) klase

+ Povrsina() : real

Imaju smisla samo u kontekstu nasleđivanja, kada ima virtualnih metoda.



```

class Figura{
    public:
        virtual double Obim() = 0;
        virtual double Povrsina() = 0;
};

class Krug{
    private:
        double r;
    public:
        double Obim() {return 2*r*3.14;}
        double Povrsina() {return r*r*3.14;}
};

class Pravougaonik{
    protected:
        double a,b;
    public:
        double Obim() {return 2*(a+b);}
        double Povrsina() {return a*b;}
};

```

Ove metode su apstraktne, tj. nepotpune jer nisu do kraja definisane – ne znamo sta rade, pa je zato i klasa apstraktna.

Napomena: Apstraktne metode moraju biti virtuelne. Klasa koja sadrzi barem jednu apstraktnu metodu je apstraktna klasa. Apstraktna klasa se ne moze instancirati, tj. ne mozemo kreirati objekat apstraktne klase jer ima metode koje nisu do kraja definisane i njihovo pozivanje bi dovelo do greske.

```

Figura *pF;
Pravougaonik *pP;
Krug *pK;
....
pF = pP;

```

```

pF->Povrsina(); //ovde ce biti pozvana metoda Povrsina iz klase Pravougaonik
....
pF = pK;
pF->Povrsina(); //ovde ce biti pozvana metoda Povrsina iz klase Krug

```

Vidimo da jedna ista naredba: `pF->Povrsina()`; poziva dve razlicite funkcije (u dve razlicite klase) u zavisnosti od toga na koju klasu pokazuje pokazivac `pF`. Ova osobina je veoma zgodna. Primera radi, moze da se upotrebi za pisanje funkcije ciji je ulazni parametar referenca (ili pokazivac) na apstraktnu klasu `Figura`, sto ce omoguciti da funkciju pozovemo sa objektom iz bilo koje od klasa `Krug` ili `Pravougaonik` kao ulaznim parametrom:

```

void Prikaz(const Figura &rF)
{
    cout<<"Obim"<< rF.Obim()
        <<"Povrsina"<< rF.Povrsina()
        <<endl;
}

```

Ako funkciju pozovemo sa:

```

Krug &rK;

```

```

....

```

```

Prikaz(rK);

```

bice ispisani obim i povrsina kruga, a za poziv:

```

Pravougaonik &rP;

```

```

....

```

```

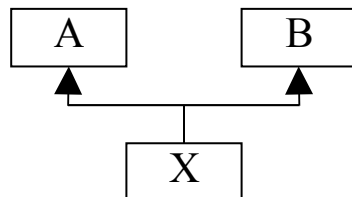
Prikaz(rP);

```

ce biti ispisani obim i povrsina pravougaonika.

20. Visestruko nasledjivanje u C++

Postoji samo u C++ (u C++ je ugradjeno, mada se u ostalim programskim jezicima moze konstruisati). Visestruko nasledjivanje je nasledjivanje od dva ili vise predaka (ne moze se smatrati da je jednostruko nasledjivanje specijalan slucaj visestrukog, jer se znatno razlikuju).



Sustinski problem je u tome sto se u realnosti veza visestrukog nasledjivanja retko pojavljuje, npr. amfibija je drumsko i recno vozilo. Primer loseg koriscenja visestrukog nasledjivanja: klasu Vozac realizovati kao naslednika klase Covek i Automobil (pogresno, jer bi to znacilo da vozac ima tockove, volan... Pravilno bi bilo realizovati klasu Vozac kao naslednika klase Covek, sa dodatkom metode "vozi").

Tehnicki problem: klase A i B od kojih se nasledjuje mogu imati metode sa istim imenom (sto je cak vrlo verovatno, jer se koristi princip ocuvanja semantike). Dakle, ako klase A i B obe imaju metode sa istim nazivom (npr. "m"), pitanje je od koje klase ce potomak preuzeti metodu "m". Jedno od resenja je princip preimenovanja, gde se jedna od metoda preimenuje (sto i nije najbolje resenje, jer metode mogu raditi slican posao i preimenovanjem krsimo pravilo ocuvanja semantike). Resenje u C++ je takvo da se obe metode preuzmu, pa se kod poziva klase X korisnik odlucuje za jednu.

```
X p;
```

```
....
```

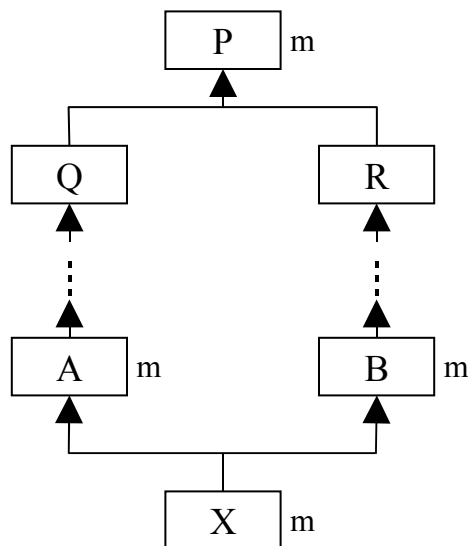
```
p.A::m() //pozivanje metode "m" iz klase A preko objekta p koji pripada  
        // klasi naslednici (X)
```

```
p.B::m() //pozivanje metode "m" iz klase B preko objekta p koji pripada  
        // klasi naslednici (X)
```

Generalno resenje je da se metoda "m" u klasi X redefinisuje i tada nema mogucnosti zabune.

Ponovljeno nasledjivanje

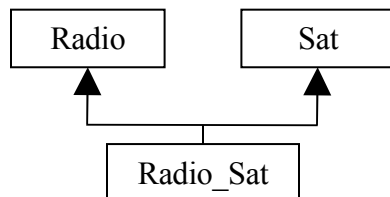
Ovo nije mehanizam, vec problem! Kod visestrukog nasledjivanja moze da se dogodi sledeca situacija:



U prikazanom primeru, metode “m” u klasama A i B su iste (jer su je obe klase nasledile od klase P). Sada u klasi X imamo dve kopije metode “m” koje su iste, pa nastaje problem. Jedno resenje bi bilo da se koristi virtuelno nasledjivanje, tj. da je cela klasa P virtuelna, ali je tesko predvideti da ce ovako nesto biti potrebno! Zato je resenje da se metoda u klasi X redefinise.

Kada koristiti visestruko nasledjivanje: kada se instanca klase naslednice X u svakom trenutku ponasa i kao klasa A i kao klasa B.

Primer: Radio-sat je kombinacija radio prijemnika i sata:



Ako u programskom jeziku ne postoji visestruko nasledjivanje, onda se klase od kojih se nasledjuje stave kao podaci clanovi u klasi koja ih nasledjuje:

```

class Radio_Sat{
public:
    Radio r;
    Sat s;
    ....
};
  
```

U C++ (gde postoji visestuko nasledjivanje) se to radi ovako:

```

class Radio_Sat : public Radio, public Sat{
    ....
};
  
```

21. Genericke klase u C++

One su parametrizovane drugim klasama. Tipican primer su strukture podataka realizovane preko klasa. Na primer, stek ne mozemo realizovati kao klasu jer ne znamo sta ce se u njega smestati (koji tip podataka: int, double, char...), ali zato bi bilo izuzetno zgodno, jer svaki od ovih stekova ima iste metode: Top, Pop, Push, itd.

Mehanizam za realizaciju generickih klasa je TEMPLATE (nacin na koji se u C++ realizuju osobine genericnosti). Nacin na koji se u teoriji prikazuje da je klasa Stack parametrizovana klasom T je: Stack[T].

```

template <class T1, class T2, ..., class Tn>
class A{
  
```

Ovde su navedene klase, ali je to mogao biti i bilo koji tip

```

        //sada se klase T1, T2, ...,Tn koriste kao parametri klase A, tj. kao da joj
        //pripadaju
    };

```

Definisanje outline funkcije:

```

template <class T1, class T2, ..., class Tn>
void A<T1, T2, ...,Tn> :: f(...) {...}

```

Da smo umesto parametra class Tn stavili int K, tada bi K bila konstanta, koja bi dobila vrednost kod poziva.

Objekat genericke klase A se definise:

```

A<S1, S2, ..., Sn> a;

```

Primer: realizacija genericke klase za stek:

```

template <class T, unsigned CAPACITY>
class Stack {
    private:
        T s[CAPACITY]; //niz od CAPACITY(broj) elemenata klase T
        int top;
    public:
        Stack() {top=-1;}
        int Empty() const {return top<0;}
        int Full() const {return top==CAPACITY-1;}
        void Pop() {top--;}
        T Top() const {return S[top];} //povratna vrednost je iz klase T
        void Push(T el) {s[++top] = el;}
};

```

class T koristimo kao tip elementa,
a konstantu CAPACITY koristimo
da odredimo velicinu steka.

Kod definisanja u main-u:

```

Stack <char, 256> Cst; //stek sa karakterima duzine 256 elemenata
Stack <int, 1000> Ist; //stek sa elementima tipa int duzine 1000 elemenata

```

Kontejnerska klasa (npr. stek) se generise isto kao genericke klase. Da bi se ovakva klasa prevela, genericka klasa mora biti kompletna u zaglavlju modula, dakle, kompletan TEMPLATE mora da se nadje u zaglavlju modula.

22. Veza asocijacije

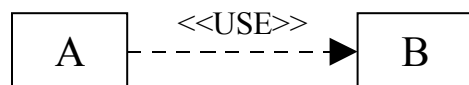
Asocijacija je veza izmedju dve semanticki nezavisne klase, npr.:



“1..2” oznacava da predmet moze imati jednog ili dva nastavnika (posto ovde spisak koji govori koliko predmet moze imati nastavnika ima konacno mnogo brojeva, bice realizovano kao niz). “0..*” oznacava da nastavnik moze predavati ni jedan ili vise predmeta (posto ovde spisak koji govori koliko predmeta moze predavati nastavnik nije tacno ogranicen, bice realizovan kao lista). Strelica na dijagramu govori u kojem smeru se cita ova veza.

U semama baza podataka je ova veza najzastupljenija. Kod asocijacije najvaznija stavka je KARDINALITET (npr. govori KOLIKO predmeta moze da predaje nastavnik). Sama veza se takodje moze realizovati kao klasa (npr. klasa Predaje).

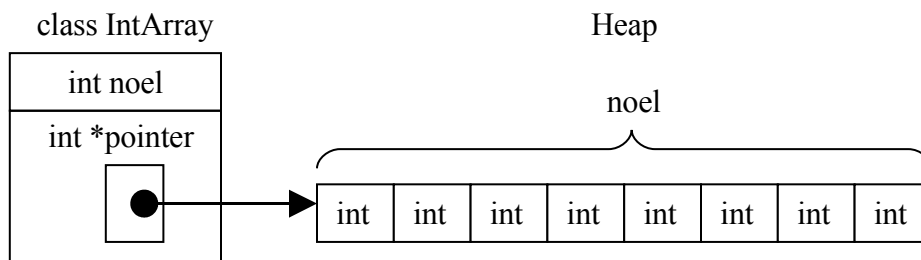
23. Veza zavisnosti



“<<USE>>” oznacava VEZU KORISCENJA. Ako u bar jednoj metodi klase A postoji formalni parametar klase B, tada izmedju ove dve klase postoji veza koriscenja. Strelica se stavlja ka klasi od koje neka klasa zavisi (ovde klasa A zavisi od klase B).

Zadaci:

1. **Zadatak:** Realizovati dinamički niz čiji su elementi celobrojne vrednosti.



```

/*****
ZAGLAVLJE MODULA SA KLASOM NIZA CIJI SU ELEMENTI CELI BROJEVI
  
```

Autor: Danilo Babin, 9715

Naziv datoteke: "INTARRAY.H"

```

*****/
  
```

```

#include <stdlib.h>
#include <iostream.h>
  
```

```

#ifndef IntArray_def
#define IntArray_def

class IntArray{

private:
    int noel;
    int *pointer;

public:
    int Empty();
    int NumberOfElements() {return noel;}
    void Set(int r, int *i=NULL); //menja dimenzije niza
    IntArray(int r, int *i);
    IntArray();
    IntArray(int r);
    IntArray(IntArray &a);
    ~IntArray();
    int& operator [] (int r);
    IntArray& operator =(IntArray &m);
    friend IntArray operator +(IntArray &a1, IntArray &a2);
    friend ostream& operator <<(ostream &o, IntArray &a);
};

#endif

```

```

/*****
MODUL SA KLASOM NIZA CIJI SU ELEMENTI CELI BROJEVI

Autor: Danilo Babin, 9715
Naziv datoteke: "INTARRAY.CPP"
*****/

```

```

#include "intarray.h"

```

```

int IntArray::Empty()
{
    if(noel==0) return 1;
    else return 0;
}

```

```

int& IntArray::operator [] (int r)
{
    if(r<0 || r>=noel || Empty())
    {
        cout<<endl<<"( operator["<<r<<"] )
        Pogresno indeksiranje elemenata!"<<endl;
        exit(EXIT_FAILURE);
    }
    return pointer[r];
}

```

Preklapanjem operatora indeksiranja omogućeno je pojedinačno prozivanje elemenata i istovremeno je onemogućeno indeksiranje izvan granica niza

Povratna vrednost je referenca, sto ce omogućiti da se napise: a[2] = 5; ili b=a[7]; za: IntArray a(10); int b;

```
}
```

```
void IntArray::Set(int r, int *i)
```

```
{
```

```
    int j;
```

```
    (*this).~IntArray();
```

```
    noel=r;
```

```
    pointer = new int [noel];
```

```
    if(i==NULL) return;
```

```
    else
```

```
    {
```

```
        for(j=0; j<noel; j++) pointer[j] = i[j];
```

```
    }
```

```
}
```

Metoda za promenu dimenzije vec konstruisanog objekta. Parametar r odredjuje velicinu niza, a "i" je pokazivac na staticki niz cije se vrednosti mogu uzeti za vrednosti objekta. Drugi parametar ima podrazumevanu vrednost, tj. metoda se moze pozvati bez njega (pogledaj .H fajl)

```
IntArray::IntArray()
```

```
{
```

```
    noel=0;
```

```
    pointer=NULL;
```

```
}
```

Ovaj konstruktor je podrazumevani ugradjeni.

```
IntArray::IntArray(int r)
```

```
{
```

```
    noel=r;
```

```
    pointer = new int [noel];
```

```
}
```

```
IntArray::IntArray(int r, int *i)
```

```
{
```

```
    int j;
```

```
    noel=r;
```

```
    pointer = new int [noel];
```

```
    for(j=0; j<noel; j++) pointer[j] = i[j];
```

```
}
```

Konstruktor koji od statickog niza pravi objekat

```
IntArray::IntArray(IntArray &a)
```

```
{
```

```
    int i;
```

```
    noel = a.noel;
```

```
    pointer = new int [noel];
```

Konstruktor kopije

```

        for(i=0; i<noel; i++) pointer[i] = a.pointer[i];
    }

```

```

IntArray::~IntArray()
{
    delete [] pointer;
    noel = 0;
    pointer=NULL;
}

```

U destrukturu je veoma vazno da se svi parametri klase postave u inicijalne vrednosti, jer postoji mogucnost da se on eksplicitno koristi (kao u sledecoj funkciji) za brisanje dinamicnih elemenata. Vazno je sve pokazivace postaviti na NULL!

```

IntArray& IntArray::operator =(IntArray &a)
{
    int i;

    if (&a==this) return *this;

    (*this).~IntArray();

    noel = a.noel;

    pointer = new int [noel];

    for(i=0; i<noel; i++) pointer[i]=a.pointer[i];
    return *this;
}

```

Ovde je obavezno uneti zastitu od naredbe m=m; (za: IntArray m(123)), sto se radi proverom adresa objekata:
 if (&a==this) return *this;
 jer bi bez nje u sledecoj naredbi:
 (*this).~IntArray();
 objekat bio izbrisan i kao rezultat bismo na kraju dobili prazan objekat!

```

IntArray operator +(IntArray &a1, IntArray &a2)
{
    int i;
    IntArray w;

    if( a1.noel != a2.noel )
    {
        cout<<endl<<"(operator +) Greska! Nizovi nisu istih dimenzija";
        exit;
    }

    w.noel = a1.noel;
    w.pointer = new int [w.noel];

    for(i=0; i<w.noel; i++)
        w.pointer[i] = a1.pointer[i] + a2.pointer[i];

    return w;
}

```

Operator + se realizuje kao prijateljska funkcija jer ne menja podatke clanove datog objekta

```

ostream& operator <<(ostream &o, IntArray &a)
{

```

```

    int i;

    o<<endl<<"Broj elemenata niza: "<<a.noel
    <<endl<<"Elementi niza su:"<<endl;

    for(i=0; i<a.noel; i++)
        o<<"["<<i<<"]"<<"="<<a[i]<<"  ";

    o<<endl;
    return o;
}

```

Operator << je operator za umetanje na izlazni tok. Koristi se preko objekta "cout" (dakle, ovo nije naredba nego objekat!) koji se automatski konstruise na pocetku svakog programa i on pripada klasi "ostream" koja se nalazi u modulu <iostream.h>. Dakle u operatoru << koji preklapamo, bice prosledjeni "cout" objekat i objekat cije podatke clanove zelimo proslediti na izlazni tok (npr. IntArray ia;), pa bi se mogao pozvati sa: operator<<(cout, ia); ali se to cesce radi sa: cout<<ia; Povratna vrednost ove funkcije je takodje referenca na objekat klase "ostream" i to bas onaj objekat koji smo prosledili kao parametar – dakle, povratna vrednost ce takodje biti objekat "cout". Ovo nam omogucava da napisemo npr: cout<<"neki tekst"<<ia<<endl<<"kraj";

```

/*****
    TEST PROGRAM MODULA SA KLASOM NIZA CIJI SU ELEMENTI CELI BROJEVI

    Autor: Danilo Babin, 9715
    Naziv datoteke: "T_INTARR.CPP"
    *****/

```

```

#include "intarray.h"
#include <conio.h>

void f(IntArray a)
{
    cout<<endl<<"funkcija f:"<<a;
}

void main(void)
{
    int niz1[10], niz2[10], niz4[6], i;

    clrscr();

    cout<<"Inicijalizacija niza a1 od obicnog niza:"<<endl;

    IntArray a1(10, niz1);

    cout<<endl<<"Elementi pomocnog niza su:"<<endl;
    for(i=0; i<10; i++) cout<<"["<<i<<"]"<<"="<<niz1[i]<<"  ";
}

```



```

cout<<endl<<a1<<"Element na poziciji [4] je "<<a1[4]<<endl;

IntArray a2(10, niz2);

cout<<endl<<"Inicijalizacija niza a2:"<<endl<<a2;

cout<<endl<<"Zbir elemenata na pozicijama [3] = "<<a1[3]+a2[3];
a2=a1+a2;
cout<<endl<<endl<<"Zbir nizova a2 = a1 + a2 je:"<<a2;
getch();

clrscr();
cout<<"Provera konstruktora kopije:"<<endl;

IntArray a3(a1);

cout<<endl<<"Inicijalizovan je niz a3 koji je isti kao a1:"
<<endl<<"Niz a1:"<<a1<<endl<<"Niz a3"<<a3;

f(a3);

getch();
clrscr();

cout<<"Provera metode za promenu velicine niza:"<<endl<<endl
<<"Od niza a3 pravimo novi niz duzine 4:"<<endl;
a3.Set(4);
cout<<"Niz a3:"<<a3;

cout<<endl<<"Od niza a3 pravimo niz duzine 6 od elementa pomocnog
niza"<<endl;
a3.Set(6,niz1);
cout<<"Niz a3:"<<a3;

IntArray a4(6, niz4);

a1=a4;
cout<<endl<<"Provera dodele za nizove razlicitih
velicina:"<<endl<<endl
<<"Inicijalizovan je niz a4 dimenzije 6:"<<a4<<endl
<<"Nizu a1 dodelimo vrednost niza a4 :"<<a1;

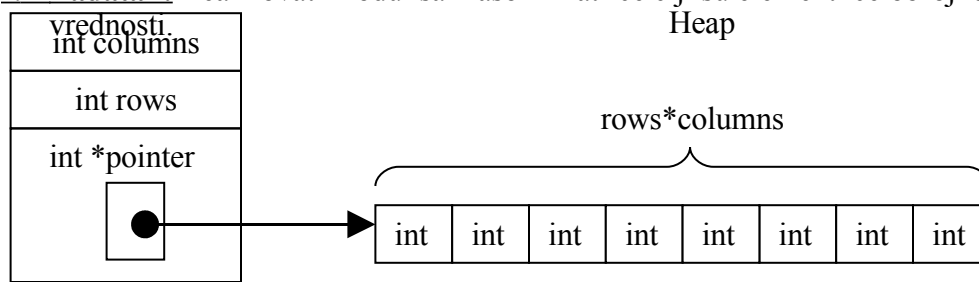
getch();

clrscr();
cout<<"Provera destruktora za niz a1:"<<endl;
a1.~IntArray();
cout<<a1;

getch();
}

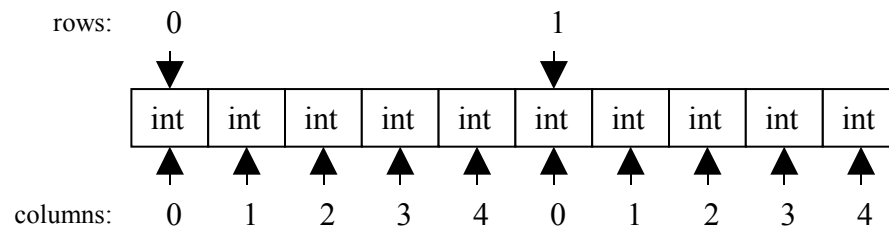
```

2. Zadatak Realizovati modul sa klasom matrice ciji su elementi celobrojne



Indeksiranje: (npr. za rows=2 i columns=5)

Heap



```
/*
*****
ZAGLAVLJE MODULA SA KLASOM MATRICE CIJI SU ELEMENTI CELI BROJEVI

Autor: Danilo Babin, 9715
Naziv datoteke: "INTMTRX2.H"
*****
*/

#include <stdlib.h>
#include <iostream.h>

#ifndef IntMatrix_def
#define IntMatrix_def

class IntMatrix{

private:
    int columns, rows;
    int *pointer;

public:
    int Empty();
    int NumberOfColumns() {return columns;}
    int NumberOfRows() {return rows;}
    IntMatrix(int r, int c);
    IntMatrix();
    IntMatrix(IntMatrix &m);
    ~IntMatrix();
    int& Get(int r); //za sekvencijalno citanje (kao niz)
```

```

        int* operator [] (int r);
        IntMatrix& operator =(IntMatrix &m);
        friend IntMatrix operator +(IntMatrix &m1, IntMatrix &m2);
        friend ostream& operator <<(ostream &o, IntMatrix &m);
};

#endif

/*****
        MODUL SA KLASOM MATRICE CIJI SU ELEMENTI CELI BROJEVI

        Autor: Danilo Babin, 9715
        Naziv datoteke: "INTMTRX2.CPP"
*****/

#include "intmtrx2.h"

int IntMatrix::Empty()
{
    if(columns==0 || rows==0) return 1;
    else return 0;
}

int& IntMatrix::Get(int r)
{
    if(r<0 || r>=rows*columns || Empty())
    {
        cout<<endl<<"( Get("<r<<" ) Pogresno indeksiranje
        elemenata!"<<endl;
        exit(EXIT_FAILURE);
    }
    return pointer[r];
}

int* IntMatrix::operator [] (int r)
{
    if(r<0 || r>=rows || Empty())
    {
        cout<<endl<<"(operator ["<r<<" ) Pogresno indeksiranje
        elemenata!"<<endl;
        return NULL;
    }
    return &pointer[r*columns];
}

```

Ova metoda je uvedena da bi se elementima matrice moglo pristupati kao da je niz u pitanju, tj. ova klasa se može koristiti i kao matrica i kao niz

operator[] je operator indeksiranja. Povratna vrednost je POKAZIVAC (adresa) na pocetak trazenog reda. Ovo omogućava da se napise m[2][4], naredba se izvršava s leva na desno, što znači da se prvo izvrši deo m[2] čiji je rezultat pokazivac (adresa) na pocetak trazenog reda matrice m. Zatim se izvršava: adresa[4], a ovo je ugrađeni operator indeksiranja u C++.

Treba primetiti da ovaj način indeksiranja ima manu jer je zasticeno od nepravilnog indeksiranja redova matrice, dok indeksiranje kolona nije zasticeno! Npr. za: IntMatrix m(2,5); ne bismo mogli napisati: m[3][2]; (pogledaj operator[]), ali bismo mogli napisati m[1][56] jer ugrađeni operator indeksiranja nema u C++ ugrađenu zastitu, pa bi se ovim dobila pogresna (nepostojeca) vrednost.

```

IntMatrix::IntMatrix()
{
    columns=rows=0;
    pointer=NULL;
}

```

```

IntMatrix::IntMatrix(int r, int c)
{
    int j;

    columns=c;
    rows=r;

    pointer = new int [rows*columns];
}

```

```

IntMatrix::IntMatrix(IntMatrix &m)
{
    int i;

    rows=m.rows;
    columns=m.columns;

    pointer = new int [rows*columns];

    for(i=0; i<rows*columns; i++) pointer[i] = m.pointer[i];
}

```

```

IntMatrix::~IntMatrix()
{
    delete [] pointer;
    rows = columns = 0;
    pointer=NULL;
}

```

Na ovaj nacin se moze osloboditi zauzeta memorija za niz, bez navodjenja dimenzija niza. Ovo se NE SME ovako raditi ako su elementi niza objekti koji imaju dinamicke elemente, jer tada memorija koju su oni rezervisali nece biti oslobodjena!

```

IntMatrix& IntMatrix::operator =(IntMatrix &m)
{
    int i;

    if (&m==this) return *this;

```

Posto je "this" pokazivac na tekuci objekat, moglo se napisati i:
this->~IntMatrix();

```

        (*this).~IntMatrix();

        rows=m.rows;
        columns=m.columns;

        pointer = new int [rows*columns];

        for(i=0; i<rows*columns; i++) pointer[i]=m.pointer[i];
        return *this;
    }

IntMatrix operator +(IntMatrix &m1, IntMatrix &m2)
{
    int i;
    IntMatrix w(m1.rows, m1.columns);

    if( (m1.rows!=m2.rows) || (m1.columns!=m2.columns) )
    {
        cout<<endl<<"(operator +) Greska! Matrice nisu istih
        dimenzija";
        return w;
    }

    for(i=0; i<w.rows*w.columns; i++)
        w.pointer[i] = m1.pointer[i] + m2.pointer[i];

    return w;
}

ostream& operator <<(ostream &o, IntMatrix &m)
{
    int i,j;

    o<<endl<<"Dimenzije matrice su: ["<<m.rows<<","<<m.columns<<"]"
    <<endl<<"Elementi matrice su:";

    i=0;
    while(i<m.rows)
    {
        o<<endl;
        j=0;
        while(j<m.columns)
        {
            o<<"["<<i<<"] ["<<j<<"] "<<"="<<m[i][j]<<" ";
            j++;
        }
        i++;
    }
    o<<endl;
    return o;
}

```

```

/*****
TEST PROGRAM MODULA SA KLASOM MATRICE CIJI SU ELEMENTI CELI BROJEVI

Autor: Danilo Babin, 9715
Naziv datoteke: "TSTIMTRX.CPP"
*****/

#include "intmtx2.h"
#include <conio.h>

//Da elementi matrice budu brojevi ozmedju [0..255]
void Initialization(IntMatrix &m)
{
    int i;

    for(i=0; i<m.NumberOfRows()*m.NumberOfColumns(); i++)
        m.Get(i) = m.Get(i) & 255;
}

void main(void)
{
    int i;
    clrscr();

    cout<<"Inicijalizacija matrice m1:"<<endl;

    IntMatrix m1(3,5);
    Initialization(m1);
    i=m1[1][4];
    cout<<m1<<endl<<"Element na poziciji [1][4] je "<<i<<endl;

    IntMatrix m2(3,5);
    Initialization(m2);
    cout<<endl<<"Inicijalizacija matrice m2:"<<endl<<m2;

    cout<<endl<<"Zbir elemenata na pozicijama [0][3] =
    "<<m1[0][3]+m2[0][3];
    m2=m1+m2;
    cout<<endl<<endl<<"Zbir matrica m2 = m1 + m2 je:"<<m2;
    getch();

    clrscr();
    cout<<"Provera konstruktora kopije:"<<endl;

    IntMatrix m3(m1);
    cout<<endl<<"Inicijalizovana je matrica m3 koja je ista kao m1:"
    <<endl<<"Matrica m1:"<<m1<<endl<<"Matrica m3"<<m3;

    IntMatrix m4(4,2);
    Initialization(m4);
    m1=m4;
    cout<<endl<<"Provera dodele za matrice razlicitih
    velicina:"<<endl<<endl
    <<"Inicijalizovana je matrica m4 dimenzija [4,2]:"<<m4<<endl
    <<"Matrici m1 dodelimo vrednost matrice m4 :"<<m1;

```

```

    getch();

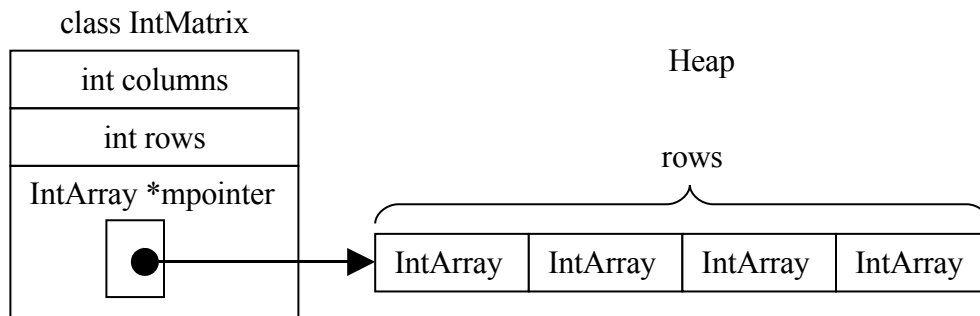
    clrscr();
    cout<<"Provera destruktora za matricu m1:"<<endl;
    m1.~IntMatrix();
    cout<<m1;

    getch();
}

```

3. Zadatak: Realizovati modul sa klasom matrice ciji su elementi celobrojne vrednosti. Zastiti od nepravilnog indeksiranja.

Da bismo ovo realizovali koristicemo prethodno napisani modul sa klasom niza:



Podatak clan “columns” je mozda nepotreban jer je broj kolona isti sa brojem elemenata svakog pojedinacnog niza IntArray, ali je uveden da bi svi znacajni podaci bili podaci clanovi.

```

/*****
    ZAGLAVLJE MODULA SA KLASOM MATRICE CIJI SU ELEMENTI CELI BROJEVI

    Autor: Danilo Babin, 9715
    Naziv datoteke: "INTMTRX4.H"
    *****/

#include <iostream.h>
#include "intarray.h"

#ifndef IntMatrix_def
    #define IntMatrix_def

```

```

class IntMatrix{

    private:
        int columns, rows;
        IntArray *mpointer;

    public:
        int Empty();
        int NumberOfColumns() {return columns;}
        int NumberOfRows() {return rows;}
        IntMatrix();
        IntMatrix(int r, int c);
        IntMatrix(IntMatrix &m);
        ~IntMatrix();
        IntArray& operator [] (int r);
        IntMatrix& operator =(IntMatrix &m);
        friend IntMatrix operator +(IntMatrix &m1, IntMatrix &m2);
        friend ostream& operator <<(ostream &o, IntMatrix &m);
};

#endif

/*****
        MODUL SA KLASOM MATRICE CIJI SU ELEMENTI CELI BROJEVI

    Autor: Danilo Babin, 9715
    Naziv datoteke: "INTMTRX4.CPP"
*****/

#include "intmtrx4.h"

int IntMatrix::Empty()
{
    if(columns==0 || rows==0) return 1;
    else return 0;
}

IntArray& IntMatrix::operator [] (int r)
{
    if(r<0 || r>=rows || Empty())
    {
        cout<<endl<<"(operator ["<<r<<"] Pogresno indeksiranje reda
        matrice!"<<endl;
        exit(EXIT_FAILURE);
    }
    return mpointer[r];
}

```

operator[] je operator indeksiranja, samo je ovde povratna vrednost referenca na objekat klase IntArray. Kada se napise: m[3][4]; prvo se izvršava m[3] koji daje objekat klase IntArray i zatim se izvršava IntArray[4], cime se kao povratna vrednost dobija referenca na int. Na ovaj nacin je izvršena zastita od pogresnog indeksiranja redova u klasi IntMatrix, dok je zastita od pogresnog indeksiranja kolona izvršena u klasi IntArray


```

IntMatrix::IntMatrix()
{
    columns=rows=0;
    mpointer=NULL;
}

```

```

IntMatrix::IntMatrix(int r, int c)
{
    int i;

    columns=c;
    rows=r;

    mpointer = new IntArray[rows];
    for(i=0; i<rows; i++) (mpointer[i]).Set(columns);

}

```

Prvo smo zauzeli memoriju za prazne objekte IntArray, a zatim smo metodom Set(int r) klase IntArray svakom objektu klase IntArray dodelili dinamicke elemente.



```

IntMatrix::IntMatrix(IntMatrix &m)
{
    int i;

    rows=m.rows;
    columns=m.columns;

    mpointer = new IntArray [rows];

    for(i=0; i<rows; i++) mpointer[i] = m.mpointer[i];
}

```

```

IntMatrix::~~IntMatrix()
{
    int i;

    for(i=0; i<rows; i++) (mpointer[i]).~IntArray();
    delete [] mpointer;
    mpointer = NULL;
    rows = columns = 0;
}

```

Da bi se izvršila destrukcija objekta, potrebno je prvo izvršiti destrukciju objektata klase IntArray sto se radi sa:

```
for(i=0; i<rows; i++) (mpointer[i]).~IntArray();
```

Ovim se uklonjaju njihovi dinamicki elementi i tek zatim se moze osloboditi memorija niza sa:

```
delete [] mpointer;
```

Na kraju se podaci clanovi stavljaju na inicijalne vrednosti.

```

IntMatrix& IntMatrix::operator =(IntMatrix &m)
{
    int i;

    if(&m==this) return *this;

    this->~IntMatrix();

    rows=m.rows;
    columns=m.columns;

    mpointer = new IntArray [rows];

    for(i=0; i<rows; i++) mpointer[i] = m.mpointer[i];

    return *this;
}

IntMatrix operator +(IntMatrix &m1, IntMatrix &m2)
{
    int i=0,j=0;

    if( (m1.rows!=m2.rows) || (m1.columns!=m2.columns) )
    {
        cout<<endl<<"(operator +) Greska! Matrice nisu istih
        dimenzija";
        exit;
    }

    IntMatrix w(m1.rows, m1.columns);

    i=0;
    while(i<w.rows)
    {
        j=0;
        while(j<w.columns)
        {
            w[i][j]=m1[i][j]+m2[i][j];
            j++;
        }
        i++;
    }

    return w;
}

ostream& operator <<(ostream &o, IntMatrix &m)
{
    int i,j;

    o<<endl<<"Dimenzije matrice su: ["<<m.rows<<","<<m.columns<<"]"
    <<endl<<"Elementi matrice su:";

    i=0;
    while(i<m.rows)

```

```

    {
        o<<endl;
        j=0;
        while(j<m.columns)
        {
            o<<"["<<i<<"["<<j<<" "<<"="<<m[i][j]<<" ";
            j++;
        }
        i++;
    }
    o<<endl;
    return o;
}

```

```

/*****

```

```

    TEST PROGRAM MODULA SA KLASOM MATRICE CIJI SU ELEMENTI CELI BROJEVI

```

```

    Autor: Danilo Babin, 9715

```

```

    Naziv datoteke: "T_INTMT4.CPP"

```

```

*****/

```

```

#include "intmtx4.h"

```

```

#include <conio.h>

```

```

void main(void)

```

```

{

```

```

    clrscr();

```

```

    IntMatrix m1(3,5);

```

```

    cout<<"Inicijalizacija matrice m1:"<<m1<<endl;

```

```

    cout<<"Element na poziciji [1][4] je "<<m1[1][4]<<endl;

```

```

    IntMatrix m2(3,5);

```

```

    cout<<endl<<"Inicijalizacija matrice m2:"<<endl<<m2;

```

```

    cout<<endl<<"Zbir elemenata na pozicijama [0][3] =

```

```

    "<<m1[0][3]+m2[0][3];

```

```

    m2=m1+m2;

```

```

    cout<<endl<<endl<<"Zbir matrica m2 = m1 + m2 je:"<<m2;

```

```

    getch();

```

```

    clrscr();

```

```

    cout<<"Provera konstruktora kopije:"<<endl;

```

```

    IntMatrix m3(m1);

```

```

    cout<<endl<<"Inicijalizovana je matrica m3 koja je ista kao m1:"

```

```

        <<endl<<"Matrica m1:"<<m1<<endl<<"Matrica m3"<<m3;

    IntMatrix m4(4,2);
    m1=m4;
    cout<<endl<<"Provera dodele za matrice razlicitih
    velicina:"<<endl<<endl
        <<"Inicijalizovana je matrica m4 dimenzija [4,2]:"<<m4<<endl
        <<"Matrici m1 dodelimo vrednost matrice m4 :"<<m1;

    getch();

    clrscr();
    cout<<"Provera destruktora za matricu m1:"<<endl;
    m1.~IntMatrix();
    cout<<m1;

    getch();
}

```

4. Zadatak: Realizovati modul sa klasom Boolean (logicki tip). Realizovati operatorski i minimalno uraditi operacije AND, OR, EXOR.

Ovu klasu bi tako trebalo realizovati da moze da se “slaze” sa razlicitim logickim izrazima i funkcijama koje kao povratnu vrednost vracaju neki indikator. Primetiti da u C-u ne postoji logicki tip, vec se svaka celobrojna vrednost razlicita od nule smatra logickom istinom, a svaka celobrojna vrednost jednaka nuli smatra logickom neistinom.

Boolean

int value;

```

/*****
    ZAGLAVLJE MODULA SA KLASOM BOOLEAN (LOGICKI TIP)

    Autor: Danilo Babin, 9715
    Naziv datoteke: "BOOLEAN.H"
    *****/

#include <iostream.h>

#ifndef Boolean_def
    #define Boolean_def

    #define TRUE 1
    #define FALSE 0

    class Boolean{

    private:
        int value;

```

```

public:

    Boolean(int r=TRUE);

    Boolean& operator =(const Boolean &b);
    friend Boolean operator ||(const Boolean &b1, const Boolean &b2);
    friend Boolean operator &&(const Boolean &b1, const Boolean &b2);
    friend Boolean operator ^(const Boolean &b1, const Boolean &b2);
    friend Boolean operator ==(const Boolean &b1, const Boolean &b2);
    friend ostream& operator <<(ostream &o, Boolean &b);
};

#endif

/*****
MODUL SA KLASOM BOOLEAN (LOGICKI TIP)

Autor: Danilo Babin, 9715
Naziv datoteke: "BOOLEAN.CPP"
*****/

#include "boolean.h"

Boolean::Boolean(int r)
{
    if(r!=0) value = TRUE;
    else value = FALSE;
}

```

Ovo je konstruktor koji ima podrazumevanu vrednost za parametar tipa int (pogledaj .H fajl). Zato se ovaj konstruktor moze pozvati bez ulazne vrednosti, pa je ujedno i podrazumevani konstruktor. Posto se konstruktor moze pozavati sa jednim parametrom tipa int, on vrši automatsku konverziju iz tipa int u objekat klase Boolean. Ovo nam je znacajno jer necemo morati pisati sve operatore sa svim mogucim varijacijama ulaznih parametara (Boolean &b1, Boolean &b2) , (Boolean &b, int i), (int i, Boolean &b), vec je dovoljno napisati samo sa ulaznim vrednostima (Boolean &b1, Boolean &b2), na ako se umesto objekta klase Boolean prosledi parametar tipa int, situacija se resava automatskom konverzijom iz tipa int u objekat Boolean, koji vrši konstruktor sa jednim parametrom datog tipa.

Napomena: U svakom operatoru u kojem ocekujemo da dodje do automatske konverzije tipa u objekat, npr. Za operator &&, potrebno je ispred parametara napisati: **const** !

Boolean operator &&(const Boolean &b1, const Boolean &b2);

Ovo je obavezno uraditi (bez toga program nece raditi) iz sledeceg razloga. Objekti se ovde prosledjuju po referenci, sto omogucava njihovo menjanje unutar funkcije. Automatska konverzija tako funkcioniše da se kreira privremeni objekat i da se prosledjuje referenca na privremeni objekat. Ako mi unutar funkcije zelimo menjati prosledjeni parametar, posto je prosledjena referenca na privremeni objekat menjacemo privremeni objekat, dok ce prosledjeni parametar ostati nepromenjen. U slucaju da smo zeleli menjati prosledjeni parametar, ovo je greska i nju ce prevodioc prijaviti. Zato u slucaju kada zelimo da se vrši automatska konverzija tipa, mi ne smemo menjati prosledjenu vrednost i to osiguravamo i prevodiocu dajemo do znanja dodavanjem modifikatora **const**.

```

Boolean& Boolean::operator =(const Boolean &b)
{
    value = b.value;
    return *this;
}

Boolean operator ||(const Boolean &b1, const Boolean &b2)
{
    if( (b1.value==FALSE) && (b2.value==FALSE) )
        return Boolean(FALSE);
    else return Boolean(TRUE);
}

Boolean operator &&(const Boolean &b1, const Boolean &b2)
{
    if( (b1.value==TRUE) && (b2.value==TRUE) ) return Boolean(TRUE);
    else return Boolean(FALSE);
}

Boolean operator ^(const Boolean &b1, const Boolean &b2)
{
    if( ( (b1.value==TRUE) && (b2.value==TRUE) ) ||
        ( (b1.value==FALSE) && (b2.value==FALSE) ) )
        return (Boolean(FALSE));
    else return (Boolean(TRUE));
}

Boolean operator ==(const Boolean &b1, const Boolean &b2)
{
    if(b1.value==b2.value) return Boolean(TRUE);
    else return Boolean(FALSE);
}

ostream& operator <<(ostream &o, Boolean &b)
{
    if(b.value==TRUE) o<<endl<<"Logicka vrenost: TRUE"<<endl;
    else o<<endl<<"Logicka vrednost: FALSE"<<endl;
    return o;
}

```

```
}
```

```
/******
```

```
TEST PROGRAM MODULA SA KLASOM BOOLEAN (LOGICKI TIP)
```

```
Autor: Danilo Babin, 9715
```

```
Naziv datoteke: "T_BOOLEAN.CPP"
```

```
*****/
```

```
#include "boolean.h"
```

```
#include <conio.h>
```

```
void main(void)
```

```
{
```

```
    Boolean a, b(TRUE), c(FALSE), d(TRUE), e(FALSE);
```

```
    clrscr();
```

```
    cout<<"b:"<<b<<"c:"<<c<<"d:"<<d<<"e:"<<e<<endl;
```

```
    a=b&&c;
```

```
    cout<<"a=b&&c"<<a<<endl;
```

```
    a=b&&d;
```

```
    cout<<"a=b&&d"<<a<<endl;
```

```
    a=b&&5;
```

```
    cout<<"a=b&&5"<<a<<endl;
```

```
    a=5&&b&&c;
```

```
    cout<<"a=5&&b&&c"<<a<<endl;
```

```
    a=b&&0;
```

```
    cout<<"a=b&&0"<<a<<endl;
```

```
    a=0&&b;
```

```
    cout<<"a=0&&b"<<a<<endl;
```

```
    a=TRUE;
```

```
    cout<<"a=TRUE"<<a<<endl;
```

```
    a=0;
```

```
    cout<<"a=0"<<a<<endl;
```

```
    getch();
```

```
    clrscr();
```

```
    cout<<"b:"<<b<<"c:"<<c<<"d:"<<d<<"e:"<<e<<endl;
```

```
    a=b||c;
```

```
    cout<<"a=b||c"<<a<<endl;
```

```
    a=c||e;
```

```
    cout<<"a=c||e"<<a<<endl;
```

```
    a=c||5;
```

```
    cout<<"a=c||5"<<a<<endl;
```

```
    a=5||b||c;
```

```
    cout<<"a=5||b||c"<<a<<endl;
```

```
    a=c||0;
```

```
    cout<<"a=c||0"<<a<<endl;
```

```
    a=0||b;
```

```
    cout<<"a=0||b"<<a<<endl;
```

```
    getch();
```

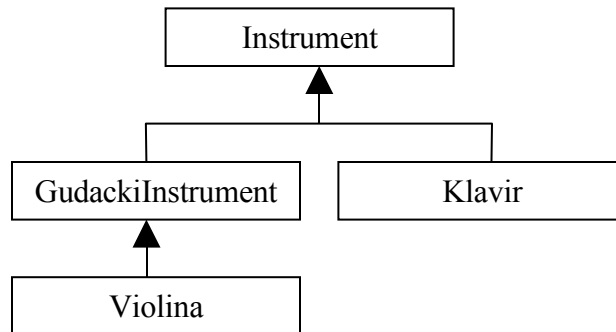
```
    clrscr();
```

```

cout<<"b:"<<b<<"c:"<<c<<"d:"<<d<<"e:"<<e<<endl;
a=b^c;
cout<<"a=b^c"<<a<<endl;
a=b^d;
cout<<"a=b^d"<<a<<endl;
a=b^5;
cout<<"a=b^5"<<a<<endl;
a=5^c;
cout<<"a=5^c"<<a<<endl;
a=b^0;
cout<<"a=b^0"<<a<<endl;
a=0^c;
cout<<"a=0^c"<<a<<endl;
getch();
}

```

5. Zadatak: Realizovati objektnu hijerarhiju klasa koja sadrži klase: instrument, gudacki instrument, violina, klavir. Svaka klasa sadrži metodu za sviranje odgovarajućeg tona i prikaz jedinstvenog naziva klase instrumenta (promenljiva nivoa klase). Za pojedine instrumente znaju se i godina i zemlja proizvodnje. Metoda za sviranje treba da ispise samo naziv konkretnog instrumenta koji svira i ton koji se svira, pri čemu se tonovi zadaju svojim nazivom na tonskoj skali (c d e f g a h). Obezbediti preklapanje operatora za umetanje na izlazni tok koji treba da ispise sve podatke o datom instrumentu. Napisati slobodnu polimorfnu funkciju koja omogućava da se bilo koji instrument pobudi da odsvira zadati ton.



```

/*****
ZAGLAVLJE MODULA APSTRAKTNE KLASKE "INSTUMENT"

Autor: Danilo Babin, 9715
Naziv datoteke: "INSTRUME.H"
*****/

#ifndef Instrument_def
#define Instrument_def

```



```

class Instrument{
    public:
        virtual void Sviraj(char ton) = 0;
        virtual void Naziv() = 0;
};

#endif

```

```

/*****
        ZAGLAVLJE MODULA APSTRAKTNE KLASSE GudackiInstrument

    Autor: Danilo Babin, 9715
    Naziv datoteke: "GUDACKII.H"
*****/

#include "instrume.h"

#ifndef GudackiInstrument_def
#define GudackiInstrument_def

class GudackiInstrument : public Instrument{
};

#endif

```

```

/*****
        ZAGLAVLJE MODULA KLASSE Violina

    Autor: Danilo Babin, 9715
    Naziv datoteke: "VIOLINA.H"
*****/

#include <iostream.h>
#include "gudackii.h"

#ifndef Violina_def
#define Violina_def

class Violina : public GudackiInstrument{
    private:
        static char* naziv;
};

```

```

        int godina;
        char *zemlja;
    public:
        Violina(int g=2005, char *z=0);
        void Naziv();
        void Sviraj(char ton);
        friend ostream& operator <<(ostream &o, const Violina &v);
};

#endif

/*****
MODUL KLASA Violina

Autor: Danilo Babin, 9715
Naziv datoteke: "VIOLINA.CPP"
*****/

#include "violina.h"

void Violina::Sviraj(char t)
{
    if (t=='c' || t=='d' || t=='e' || t=='f' || t=='g' || t=='a' ||
t=='h')
        cout<<naziv<<endl<<"Ton koji svira je: "<<t<<endl;
    else cout<<"Prosledjeni ton ne postoji!"<<endl;
};

Violina::Violina(int g, char *z)
{
    godina = g;
    if (z==0) zemlja="nepoznata";
    else zemlja=z;
};

ostream& operator <<(ostream &o, const Violina &v)
{
    o<<"Naziv klase: "<<(v.naziv)<<endl<<"Godina proizvodnje:
"<<v.godina
    <<endl<<"Zemlja porekla: "<<v.zemlja<<endl;

    return o;
};

void Violina::Naziv()
{
    cout<<endl<<"Naziv klase: "<<naziv<<endl;
};

```

```

/*****
                        ZAGLAVLJE MODULA ZA KLASU KLAVIR

Autor: Danilo Babin, 9715
Naziv datoteke: "KLAVIR.H"
*****/

#include "instrume.h"
#include <iostream.h>

#ifndef Klavir_def
#define Klavir_def

class Klavir : public Instrument{
private:
    static char* naziv;
    int godina;
    char *zemlja;

public:
    Klavir(int g=2005, char *z=0);
    void Naziv();
    void PromeniZemlju(char *z);
    void Sviraj(char ton);
    friend ostream& operator <<(ostream &o, Klavir &k);
};

#endif

/*****
                        MODUL KLASE KLAVIR

Autor: Danilo Babin, 9715
Naziv datoteke: "KLAVIR.CPP"
*****/

#include "klavir.h"

void Klavir::Sviraj(char t)
{
    if (t=='c' || t=='d' || t=='e' || t=='f' || t=='g' || t=='a' ||
t=='h')
        cout<<naziv<<endl<<"Ton koji svira je: "<<t<<endl;
    else cout<<"Prosledjeni ton ne postoji!"<<endl;
};

Klavir::Klavir(int g, char *z)
{
    godina = g;
    if (z==0) zemlja="nepoznata";
    else zemlja=z;
};

```

```

ostream& operator <<(ostream &o, Klavir &k)
{
    o<<"Naziv klase : "<<(k.naziv)<<endl<<"Godina proizvodnje:
"<<k.godina
    <<endl<<"Zemlja porekla: "<<k.zemlja<<endl;

    return o;
};

void Klavir::Naziv()
{
    cout<<endl<<"Naziv klase: "<<naziv<<endl;
};

/*****
    TEST PROGRAM ZA MODULE KLASA Instrument, GudackiInstrument,
    Klavir, Violina

    Autor: Danilo Babin, 9715
    Naziv datoteke: "T_INSTRU.CPP"
*****/

#include <conio.h>
#include "klavir.h"
#include "violina.h"

void Odsviraj(Instrument &i, char ton)
{
    i.Sviraj(ton);
    cout<<endl;
}

void func(Klavir k)
{
    cout<<"func: "<<endl<<k<<endl;
    k.PromeniZemlju("Francuska");
    cout<<endl<<k<<"kraj func"<<endl;
}

char* Violina::naziv="violinica";
char* Klavir::naziv="klavirche";

void main(void)
{
    Klavir ks(2001,"Srbija");
    Klavir k(1995);
    Violina vi(1778, "Italija");

    clrscr();
    cout<<ks<<endl<<k<<endl<<vi<<endl;
    Odsviraj(ks, 'f');
    Odsviraj(k, 'c');
    Odsviraj(vi, 'd');
}

```

```
    getch();
    clrscr();
    cout<<ks<<endl;
    ks.PromeniZemlju("Austria");
    cout<<"PromeniZemlju: "<<endl<<ks<<endl;

    func(ks);
    cout<<endl<<ks;

    getch();
}
```