Programiramabilni uređaji i objektno orijentisano programiranje

Nasljeđivanje – Virtuelni metodi
Šabloni

Polimorfizam i nasljeđivanje

- Jedan od problema koji nam se javlja prilikom nasljeđivanja je situacija da u sistemu možemo da imamo veliki broj klasa izvedenih iz jedne osnovne, a da na sve objekte izvedenih klasa želimo primijeniti neku operaciju koja može biti kao kod osnovne klase, ali može da bude i verzija iz izvedene klase.
- Najelegantniji način koji nam može pasti na um je da u jednoj petlji prođemo kroz sve objekte i da odradimo operaciju nad svim objektima.
- Na prvi pogled ovo je nemoguće, jer ćemo morati da primijenimo petlju za svaki tip objekata izvedene klase ili da implementiramo program koji će imati veliku if selekciju ili eventualno switch-case koja bi se pitala kojoj od izvedenih klasa pripada koji od objekata.

Virtuelni metodi

- U OOP je dodat jednostavan koncept virtuelnih metoda koji omogućava prevazilaženje prethodno navedenog problema.
- Virtuelni metod se definiše u osnovnoj klasi navođenjem ključne riječi virtual ispred naziva metoda.
- Izvedene klase mogu, ali i ne moraju, da definišu svoju verziju virtuelnih metoda.
- Ako se virtuelni metod poziva preko pokazivača ili reference na osnovnu klasu izvršava se ona verzija koja odgovara konkretnoj izvedenoj klasi na koju pokazivač pokazuje.

Virtuelni mehanizam

- Ponovo napominjemo da virtuelni mehanizam funkcioniše samo u slučaju da se (virtuelni) metodi pozivaju preko pokazivača ili reference na osnovnu klasu (koja može da pokazuje ili referencira objekat izvedene klase), dok virtuelni mehanizam ne funkcioniše ako se metodi pozivaju direktno preko objekata.

- Posmatrajmo sljedeći slučaj:

```
class clan_biblioteke{
//…
protected: int r;
public:
virtual int plati_clan() {return r-=clanarina;}
//…
};
```

Virtuelni metod u osnovnoj klasi.

Virtuelni mehanizam - Primjer

- Posmatrajmo izvedenu klasu:

```
class pocasni_clan:public clan_biblioteke{
//…
virtual int plati_clan() //virtuelna funkcija
{return r;}
};
```

- U predmetnom primjeru biblioteka naplaćuje članarinu, a ako je u pitanju počasni član njemu se članarina ne naplaćuje.
- Sada se može preko pokazivača na član osnovne klase pozvati funkcija plati_clan(), a koja će od funkcija biti pozvana zapravo zavisi od toga na koji objekat ovaj pokazivač pokazuje (objekat osnovne klase ili objekat izvedene klase).

Virtuelni metodi - Pravila

- Ako je funkcija virtuelna u osnovnoj klasi ona je virtuelna i u izvedenoj klasi, pa ključnu riječ **virtual** nije potrebno pisati u izvedenoj klasi.
- Praksa je, ipak, radi lakšeg praćenja programa pisati ovu riječ.
- Da bi virtuelni mehanizam funkcionisao potrebno je da funkcije u svim klasama budu iste po tipu (sve mora biti isto i argumenti i rezultati).
- Ako se virtuelne funkcije razlikuju po argumentima, onda će funkcija iz izvedene klase da sakrije f-ju iz osnovne klase (ovo nije preklapanje – overloading, već zasjenjivanje, što se u engleskoj terminologiji naziva overriding).

Virtuelni metodi – Pravila i Primjer

- Naravno, nije dozvoljeno da se funkcije iz osnovne i izvedene klase razlikuju samo po tipu rezultata i to bi dovelo do greške u kompajliranju. Zbog čega?

- Posmatrajmo sljedeći generički primjer:

```
class B{public:
virtual void vf1();
virtual void vf2();
virtual void vf3();
void f();
};
```

```
class D:public B{public:
void vf1();
void vf2(int);
//sakriva - zasjenjuje B::vf2();
int vf3();
//greška, razlika samo po rezultatu
void f(); //nije virtuelna f-ja
};
```

Virtuelni metodi – Primjer - Nastavak

- Posmatrajmo sada glavni program koji barata sa klasama B i D:

```
void main(){
D d;
B *pb=&d;       //pokazivač na osnovnu klasu koji
          //pokazuje na objekat izvedene klase,
          //što je dozvoljeno jer je izvedena
          //klasa dostupna
pb->vf1();
//radi virtuelni mehanizam, f-ja iz D klase
pb->vf2();
//ne radi virtuelni mehanizam, jer je f-ja iz D
//klase različita od f-je iz B klase, pa se
//poziva f-ja iz B klase, po tipu pokazivača
pb->f(); //poziva se nevirtuelna f-ja iz B klase
}
```

Virtuelne f-je kao dogradnja

- Često se želi postići da virtuelna f-ja dopunjuje rad funkcije iz osnovne klase, pa je tada pogodno postići da virtuelna funkcija iz izvedene klase pozove svoj par iz osnovne klase. Primjer kako se to može uraditi:

```
class B{
//...
public:
//...
virtual void f();
};
```

```
class D:public B{
//...
public:
void f(){
B::f();
//poziv f-je iz osnovne klase
//...
}
};
```

v-table

- Postavlja se pitanje kako kompajler zapravo reguliše pozivanje funkcija u situaciji kada se za isti objekat, u zavisnosti da li je funkcija pozvana preko samog objekta, reference ili pokazivača na njega, poziva funkcija asocirana sa različitim klasama (osnovnim ili izvedenim).
- Realizacija ovog koncepta nije nimalo jednostavna.
- Obično se koristi tzv. virtuelna tabela ili v-table u kojoj kompajler upisuje koja će se f-ja pozvati u zavisnosti od toga kako se pristupa datom objektu.
- Sreća u nesreći je činjenica da rad sa v-table reguliše kompajler i da o tome programeri ne treba da vode mnogo računa.

v-table

- Kompajler za klasu u kojoj se nalazi makar jedna virtuelna funkcija kreira v-table.
- Kako v-table zahtjeva memorijski prostor, a još gore zahtjeva i vrijeme za njeno konsultovanje prilikom pozivanja funkcija, to njeno postojanje slabi performanse našeg programa.
- Osnovno pravilo je da veoma jednostavne klase ne treba da imaju virtuelnih funkcija, jer održavanje v-table kvari performanse ovakvih klasa u značajnoj mjeri.
- Napominjemo da nije obaveza kompajlera da kreira v-table, već može rad sa virtuelnim f-jama da realizuje i na drugi način, ali da kompajleri to po pravilu rade preko v-table.

v-table

- Koga interesuju detalji rada sa v-table može da konsultuje udžbenik od Milićeva.
- Naše je mišljenje da je to relativno nevažno za programera koji prati prethodno uvedena uputstva (bez virtuelnih funkcija kod jednostavnih klasa).
- Ostatak posla umjesto nas radi kompajler i od značaja je samo za programere koji žele da izučavaju rad kompajlera ili da ga kreiraju.

Konstruktor i destruktor kao virtuelne f-je

- Konstruktor se poziva prije nego je objekat kreiran (on potpomaže kreiranje objekta).
- Dakle, on se poziva i prije nego postoji v-table za objekat (v-table se mora kreirati sa svakim objektom neke klase).
- Stoga, **konstruktor ne može biti virtuelna funkcija** jer u trenutku njegovog pozivanja ne postoji v-table!
- Ovo važi za sve konstruktore, uključujući konstruktor kopije, ali ćemo priču o konstruktoru kopije malo kasnije detaljnije objasniti.

Destruktor kao virtuelna f-ja

- Ako smo pozvali destruktor objekta izvedene klase preko pokazivača na osnovnu klasu, i ako destruktor nije virtuelna funkcija, došlo bi do pozivanja destruktora za osnovnu klasu, a ovaj bi dealocirao samo dio objekta koji potiče od osnovne klase ostavljajući "da visi" dio objekta koji je nadodala izvedena klasa.
- Stoga, **destruktor** **(ako je bilo koja funkcija u klasi virtuelna)** **bi morao biti virtuelna funkcija** **(ovo nije problem, jer u trenutku poziva destruktora postoji objekat pa samim tim i v-table za objekat).**
- O ovome vodi računa programer, a ne kompajler!

Virtuelni destruktor - Primjer

- Posmatrajmo sljedeći generički primjer:

```
class B{//...
public: virtual ~B();
//...
};
class D:public B{
//...
public:
D();
virtual ~D();
//...
};
```

```
F-ja nečlanica:
void release(B *pb)
{delete pb;}
void main(){
B b;
D d;
B *b1=&d;
release(&b);
//poziva destruktor osn. klase
release(b1);
//poziva destruktor
//izvedene klase
}
```

Virtuelni konstruktor kopije

- Već smo rekli: konstruktor (pa ni konstruktor kopije) ne može biti virtuelna funkcija.
- Međutim, ponekad mi želimo da imamo fleksibilnost virtuelnog konstruktora kopije:
  - Na osnovu pokazivača ili reference na objekat osnovne klase koji pokazuje na objekat izvedene klase da kreiramo kopiju objekta izvedene klase.
- Da bi ovo postigli moramo da koristimo neki "obični" metod koji će praviti ovu kopiju (koja se ponekad naziva **klon**).

Klonirajući metod - Primjer

- U osnovnoj klasi (B) kreiramo metod sljedećeg oblika:

```
virtual B* Clone(){return new B(*this);}
```

dok u svim izvedenim klasama formiramo metode tipa:

```
virtual B* Clone(){return new D(*this);}
```

- Metod se poziva preko pokazivača na osnovnu klasu koji može da pokazuje na bilo koji objekat (osnovne ili izvedene klase) b->Clone() i vraća kopiju aktuelnog objekta.
- Naravno, metod se ne mora zvati Clone!

Apstraktne klase - Potreba

- Pretpostavimo sljedeću situaciju:
  - Pravimo pogram za kompjutersku 2D grafiku. Svaka klasa koju kreiramo mora da omogući računanje obima objekata i da ima mogućnost definisanja tipa i debljine linije koja ograničava objekat, te da ima mogućnost definisanja pomjeranja objekata. Sve objekte smo grupisali u dvije kategorije: Poligoni i Ovali. Očigledno da ćemo klase koje odgovaraju Poligonima i Ovalima staviti visoko u hijerarhiji nasljeđivanja i da ćemo iz njih izvoditi druge klase. Ali i obije ove klase imaju dosta toga zajedničkog (recimo pomjeranje, svi objekti imaju obim koji se računa različitim formulama, tip i debljina obvojnice). Stoga je zgodno ove dvije klase dalje generalizovati u jednu roditeljsku (osnovnu) klasu koja se naziva Oblik.

Oblik - Problem

- Za klasu Oblik ne možemo da definišemo objekte, jer svi objekti su ili Ovali ili Poligoni. Dakle, objekti ove klase će postojati samo kao podobjekti prethodnih klasa.
- Pored toga, za ovu klasu ne možemo da kreiramo funkciju za računanje obima jer takva funkcija za generalni oblik ne postoji (pretpostavimo da je tako).
- Ipak, sve klase izvedene iz klase Oblik imaju obim i zbog funkcionisanja virtuelnog mehanizma važno je da se ova f-ja može pozivati preko pokazivača na ovu osnovnu klasu, odnosno moramo imati nešto što će "glumiti" predmetnu funkciju u ovoj klasi.

Čiste virtuelne funkcije

- Predmetni problem se prevazilazi "čistim virtuelnim f-jama" koje za datu klasu nemaju realizaciju.
- Ovakve funkcije se deklarišu kao:
  virtual int Obim()=0;

  Označava da f-ja nema realizaciju za datu klasu, odnosno da je čista virtuelna f-ja.
- Ako klasa ima makar jednu čistu virtuelnu f-ju onda se naziva apstraktnom, a apstraktne klase nemaju objekte, osim kao podobjekte za izvedene klase.
- Ako klasa izvedena iz apstraktne klase ne predefiniše sve verzije čistih virtuelnih funkcija onda je i ona apstraktna!!!

C++ i Java i nasljeđivanje

- Java je popularni programski jezik koji podsjeća na C++, ali koji je namjenjen za distribuirano izvršavanje (program sa jednog računara se izvršava – interpretira na drugom računaru).
- Pored razlike u korišćenju pokazivača (jasno, ne smije se dozvoliti "brljanje" po pokazivačima na tuđem računaru), druga krupna razlika je u nepostojanju višestrukog nasljeđivanja kod Java-e.
- Ovo je nedostatak (uveden radi pojednostavljenja) kojeg Java prevazilazi uvođenjem drugih koncepata (što kasnije dovodi do usložnjavanja).

Nizovi i nasljeđivanje

- Niz objekata izvedene klase **nije** jedna vrsta niza osnovne klase.
- Stoga, ako želimo da koristimo polimorfizam (virtuelni mehanizam) kod nizova, moramo da koristimo pristup objektima preko pokazivača, a ne preko podataka članova niza.

Primjer nasljeđivanja

- Primjer nasljeđivanja će biti jednostavan. Imaćemo klasu Figure koja će imati jedan podatak član tipa Tačka **(moraćemo i ovu klasu da realizujemo u nekom rudimentarnom obliku)** koja će biti referentna tačka za dati oblik i čijim pomjeranjem će biti ostvarivano pomjeranje čitavog oblika. Klasa će imati i virtuelnu f-ju članicu Povrs().
- Iz predmetne klase izvodimo klase Kvadrat i Krug, dok iz njih izvodimo klase Pravougaonik i Elipsa.
- Napominjemo da teoretičari često ne preporučuju realizaciju među-klasa Kvadrat i Krug, jer se ove klase ne razlikuju dovoljno od onih izvedenih klasa koje imaju dodatu samo po još jednu osobinu. Mi ćemo ovdje to učiniti u svrhu ilustracije nasljeđivanja i virtuelnih funkcija.

Primjer - Komentar

- Naime, teorija preporučuje da se izvedena klasa kreira onda kada se između osnovne klase i izvedene klase može uspostaviti sljedeća veza:
  - Izvedena klasa je jedna vrsta osnovne klase.
- Ovdje Pravougaonik nije jedna vrsta Kvadrata, već obratno.
- Ne ulazeći u ova teorijska razmatranja mi ćemo izvršiti nasljeđivanje na opisani način kako bismo ilustrovali ovaj koncept, a bez želje da ga podrobnije analiziramo.
- Detaljnija teorijska razmatranja zahtijevaju obično poseban kurs koji bi se bavio OO analizom, modeliranjem i dizajnom.
- Neki teoretičari kažu da je ovo ipak dozvoljeno nasljeđivanje tipa: nasljeđivanje proširenjem.

Klasa Tacka za kontejner i Figure

```
class Tacka{
    private:
        double x,y;
    public:
        Tacka(double a=0.0, double b=0.0):x(a),y(b){}
        ~Tacka(){}
        void PomjeriTacku(Tacka t){
                x+=t.x;
                y+=t.y;} //f-ja pomjera tačku za
                //koordinate definisane tačkom t
        void Citaj(){cout<<"Koordinate tacke su: ("<<x<<","<<y<<")"<<endl;}
};
class Figure{
    private:
        Tacka ReferTacka;
    public:
        Figure(){}
        Figure(Tacka a):ReferTacka(a){}
        virtual ~Figure(){}
        void Pomjeri(Tacka t)
        {ReferTacka.PomjeriTacku(t);
        //nemamo pristup ReferTacka.x private je!!!
        }
        virtual double Povrs()=0;//ne umijemo je
        //izracunati dok ne vidimo koji je tip Figure
};
```

Klase Krug i Kvadrat

```
class Krug:public Figure{
    protected:
        double r;
    public:
        Krug(){}
        Krug(Tacka a,double b):Figure(a),r(b){}
        virtual ~Krug(){}
        virtual double Povrs() {return r*r*3.14;}
        };
class Kvadrat:public Figure{
    protected:
        double a;
    public:
        Kvadrat(){}
        Kvadrat(Tacka t,double d):Figure(t),a(d){}
        virtual ~Kvadrat(){}
        virtual double Povrs() {return a*a;}
        };
```

Klase Elipsa i Pravougaonik

```
class Elipsa:public Krug{
    private:
        double c;
    public:
        Elipsa(){}
        Elipsa(Tacka t,double d1, double d2):Krug(t,d1),c(d2){}
        virtual ~Elipsa(){}
        virtual double Povrs(){return c*r*3.14;}
        };

class Pravougaonik:public Kvadrat{
    private:
        double b;
    public:
        Pravougaonik(){}
        Pravougaonik(Tacka t,double d1, double d2):Kvadrat(t,d1),b(d2){}
        virtual ~Pravougaonik(){}
        virtual double Povrs(){return a*b;}
        };
```

Kako se pozivaju i zašto

```
int main()
{
    Figure *pf[4];
    Tacka t;//inicijalizuje tacku na (0,0)
    pf[0] = new Krug(t,1);
    pf[1] = new Kvadrat(t,1);
    pf[2] = new Elipsa(Tacka(2,2),1,2);
    pf[3] = new Pravougaonik(t,1,2);
    for(int i = 0;i<4;i++)
        cout<<pf[i]->Povrs()<<endl;
    for(int i = 0;i<4;i++)
        delete pf[i];
    cout << "Hello world!" << endl;
    return 0;
}
```

```
3.14
1
6.28
2
Hello world!

Process returned 0 (0x0)   execution time : 0.067 s
Press any key to continue.
```

Šabloni

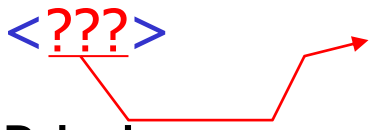
- Šabloni (engl. template) je treći, finalni, i ujedno najmoćniji alat za realizaciju koncepta **polimorfizma**.
- Koja su dva alata prethodno korišćena za realizaciju ovog koncepta?
- **Šablonizovati** se u C++ mogu **klase i funkcije**.
- Motivacija za uvođenje šablona polazi od sljedećeg.
  - Kreirali smo klasu lista sa cijelim brojevima koje upisujemo u elemente liste. Upotrijebili smo mnogo vremena i svo svoje programersko znanje da to uradimo. U tom trenutku pojavljuje se potreba da realizujemo klasu koja apstrahuje listu kompleksnih brojeva, pa zatim listu Studenata itd. Kako riješiti ovaj problem?

Šabloni - Motivacija

- Postojalo je više ideja kako prethodni problem prevazići:
  - Nekom vrstom kontejnera koji bi kombinovao klasu lista sa tipom podatka. Ovo se pokazalo nepraktičnim, jer je bilo nezgodno za implementiranje kod standardnih tipova podataka, a i veze sa pojedinim tipovima su se morale posebno uspostavljati. Odnosno, nije se pravila nikakva značajnija ušteda.
  - Drugi način bio je složeno nasljeđivanje lista i pojedinih tipova podataka, što opet nije moguće za standardne tipove i ponovo imamo veliku pisaniju u pogledu pisanja funkcija iz nasljeđenih klasa itd.
  - Jedini pravi način je da se napravi šablon. Klasa koja kao parametar uzima tip podatka. Napravimo šablon klase *Lista* i u zavisnosti od parametra koji joj proslijeđujemo dobijamo klasu cijelih brojeva, klasu *Studenata*, *Jabuka* ili bilo čega drugog. Na osnovu ovoga dolazimo do vrhunca reusability koncepta i do vrhunca jednostavnosti u održavanju koda: mnoštvo klasa kreiramo i editujemo na jednom mjestu.

Što se šablonizuje?

- Šablonizuju se klase i funkcije.
- Klase se mogu šablonizovati po tipu podataka, ali i po stvarnim objektima.
- Funkcije se mogu šablonizovati samo po tipovima podataka.
- Šablonska klasa ili funkcija se najavljuje ključnom riječju template



Parametri šablona.

- Primjer:

```
template <class T>
class Matrica{
private: T **m; int r,k;
public:
//razni metodi u kojima postoji tip T
};
```

Parametar T je tip podatka. Može biti i ugrađeni, ali i klasni.

typename ili class

- Postojalo je dosta primjedbi na upotrebu ključne riječi class da kod šablona označi tip podatka jer navodno to ne sugeriše da može biti i standardni (ugrađeni) tip podatka.
- Stoga, ANSI standard C++-a za ovu aplikaciju predviđa ključnu riječ: typename.
- Treba, ipak, voditi računa da većina kompajlera podržava class, a manjina typename, pa to znači da je ipak uputnije koristiti riječ class do daljnjeg.

Instanca šablona

- Ako nam treba matrica cijelih brojeva mi ćemo je najaviti kao:

```
Matrica <int> MatrInteger;
```

dok, ako nam treba matrica studenata deklarišemo je **(pod uslovom da je tip podatka Student kreiran)** kao:

```
Matrica <Student> MatrStudent;
```

- Uvijek imajte na umu da su tipovi Matrica <int> i Matrica <Student> različiti tipovi podataka koji se kreiraju iz istog šablona, te da u njima dolazi do zamjene argumenta po kome je šablon parametrizovan tipom koji se umeće u zagrade <>.

Šablon – Dodatni podaci

- Šablon se ponekad naziva **generičkim mehanizmom**, jer pomoću njega generišemo više klasa ili funkcija.
- Šablon može imati i više od jednog parametra: template <class T,class P>.
- Funkcije članice šablonske klase često imaju argumente koji su tipa koji predstavlja parametar šablona. Pogledajmo primjer:

```
template <class T>
class Matrica {
private: T **m; int k,r;//....
public: Matrica(int,int);
T &operator()(int,int);
T det(Matrica <T> &) const;//...
};
```

konstruktor sa dva argumenta upotrebu konstruktora ćemo naknadno objasniti

funkcija kao argument uzima konkretnu matricu iz šablona

dozvoljeno, što ćemo tumačiti kasnije

Primjer nastavak
Objekat parametar šablona

- Pojedine matrice možemo deklarisati pozivom konkretnog konstruktora na sljedeći način:

```
Matrica <int> m1(7,3);
Matrica <complex> m2(1,2);
```

- Šablon klase **(za razliku od šablona f-je)** može imati argument koji je stvarni objekat. Na primjer, možemo imati klasu vektora date dužine:

```
template <class T,int N>
class vektor{
private:
T v[N];//upotreba objekta parametra šablona
public:
T& operator() (int i){if(i>=0 && i<N) return v[i];
// sada može vektor <int,5> v1; … v1(5)=3
else //neka poruka o grešci ili slično}
};
```

stvarni objekat koji je parametar šablona

Primjer objekta parametra šablona

- Tumačite sljedeće deklaracije:

```
vektor <complex,5> v1;
vektor <int,5> v2;
vektor <complex,6> v3;
```

- Zapamtite da su v1, v2 i v3 podaci različitih tipova: vektor <complex,5>, vektor <int,5> i vektor <complex, 6>, ali da se svi ovi tipovi generišu iz istog šablona.

Šablonske funkcije

- Parametar šablona funkcije može biti samo tip podatka (class ili typename), ali ne i stvarni memorijski objekat.
- Barem jedan od argumenata šablonske f-je mora biti parametrizovani tip podatka:

```
template <class T>
void f(T m)
```

- Ili, što može biti još interesantnije:

```
template <class T>
void sort(vektor <T>)
```

- U drugom slučaju argument je objekat šablonske klase vektor koja se generiše iz šablona klase za konkretnu "vrijednost" tipa T.

Šablonske f-je kao dio šablona klase

- Prethodno navedena činjenica da jedan argument šablonske f-je mora biti parametar šablona se kosi sa onim što smo uradili kod šablonske klase Matrica gdje smo imali funkciju koja je bila deklarisana kao:
  T &operator()(int,int);
- Ipak, ovdje pravilo nije ugroženo, jer je implicitni argument ove funkcije objekat za koji se funkcija poziva, a on je tipa Matrica <T>, odnosno parametarskog tipa.
- **Ono što nije dozvoljeno je situacija kada šablonska f-ja nečlanica nema kao argument parametar šablona!!!**

Šablonska f-ja – Razrješenje poziva

- Primjer nedozvoljene šablonske f-je je:

```
template <class T>
T create()
```

- Razlog je, ja se nadam, jasan. Kako f-ja nema argument, to se ne zna koja bi se funkcija od onih iz šablona pozvala prilikom poziva ove funkcije.
- Kod rada sa šablonima može da postoji i dodatni problem koji se ogleda u činjenici da, pored šablonske funkcije sa datim imenom, može postojati i nešablonska funkcija istog imena. Onda postoji problem koja će funkcija biti pozvana.

Razrješenje poziva

- U slučaju da imamo nešablonske funkcije sa istim imenom kao što je ime šablonske funkcije procedura za razrješenje poziva je sljedeća:
  - Prioritetno se izvršavaju nešablonske funkcije koje imaju potpuno poklapanje argumenata.
  - Sljedeće: traži se ona funkcija koja se može kreirati iz šablona koja ima potpuno poklapanje argumenata. U ovom koraku strogo nije dozvoljena nikakva konverzija podataka sa argumentima funkcije.
  - Ako se ne nađe funkcija za prva dva slučaja traži se funkcija koja ima najbolje poklapanje argumenata bilo iz šablona bilo nešablonska, s time da se ako dođe do dvosmislenosti kod pozivanja prijavljuje greška.

Primjer razrješenja poziva

- Pretpostavimo šablon:

```
template <class T>
void sort(vektor <T> &v){/**/}
```

a pored toga i realizovanu funkciju

```
void sort(vektor <int> &v){/**/}
```

- Ako se poziva f-ja za argument tipa vektor <int> pozvaće se druga realizacija, dok će se u svim ostalim slučajevima pozvati (za svaki drugi argument vektor <T>) realizacija koja je šablonizovana.
- Obratiti pažnju da druga reačizacija nije šablonska, `vektor <int>` je konkretan tip podataka.

Šabloni funkcija i klase

- Svaka f-ja šablonske klase je implicitno šablonska funkcija (ima implicitni objekat za koji se poziva, a to je vezano za parametar šablona).
- Prijateljske funkcije šablonske klase ne moraju biti šablonske funkcije.
- Interesantna je i memorijska reprezentacija šablonske funkcije. Kompajler kreira kod svih funkcija koje se generišu iz šablona, a koje se pozivaju **(pa čak i ako se koristi samo adresa te funkcije)** u okviru datog programa.
- Dakle, što je i logično, kompajler ne kreira sve moguće funkcije iz šablona, već samo one koje se koriste u programu.

Statički članovi kod šablona

- Šablonska klasa može da ima statičke podatke članove.
- Jedan statički podatak član dijele između sebe svi objekti koji su kreirani iz jedne instance klase.
- Recimo, svi objekti tipa Matrica <int> dijele jedan statički podatak član, dok objekti Matrica <complex> dijele drugi statički podatak član, koji je različit od onoga kojega dijele objekti tipa Matrica <int>.
- Funkcije mogu takođe da posjeduju statičke promjenljive. To su promjenljive koje žive u memoriji do kraja programa, a ne do kraja izvršavanja funkcije.
- Jedna statička promjenljiva se kreira za svaku funkciju iz šablona, a ne jedna za sve šablonske funkcije koje se mogu kreirati.

Realizacija f-ja van klase

- Do sada, sve funkcije iz šablonske klase smo realizovali unutar klase. Naravno, funkcije se mogu realizovati i van klase, s tim što se mora naglasiti da je u pitanju šablonska klasa:

```
template <class T>
tip_rez Sab_klasa<T>::naziv_fje(argumenti){
//…
}
```

- Studenti često griješe kada, po analogiji sa standardnim f-jama, gdje se klasa i konstruktor nazivaju isto, pišu:

```
template <class T>
klasa<T>::klasa<T>(argumenti){}
```

Primjer greške kod konstruktora

- Ovdje je pogrešno navesti po drugi put naziv tipa po kojem se vrši parametrizacija u zaglavlju klase, jer je taj tip implicitno poznat na osnovu klase iz koje konstruktor dolazi, odnosno iz prvog navođenja šablonizovanog tipa u zaglavlju:

```
template <class T>
klasa<T>::klasa(argumenti){}
```

- Konstruktor se ipak "komplikovano" eksplicitno poziva kao:

```
klasa <int> m1 = klasa<int>(argumenti);
```

Šabloni i prijatelji

- Šablonska klasa može deklarisati tri tipa prijatelja:
  - Uobičajenu klasu ili funkciju kao prijatelja:
  - Šablonsku klasu ili funkciju kao prijatelja;
  - Specificiranu klasu ili funkciju koja se može dobiti iz šablona.

- Primjeri. Pretpostavimo da imamo šablon klase
  template <class T> class X i u njemu deklarisano:
  friend void f1();
  Ovo znači da je funkcija f1() prijatelj svim klasama koje se mogu generisati iz šablona.

- Dalje, sljedeći primjer je deklaracija prijateljske f-je:

  friend void f2(X <T> &);
  znači da je za klasu X<float> prijateljska f-ja
  f2(X<float> &), ali ne i f2(X<Student> &).

Šabloni i prijatelji - Primjeri

- Ako u klasi X koja je šablonizovana po parametru (tipu) T imamo deklaraciju:

```
friend void A::f4();
```

imamo f-ju f4() iz klase A koja je prijatelj svih klasa koje se mogu generisati iz šablona X <T>.

- Deklaracija:

```
friend void C<T>::f5(X<T> &);
```

označava da je klasi X<float> prijatelj šablonska funkcija f5(X<float> &) koja je članica šablonske klase C<float>.

Šabloni i prijatelji - Primjeri

- Deklaracija:

  friend class Y;

  označava nešablonsku klasu koja je prijatelj svih klasa iz šablona. Dok je:

  friend class Z<T>;

  šablonska klasa prijatelj. Odnosno, klasi X<float> prijatelj je samo konkretna klasa koja se može generisati iz šablona (u ovom slučaju Z<float>, a nije recimo Z<Jabuka>).
- Česta greška koja se pravi je podrazumijevanje da je prijatelj šablona obavezno šablon što smo kroz primjere vidjeli.

Primjer deklaracije šablonske f-je

- Posmatrajmo sljedeći primjer:

```
template <class T>
class List{
friend void count();
//da li je dozvoljeno i ako jeste
//kome je prijatelj
friend T max(List <T>);
//šablonska f-ja prijatelj
friend void f(List *);
//česta greška jer tip List ne postoji
//već postoje samo tipovi koji se mogu
//generisati iz šablona
//...
};
```

Šabloni u praksi

- Šabloni su vrhunski OO koncept.
- Prilikom programiranja, relativno kasno se uočava potreba za šablonom (tek onda kada kreirate pojedinačne funkcije za neke od tipova sa kojima radite).
- Stoga programer početnik teško započinje rad sa šablonima.
- Srećna okolnost je činjenica da je potreba za šablonima uočena od strane vrhunskih programera prije više godina tako da postoje brojni kvalitetni i provjereni šabloni koji su na ovaj ili onaj način dostupni.

STL

- **STL** (standard template library) su kreirali programeri Alexander Stepanov i Meng Lee iz Hewlett-Packarda.
- Kasnije je značajan doprinos dao David Musser.
- Ovo je biblioteka šablona koja je prihvaćena od strane ANSI komiteta i predstavlja standardnu biblioteku za C++.
- Stoga, gotovo svi kompajleri posjeduju ovu biblioteku.
- Mora se priznati da je u inženjerskim aplikacijama teško osmisliti šablon koji već ne postoji u STL-u.

Šabloni iz STL-a

- STL šabloni uključuju šablone: vektora, matrica, listi, redova, mapa itd.
- Pored toga, šablonizovani su brojni algoritmi za sortiranje, pretraživanje, poređenje itd.
- Slobodno se može reći da je gotovo kompletno naše programersko znanje ugrađeno u STL.
- Na Internetu postoji mnoštvo podataka o STL-u, te vam ih preporučujem za proučavanje.
- STL je realizovan da omogući maksimalnu efikasnost izvršavanja, pa stoga posjeduje i neke nelogičnosti u realizaciji. STL je, međutim, memorijski zahtjevan, jer je memorijska složenost podređena efikasnosti izvršavanja, ali smo se na memorijske zahtjeve više-manje navikli u savremenim softverima.