Programabilni uređaji i objektno orjentisano programiranje

Prijateljske funkcije i klase.

Prijatelji i potreba za njima

- Još jedan, moramo priznati, čudan termin u programiranju – **prijatelj**. Obećavam, biće još čudnijih...
- Za rješavanje jednog problema u programiranju često je neophodno više klasa.
- Pretpostavimo da rješavamo softverski problem koji se tiče ATM automata za prenos novca: imamo klase **Mušterija**, **Automat**, **Održavalac, Račun** itd.
- Održavalac mora da zna podatke o Automatu (npr. kada Automat ostane bez novca ili papira za štampanje), ali ne smije da ima podatke o Mušterijama i Računima, niti Mušterije smiju da imaju podatke o Automatu.

Kako prevazići problem javnih članova?

- Ako bi Automat učinio neke od svojih podataka članova javnim, on bi to uradio kako za klasu Održavalac, tako i za sve ostale klase u sistemu, čime bi bio ugrožen fundamentalni koncept OOP – enkapsulacija.
- Postoji način da se ovo prevaziđe: klasa može da učini **drugu klasu ili funkciju nečlanicu ili funkciju članicu druge klase** svojim **prijateljem**.
- Prijatelj može da pristupa privatnim podacima članovima klase, ali to ne mogu da rade druge funkcije i klase.

Deklaracija prijatelja

- Prijateljstvo se deklariše unutar klase koja želi da od druge klase ili funkcije napravi prijatelja, a za to se koristi ključna riječ - **friend**:

```
class Automat {/* sve što čini klasu*/
      friend Odrzavanje; //klasa Odrzavanje je
      //prijateljska klasa, podrazumijeva se da je već
      //deklarisana. U suprotnom bi se moralo pisati
      //friend class Odrzavanje
      friend void f1(); //funkcija nečlanica koja je
      //prijatelj
      friend int Lopov::UzmiKopiju(Automat); //funkcija
      //članica druge klase koja je prijatelj, podrazumijeva se
      //da prije ovog koda imamo deklaraciju class Lopov;
      friend void Lopov::UzmiOriginal(Automat &);};
```

- Dakle, funkcija nečlanica `f1()` i navedene funkcije iz klase Lopov **(ako funkcije imaju argumente, ovdje im se moraju navesti argumenti!!!)** imaju pravo da pristupaju podacima članovima klase Automat, kao i sve funkcije članice klase Odrzavanje.

Deklaracija prijatelja

- Sada bismo u glavnom programu mogli imati naredbe

```
Lopov l1;

Automat a1;

l1.UzmiOriginal(a1);

l1.UzmiKopiju(a1);
```

- Prijateljska funkcija članica druge klase se poziva preko objekta klase čija je ta funkcija članica, dok joj se objekat klase čiji je prijatelj mora proslijediti;
- U finkciji `UzmiOriginal()` sve izmjene izvršene nad podacima objekta a1, važe i nakon njenog izvršavanja, dok to nije slučaj sa funkcijom `UzmiKopiju()` – poslata joj je samo vrijednost.
- Ne bismo smjeli imati funkciju npr. `Uzmi()`, koja bi se razlikovala samo po tome da li joj se objekat klase `Automat` šalje po vrijednosti, ili po referenci, bez obzira na njen rezultat!!!

```cpp
#include <iostream>

using namespace std;
class Automat; //mora se naglasiti da je Lopov klasa
//kompletna deklaracija se moze dati kasnije

class Lopov{
    private:
        int KolikoUzimam;
        //sve sto cini klasu
    public:
        Lopov(int b=0):KolikoUzimam(b){}
        ~Lopov(){}
        int UzmiKopiju(Automat);
        void UzmiOriginal(Automat &);
        int KolikoImam(){return KolikoUzimam;}
};

//Ovdje jos ne mozemo dati definiciju funkcija clanica klase Lopov
//koji koriste podatke clanove klase Automat, jer samo znamo da
//postoji klassa Automat, ali ne i koje podatke ima

class Automat {/* sve sto cini klasu*/
    private:
        int Gotovina;
        friend class Odrzavanje; //Jos se ne zna da je Odrzavanje klasa
        friend int Lopov::UzmiKopiju(Automat);
        friend void Lopov::UzmiOriginal(Automat &);
    public:
        Automat(int a=0):Gotovina(a){}
        ~Automat(){}
        int KolikoIma(){return Gotovina;}
         };
```

```
//Tek se nakon pune deklaracije klase Automat mogu dati definicije
//prijateljskih funkcija koje koriste njene podatke clanove
int Lopov::UzmiKopiju(Automat a){
    a.Gotovina-=KolikoUzimam;
    return a.Gotovina;}

void Lopov::UzmiOriginal(Automat & a){
    a.Gotovina-=KolikoUzimam;}


int main()
{
    Automat a1(5000);
    Lopov l1(3000);
    int b;
    cout<<"Na automatu ima gotovine: "<<a1.KolikoIma()<<endl;
    l1.UzmiOriginal(a1);
    cout<<"Nakon poziva funkcije UzmiOriginal ";
    cout<<"na automatu ima gotovine: "<<a1.KolikoIma()<<endl;
    l1.UzmiKopiju(a1);
    b = l1.UzmiKopiju(a1);
    cout<<"Nakon poziva funkcije UzmiKopiju ";
    cout<<"na automatu i dalje ima gotovine: "<<a1.KolikoIma()<<endl;
    cout<<"U funkciji je bilo"<<b<<", ali to ne vidimo nakon njenog izvrsavanja"<<endl;
    cout<<"Radili smo sa kopijom objekta, ne mijenja se njegovo stanje"<<endl;
    return 0;
}
```

```
Na automatu ima gotovine: 5000
Nakon poziva funkcije UzmiOriginal na automatu ima gotovine: 2000
Nakon poziva funkcije UzmiKopiju na automatu i dalje ima gotovine: 2000
U funkciji je bilo-1000, ali to ne vidimo nakon njenog izvrsavanja
Radili smo sa kopijom objekta, ne mijenja se njegovo stanje
```

Prijatelji - Komentari

- Deklarisati prijateljsku klasu je prečica koja čini da klasa proglasi sve funkcije druge klase prijateljima.
- U prijateljskoj klasi ne treba vršiti naglašavanje da je ta klasa prijatelj nekoj drugoj klasi (a to se i ne može uraditi).
- Postavlja se pitanje da li prijateljstvo ugrožava najvažniji koncept OOP – enkapsulaciju?
- Ne!
- Prijateljstvo se ne može nametnuti spolja, sama ga klasa definiše i kontroliše.

Prijateljstvo - Karakteristike

- **Prijateljstvo nije komutativno.** Ako je klasa A prijatelj klase B to ne znači da je klasa B prijatelj klase A.
- **Prijateljstvo nije tranzitivno (prenosivo).** Ako je klasa A prijatelj klase B, a klasa B prijatelj klase C, to ne znači da je klasa A prijatelj klase C.
- Jednom riječju, prijateljstvo se (kod klasa) ne otima već se poklanja.
- Primjer prijatelja:

```
class X{
    friend void g(int, X&);
    int i; /*itd*/};

void g(int k,X &x){x.i=k;} //X x1; g(3,x1);
//pristup privatnom podatku članu iz prijatelja
//prijateljska f-ja nema pokazivač this!!!
```

Ostali razlozi za prijateljstvo

- Pored navedene situacije **(kada činimo javnim podatke članove pojedinim klasama da bi zajednički, kaže se, u društvu saradnika, rješavali neki problem),** postoje barem još četiri situacije kada vrijedi koristiti koncept prijatelja.
- Sve četiri situacije su suštinske slične, odnosno, gotovo iste, i često prevazilaze značaj osnovnog značenja prijateljstva:
  - Lakša konverzija podataka iz jednog u drugi tip podataka kod prijateljskih funkcija nečlanica nego kod funkcija članica;
  - Lakša realizacija funkcija koje imaju argumente koji su različitih klasnih tipova (sve klase koje se pojavljuju kao argument proglašavaju tu funkciju kao prijatelja);
  - Notaciono je jasnije f(x,y) nego x.f(y);
  - Upotreba kod preklapanja operatora.

Ilustracija drugih razloga

- Posmatrajmo "čuvenu" funkciju sabcomp koja nam je služila za sabiranje kompleksnih brojeva.
- Ako u klasi postoji konstruktor koji je u stanju da konstruiše kompleksni broj na osnovu float-a, mi smo mogli da izvršimo sljedeće pozive ove funkcije:

```
z2 = z1.sabcomp(z); //z, z1 i z2 su kompleksni brojevi
z2 = z1.sabcomp(a); //na osnovu float-a a konstruiše se
//kompleksni broj
```

- U operaciji sabiranja argumenti su ravnopravni, ali mi operaciju sabcomp ne možemo pozvati sa `a.sabcomp(z1)`, jer ova funkcija nije definisana za realne brojeve.
- Podacima objekta `a` se pristupa direktno, preko pokazivača this, nema kpiranja i pozivanja konstruktora koji vrši željenu konverziju.
- Dodatno, već smo uočili da ovakva notacija za pozivanje funkcije ne mora da bude baš najjasnija.

Kako prevazići prethodne probleme?

- Jednostavno!

```
class complex{
private:
    float re,im;
public:
    complex(); //default konstruktor
    complex(float); // i onaj koji pravi kompleksni
    //broj na osnovu float-a
    friend complex sabcomp(complex,complex);
    //deklaracija da postoji prijatelj f-ja nečlanica
//još koješta};
complex sabcomp(complex z1, complex z2){
  complex z;
  z.re=z1.re+z2.re; z.im=z1.im+z2.im;
  return z;}
```

- Funkcija sabcomp pristupa direktno podacima članovima klase complex iako nije članica već prijatelj.

Problem prevaziđen!

- Sada se funkcija može pozvati na bilo koji od načina:

```
sabcomp(z1,z2);
sabcomp(z,2.5);
sabcomp(2.5,z);
sabcomp(2.5,3.1);
```

  jer će u svim slučajevima na osnovu realnog broja biti, prilikom prenosa argumenata i poziva odgovarajućeg konstruktora, konstruisan kompleksni broj i funkcija ispravno izvršena.

- Ovim smo završili osnovnu priču o prijateljima. Ova priča se dobrim dijelom ponavlja u okviru sljedeće naše teme - **PREKLAPANJA OPERATORA**.

Preklapanje operatora - Potreba

- Funkcijom sabcomp iz prethodnog primjera riješili smo značajne nedostatke vezane za realizaciju ove funkcije kada je u pitanju ravnopravnost argumenata.
- Međutim, teško je pamtiti nazive funkcija, a za sabiranje kompleksnih brojeva najprirodnije bi bilo zadržati matematičku notaciju:
  z1+z2;
- Operatori **(kao što je operator +)** se lakše pamte nego što se pamte nazivi funkcija, te je stoga pogodno **(ako je moguće)** koristiti operatore, a ne direktne pozive funkcija.
- Programski jezik C++ dozvoljava ovu pogodnost i to se naziva preklapanjem operatora (operator overloading).

Koji se operatori ne mogu preklopiti

- Samo 5 od ukupnog spiska operatora u programskom jeziku C++ se ne može preklopiti. To su:
  - . i .* (ako bi im se promijenilo značenje ne bi se moglo pristupati podacima članovima klase);
  - :: (operator dosega iz istih razloga kao prethodna dva);
  - :? (iz nekog nepoznatog razloga nema dozvole da se ternarni operator preklopa);
  - sizeof (svaki objekat mora da na isti način vrati memoriju koju objekat zauzima u bajtovima, a to sizeof iz C++ već radi dobro).
- Svi ostali operatori se mogu preklapati, s time što za neke važe posebna pravila.

Operatori koji se preklapaju

+	-	*	/	%	^	&	|	~
!	=	<	>	+=	-=	*=	/=	%=
=	^=	&=	|=	<<	>>	<<=	>>=	==
!=	>=	<=	&&	||	++	--	,	->*
->	()	[]	new	delete				

- Neki operatori su isključivo binarni (npr. +=), neki mogu biti i binarni i unarni (npr. -), neki mogu biti isključivo unarni (npr. ++).
- Za neke od operatora (npr. [], (), new i delete) važe vrlo specifična pravila preklapanja.

Operatori koji se preklapaju

- Operatori >> i << se mogu standardno preklapati i to na specijalan način koji će biti učen kada budemo radili detaljno iostream biblioteku. (Ranije smo rekli da ćemo ih zvati komandama, a da su zapravo operatori.)
- Operatori ++ i - - u zavisnosti od toga da li su prefiksni ili sufiksni se preklapaju na različit način.
- Operator dodjele (=) se ne mora preklapati jer je to četvrta **(i posljednja)** funkcija članica koju nam obezbjeđuje kompajler (uz konstruktor, desktruktor i konstruktor kopije). Preklapa se kada imamo podatke članove koji su pokazivači, kao i ako želimo prilikom dodjele da izvršimo neki specijalan efekat.

Pravila kod preklapanja operatora

- Operatorska funkcija ima posebno ime **operator@**, gdje je **@** operator koji se preklapa. Preklapaju se binarni i unarni operatori. Ako se preklapa binarni operator onda se sa **a@b** zapravo poziva funkcija **a.operator@(b)**.
- Operatorska funkcija **mora** biti funkcija članica ili funkcija nečlanica koja ima barem jedan argument koji je datog klasnog tipa (ili referenca na klasu).
- Operatorske funkcije **=**, **[]**, **()** i **->** uvijek moraju biti članovi klase.
- Ne može se promijeniti broj operanada, način grupisanja ni prioritet operatorskih funkcija.

Pravila kod preklapanja operatora

- Posljednje navedeno pravilo znači da je **a+b** sigurno binarni operator i da se na ovo ne može uticati, da se kod operacija **(a+b)*(c*d+e)** uvijek izvršava prvo ono u zagradama **(zagrade su takođe operatori)**, pa onda * ima veći prioritet u odnosu na **+** bez obzira što to predstavlja.
- Ne mogu se predefinisati značenja operatora za ugrađene tipove **(+ uvijek ostaje dobro staro sabiranje za int-ove i float-e).**
- Ne mogu se uvoditi novi operatori! To što nam je dao C++ (a toga nije malo) je sve što možemo da preklopimo.

“Dobro” pravilo – članica ili nečlanica

- Vezano za izbor članice ili nečlanice, postoje dva relativno jednostavna pravila koja se trebaju poštovati (mada nije obavezno):
  - Ako operatorska funkcija može da promijeni lijevi operand (npr. kod operacije +=) onda treba da bude članica.
  - Ako operatorska funkcija ima operande sa jednakim pravima **(ponekad se kaže simetrični)**, koji ne mogu biti promijenjeni u funkciji i koji dozvoljavaju da dođe do konverzije operanada u klasni tip podatka treba koristiti funkciju nečlanicu.
- Ova drugo pravilo je pravilo do kojeg smo idošli prilikom “pomjeranja” funkcije za sabiranje kompleksnih brojeva izvan klase complex (da postane nečlanica).

Razlike operatorska funkcija - operator

- Sljedeći elementi se mogu razlikovati između standardnih operatora i operatorskih funkcija:
  - tip i vrijednost operanada (u operaciji + se mogu pojaviti drugačiji operandi nego kod standardnog sabiranja);
  - tip i vrijednost rezultata;
  - bočni efekat na operand (standardno `a = b + c;` ništa se ne dešava sa operandima `b` i `c`, ali kod preklapanja operatora možemo da definišemo promjenu nad operandima i tamo gdje kod ugrađenih tipova podataka ne postoji);
  - veze između operatora (objasnićemo kasnije).
- Ovo je (pre)velika sloboda, pa se stoga često uvode ograničenja u ovoj slobodi radi fleksibilnosti i lakšeg razumijevanja programa.

Ograničenja razlika

- Operatorska funkcija, radi lakšeg razumijevanja i nadogradnje programa, bi trebala da:
  - Po odnosu sa argumentima i rezultatu liči na standardni operator (`a = b + c;` trebalo bi da vrati rezultat odgovarajućeg tipa i da ne mijenja operande);
  - Operatori bi trebalo da imaju očekivana značenja i veze sa drugim operatorima (ako je preklopljen operator `+`, i preklopimo operator `+=`, onda `a += b;` bi trebalo da ima isti smisao kao `a = a + b;`);
  - Trebalo bi da budu definisani svi operatori koji se očekuju (ako ste preklopili `+`, a ima smisla preklopiti `+=`, to treba i učiniti u skladu sa prethodnom preporukom).
- Zapamtite da tip rezultata može kod preklopljenih operatora da bude proizvoljan, osim kod new i delete.

Preklapanje binarnih operatora

- Binarni operatori se mogu realizovati kao funkcije članice koje imaju jedan argument (drugi argument je objekat za koji se funkcija poziva, dostupan preko pokazivača **`this`**) ili kao funkcije nečlanice sa dva argumenta.
- Na primjer sabiranje kod klase kompleksnih brojeva može se realizovati kao:

```
complex operator+(complex);     //slučaj f-je članice
complex operator+(complex,complex); //f-ja nečlanica
 // koja je po pravilu prijatelj
```

- Pravilo koje smo uveli kad smo pričali o prijateljskim funkcijama i ovdje važi; veća se fleksibilnost u odnosu na argumente operatorske funkcije postiže kada se u ovom slučaju realizuje kao prijateljska funkcija nečlanica.

Preklapanje binarnih operatora

- Kako se očekuje da `operator +=` promijeni lijevi operand, to ga treba implementirati kao funkciju članicu sa jednim argumentom.
- Ako je operatorska funkcija (realizovana kao prijatelj) pozvana kao `a = b + c;` to je ekvivalentno pozivu: `a = operator+(b,c);` dok je za funkciju članicu poziv ekvivalentan sa `a = b.operator+(c);`.
- Što se dešava prilikom poziva `a = b + c;` gdje su `b` i `c` klasni tipovi podataka?
- Naravno, moramo se odlučiti samo za jedan od navedena dva načina preklapanja operatora, da bi funkcionisao gornji poziv `a = b + c;`

Mehanizam pozivanja operatorske funkcije

- Kako operator nije po default-u definisan za klasne tipove podataka, kompajler provjeri da li je taj operator preklopljen u okviru klase i ako jeste pozove odgovarajuću funkciju, a slično se dešava ako je u pitanju nečlanica.
- Preklopljeni operator `+` se za slučaj postojanja odgovarajućeg konstruktora koji konstruiše kompleksni broj na osnovu realnog (ili cijelog) broja kod prijateljske funkcije nečlanice može pozvati kao:

```
a = b + 3; c = 4 + a;
```

- Što se dešava prilikom poziva `a = 3.7 + 4.1;` da li dolazi do poziva preklopljenog operatora ili nečeg drugo?

Primjer preklopljenog operatora

- Iz ilustrativnih razloga navodimo moguću realizaciju preklopljenog operatora **+**:
  Unutar klase se najavljuje kao:

```
friend complex operator+(complex,complex);
```

  dok mu realizacija može biti:

```
complex operator+(complex z1,complex z2){
return complex(z1.real+z2.real,z1.imag+z2.imag);}
```

- Uočite trik da smo konstruktor koji prima dva argumenta uposlili za vraćanje rezultata funkcije.

Primjer operator += (rezultat referenca)

- Jedna moguća realizacija operatorske funkcije **+=** za klasu complex:

```
complex & operator+=(complex cr){
    real += cr.real;
    imag += cr.imag;
    return *this;}
```

- Ovdje je operatorska funkcija realizovana kao funkcija članica jer mijenja lijevi operand (za koji se poziva). Postavlja se pitanje zbog čega smo uopšte vraćali rezultat kada je jedini rezultat ove operacije, na prvi pogled, objekat koji je lijevi operand **`x+=y;`**?
- Da bismo mogli izvršili operacij **`x+=3;`** argument cr prosleđujemo po vrijednosti.
- Odgovor je radi kompletnosti. Neko može da ovo koristi kao: **`z=x+=y;`** čime bi rezultat ove operacije (vrijednost **`x`**-a) želio pridružiti **`z`**, a to može učiniti samo ako kao rezultat ima objekat.

Operator +=

- Odgovor na pitanje zbog čega je ovdje rezultat vraćen po referenci, mora li se to raditi i zašto je u datom primjeru ovo uvijek moguće i poželjno pokušajte sami da odgonetnete.
- Unarni operatori se mogu preklopiti kao prijatelji sa jednim argumentom ili kao funkcije članice bez argumenata. Npr.
-
```
complex operator!() const {
return complex(real,-imag);
}
```
- Obratiti pažnju da se `()` moraju pisati i u slučaju kada funkcija nema argumenata.

Sufiksni i prefiksi operatori

- Kod preklapanja unarnih operatora postoji specifičan problem: operatori inkrementiranja i dekrementiranja mogu se pojaviti u dva oblika:
  - `++a;` koji se naziva prefiksnim i
  - `a++;` koji se naziva sufiksnim (postfiksnim).
- Postavlja se pitanje kako ćemo naznačiti da želimo raditi sa jednim ili drugim tipom operatora?
- Prvo, kakav je standardni operator? On je prefiksni, kao što su –a, !a i ostali unarni operatori.
- Dakle, ovakva funkcija (ako je članica) bi se deklarisala kao:
  `X operator++();`
- Uvijek imajte na umu da je ideja ovog operatora nešto uvećati pa to uvećano upotrijebiti (vratiti kao rezultat).

++ prefiksni primjer

- Neka na raspolaganju imamo klasu **`Brojac`** koja ima samo jednu cjelobrojnu promjenljivu. Realizacija operatora ++ za ovu klasu bi mogla da bude:

```
Brojac Brojac::operator++(){
    ++vr;
    return *this;}
```

- Što se tiče sufiksnog operatora situacija je nešto drugačija. On se poziva kao **`x++;`** i kompajler mora da generiše unekoliko različit poziv operatorske funkcije **`operator++()`**.
- Stoga se prilikom poziva **`x++;`** zapravo poziva specijalna funkcija **`operator++(int)`**.
- Cjelobrojni argument se prilikom ralizacije ne mora ni imenovati. Služi samo kao signal (flag) da se poziva sufiksni operator **(mora se razlikovati po broju ili tipu argumenata od prefiksnog)**

Sufiksni operator

- Dakle, prilikom poziva sufiksnog operatora zapravo se poziva verzija preklopljene operatorske funkcije koja uzima int argument. Ovaj argument se obično i ne koristi, već je samo **flag** (zastavica) koja signalizira da je u pitanju sufiksni operator.
- Primjer za klasu Brojac:

```
Brojac Brojac::operator++(int)
    {Brojac br=*this;
    vr++;
    return br;}
```

- Na ovaj način smo uspjeli da upotrijebimo objekat, pa da ga inkrementiramo, što je ideja sufiksnih operatora. Ako bi prvo išao return ostatak funkcije se ne bi ni izvršio.

Ostali operatori

- Postoji niz specifičnih pravila vezanih za preklapanje pojedinih operatora. Neka od tih pravila ćemo izložiti na narednom času, dok će za dva operatora biti objašnjena relativno kasno u našem kursu.
- Važnije (odnosno jednostavnije) operatore ćemo detaljnije objasniti nego neke relativno složene zbog kratkoće našeg vremena.
- Prvi operator iz ove specijalne kategorije je operator pridruživanja.
- Ovaj se operator ne mora preklopiti!!! odnosno obično se ne preklapa osim kada imamo pokazivače kao podatke članove.

Operator pridruživanja

- Operator pridruživanja je posljednja od 4 funkcije koju kompajler sam kreira umjesto nas i koju ne predefinišemo osim kada nam treba neka specijalna funkcionalnost ili kada radimo sa klasama koje imaju pokazivače kao podatke članove. Tri do sada uvedene funkcije, koje kompajler sam kreira, su: konstruktor, destruktor i konstruktor kopije.
- Operator pridruživanja se po pravilu mora preklopiti kada imamo pokazivačku promjenljivu kao podatak član.
- Preklapanje operatora pridruživanja mora biti obavljeno u skladu sa onim operacijama koje su uvedene kod definisanja sopstvene verzije konstruktora kopije.