Objektno orijentisano programiranje 1

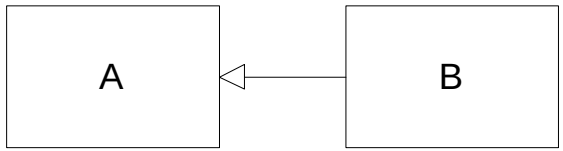
Izvođenje klasa

Pojam specijalizacije opšteg

- Jedna klasa objekata (B) može biti podvrsta neke druge klase (A)
- Klasa B je specijalni slučaj (*a-kind-of*) klase A
- Objekat klase B je (*is-a*) i objekat klase A
- Primeri:
  - Sisari su klasa koja je okarakterisana načinom ishrane mladunaca
    - Mesožderi su vrsta sisara koja se hrani mesom
    - Biljojedi su vrsta sisara koja se hrani biljkama
  - Geometrijske figure u ravni su klasa koja ima koordinate težišta
    - Krug je vrsta figure u ravni koja je okarakterisana dužinom poluprečnika
    - Kvadrat je vrsta figure u ravni koja je okarakterisana dužinom ivice
  - Vozila su klasa predmeta koji služe za prevoz
    - Teretna vozila su vrsta vozila namenja prevozu stvari i životinja
    - Putnička vozila su vrsta vozila namenjena prevozu ljudi

Izvođenje i nasleđivanje (1)

- Objekti klase B imaju sve osobine klase A i još neke specifične
- Specijalnija klasa B se izvodi iz generalnije (opštije) klase A
- Klasa B nasleđuje karakteristike klase A:
  - strukturne karakteristike (atribute)
  - karakteristike ponašanja (metode)
- Pored nasleđenih ima i sopstvene (specifične) članove
- Relacija između klasa B i A se naziva:
  - nasleđivanje (engl. *inheritance*)
  - generalizacija/specijalizacija
- Relacija nasleđivanja se prikazuje (usmerenim acikličnim) grafom (UML notacija):

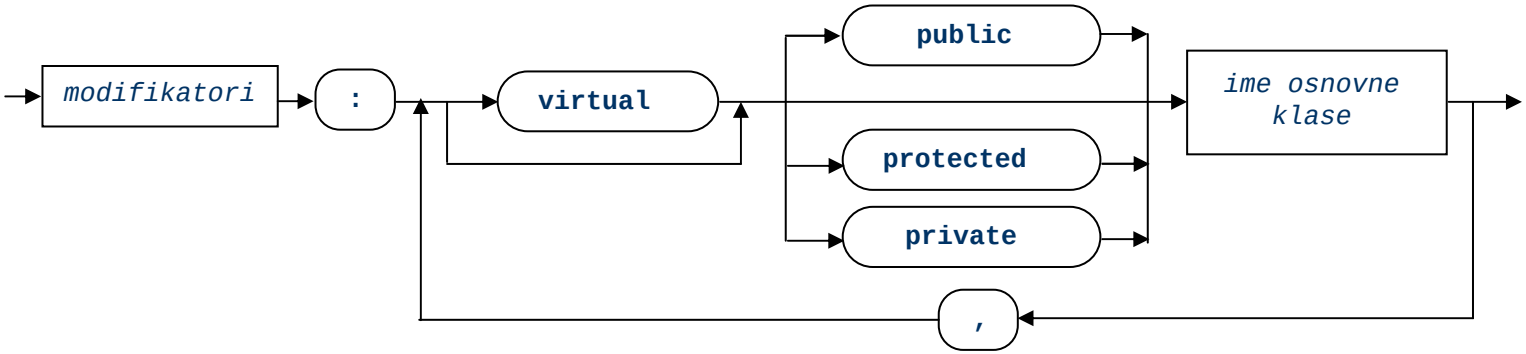


Izvođenje i nasleđivanje (2)

- Ako je klasa B specijalni slučaj (izvedena iz) klase A:
  - A je osnovna (*base*), a B izvedena (*derived*)
  - A je natklasa (*superclass*), a B potklasa (*subclass*)
  - A je roditelj (*parent*), a B dete (*child*)
- Objektni jezici koji podržavaju nasleđivanje
  - *objektno-orijentisani* (engl. *Object-Oriented*)
- Objektni jezici koji ne podržavaju nasleđivanje
  - *objektno-bazirani* (engl. *Object-Based*)
- Pri izvođenju nije potrebno vršiti
  - nikakve izmene postojeće klase
  - čak ni njeno ponovno prevođenje

Definisanje izvedene klase

- Izvedena klasa se definiše na sledeći način:
  `class` *`ime`* a zatim pre znaka `{`:



- Ako postoji samo jedna osnovna klasa – jednostruko izvođenje
- Ako postoji više osnovnih klasa u listi – višestruko izvođenje
- Izvođenje može biti i u više koraka:
  - izvedena klasa može biti osnovna klasa za sledeće izvođenje
  - takvo izvođenje se ne naziva višestrukim

Konačna klasa

- Modifikator `final` – konačna klasa
  - iz klase se ne može dalje izvoditi
  - list u stablu izvođenja
- Reč nije rezervisana, može se pojaviti i kao identifikator
  - ne preporučuje se
- Primer:

```
class A { ... };
class B { ... };
class C: A { ... };
class D: A, B { ... };
class E final: D { ... };
class F: E {...}; // ! GRESKA: klasa E je konacna
int final = 55; // final kao ime globalne promenljive
```

Primer izvođenja

- Objekti izvedene klase sadrže:
  - bezimeni podobjekat osnovne klase koji sadrži nasleđene članove
  - specifične članove navedene u definiciji izvedene klase

```
class Osnovna {
  int i;          // atribut osnovne klase
public:
  void f();       // metod osnovne klase
};

class Izvedena : public Osnovna {
  int j;          // atribut izvedene klase
public:           // nasledjen je: int i;
  void g();       // metod izvedene klase
};                // nasledjen je: void f();
```

Pristup nasleđenim članovima

- Preko objekta izvedene klase, javnim članovima osnovne klase se pristupa isto kao i članovima izvedene klase
- Primer:

```
int main () {
  Osnovna b;
  Izvedena d;
  b.f();
  b.g();    // ! GRESKA: g() je metod izvedene klase,
            // a b je objekat osnovne
  d.f();    // objekat izvedene klase d ima i metod f(),
  d.g();    // i metod g()
}
```

Konstruktor, destruktor, operator=

- Izvedena klasa ne nasleđuje iz osnovne klase:
  - konstruktore
  - destruktor
  - `operator=`
- Kao i za osnovne klase, postoje ugrađeni (implicitno definisani):
  - podrazumevani konstruktor koji ima prazno telo
  - kopirajući konstruktor koji kopira član po član (plitka kopija)
  - premeštajući konstruktor koji premešta član po član
  - destruktor koji ima prazno telo
  - kopirajući `operator=` koji vrši dodelu kopiranjem, član po član
  - premeštajući `operator=` koji vrši dodelu premeštanjem, član po član

Prava pristupa

- Članovi izvedene klase
  - imaju prava pristupa javnim članovima osnovne klase
  - nemaju prava pristupa privatnim članovima osnovne klase
- Posebno pravo pristupa - zaštićeno
- Labela `protected:` (u osnovnoj klasi)
  - označava deo klase kojem imaju pravo pristupa metodi te i izvedenih klasa
  - članovi u ovoj sekciji se nazivaju zaštićenim (*protected*)
- Zaštićenim članovima može da se pristupi iz metoda izvedene klase
  - ali samo kao nasleđenim članovima, a ne preko objekta osnovne klase

Primer prava pristupa

```
class Osnovna {
              int pb;
protected:    int zb;
public:       int jb;
};
class Izvedena : public Osnovna {
public:
  void m(int x) {
    jb=zb=x;// moze da pristupi javnom i zasticenom clanu,
    pb=x;   // ! GRESKA: privatnom clanu ne moze da pristupi
    Osnovna o;
    o.zb=x; // ! GRESKA: ne moze preko objekta osnovne klase
  }
};
void f() {
  Osnovna b;
  b.pb=1; // ! GRESKA: privatan clan
  b.zb=1; // ! GRESKA: zasticeni clan
  b.jb=1; // u redu – pristup javnom clanu
}
```

Načini izvođenja

- Način izvođenja određuje modifikator ispred imena osnovne klase
- Izvođenje može biti:
  - javno (`public`)
  - zaštićeno (`protected`)
  - privatno (`private`)
- Kaže se i za osnovnu klasu da je javna, zaštićena ili privatna
- Podrazumevano (bez modifikatora) izvođenje je privatno
- Način izvođenja određuje nasleđivanje prava pristupa:
  - određuje se stepen kontrole pristupa članovima osnovne klase preko objekta izvedene klase
  - objekat izvedene klase - može biti podobjekat u narednom izvođenju
  - ne utiče na pravo pristupa iz metoda dotične izvedene klase

Implicitna promena prava pristupa

Izvođenje	Član osnovne klase		
	javni	**zaštićen**	**privatan**
javno	javni	zaštićen	privatan
zaštićeno	zaštićen	zaštićen	privatan
privatno	privatan	privatan	privatan

Eksplicitna promena prava pristupa

- Može i eksplicitno da se promeni pravo pristupa nasleđenom članu
  - može da se uveća u odnosu na originalno (zaštićeni postaje javni)
  - može da se restaurira na originalno koje je umanjeno načinom izvođenja
  - može delimično da se restaurira
  - može i da se umanji u odnosu na originalno
- Ne može se promeniti pravo pristupa privatnom članu osnovne klase
  - jer se privatnom članu ne može pristupiti iz izvedene klase
- Postiže se uvozom člana u odgovarajuću sekciju izvedene klase
- Koristi se deklaracija `using` *`klasa::clan`*
  - može samo za članove za koje postoji pravo pristupa (ne za privatne u osnovnoj)
  - na taj način se može napraviti izuzetak za pojedini član od promene prava pristupa usled načina izvođenja
- Može (još uvek) i bez reči `using`, samo *`klasa::clan`*
  - da bi se obezbedila kompatibilnost sa ranijim standardom
  - zastarelo, izbegavati u novim programima

Primer

```
class O {
  int pb;
protected:
  int z1b,z2b;
public:
  int j1b,j2b;
};
class PI : O {
public:
  using O::j1b;     // vracanje originalnog prava pristupa
  using O::z1b;     // povecanje prava pristupa
  PI () { z2b=1; }  // moze se pristupiti iz izvedene klase
protected:
  using O::j2b;     // delimicno vracanje prava pristupa
};
int main () {
  PI pi;
  pi.j1b=0;         // javni clan klase PI
  pi.j2b=0;         // ! GRESKA: zasticen clan klase PI
  pi.z1b=0;         // javni clan klase PI
  pi.z2b=0;         // ! GRESKA: privatni clan klase PI
}
```

```
class IPI: public PI{
public:
  IPI(){
    j1b=0;      // OK
    j2b=0;      // OK
    z1b=0;      // OK
    z2b=0;      //! GRESKA
  }
}
```

Razlika privatnog i javnog izvođenja

- Između privatnog i javnog izvođenja – suštinska razlika
- Javno izvođenje realizuje koncept nasleđivanja
  - relacija "B je vrsta A" (*a-kind-of*)
  - izvedena klasa zadržava interfejs roditelja
  - objekat izvedene klase može da zameni objekat osnovne
  - to je jedino pravo izvođenje
- Privatno izvođenje realizuje koncept sadržanja
  - relacija "A je deo B" (*a-part-of*)
  - semantički slično sa slučajem kada B sadrži atribut tipa A
  - nije nasleđen interfejs roditelja - javni (i zaštićeni) članovi postali su privatni
  - objekat izvedene klase ne može da zameni objekat osnovne
- Zaštićeno izvođenje
  - unutar izvedene klase – koncept nasleđivanja
  - izvan izvedene klase – koncept sadržanja

Sakrivanje članova

- Redefinisanje identifikatora člana osnovne klase u izvedenoj klasi
  - sakriva identifikator iz osnovne
- Pristup sakrivenom članu iz osnovne klase:
  ime_osnovne_klase`::`*clan*
- Ako se u izvedenoj klasi napiše neki metod
  koji ima isto ime kao metod iz osnovne klase
  - takav metod sakriva sve nasleđene metode sa istim imenom
  - iz metoda izvedene klase se ne može direktno pristupiti sakrivenim metodima
  - nasleđeni sakriveni metodi se ne mogu pozivati ni za objekat izvedene klase
    - osim preko pokazivača na osnovu koji pokazuje na objekat izvedene klase
- U izvedenoj klasi ne može se (podrazumevano) računati
  sa preklapanjem imena metoda iz osnovne klase

Uvoz članova

- U izvedenoj klasi sve sakrivene metode treba
  - redefinisati ili
  - restaurirati (otkriti)
- Nije dobro da izvedena klasa sadrži samo neke metode iz osnovne klase
  - ne radi se o pravom nasleđivanju
  - korisnik izvedene klase očekuje isti (ili prošireni) interfejs osnovne klase
- Uvoz – otkrivaju se svi sakriveni metodi sa datim imenom
  `using` *`osnovna_klasa`*`::`*`ime_metoda`*
  - nisu vidljivi samo metodi osnovne sa istim potpisom kao u izvedenoj klasi
- Ne može da se restaurira vidljivost pojedinačnog metoda osnovne klase

Sakrivanje i uvoz članova – primer

```
class O { public:
  int a=1;
  void m1();
  void m1(int);
  void m2();
  void m2(int);
};
class I: public O {
public:
  using O::m2;
  int a=2;
  void m1(int);
  void m2(int);
  void m(){
    int x=a;   // x=I::a
    int y=O::a;// x=O::a
    m1();      // ! GRESKA
    m1(x);     // I::m1(int)
    O::m1();   // O::m1();
    O::m1(y);  // O::m1(int);
    m2();    // O::m2();
    m2(x);   // I::m2(int)
    O::m2(y);// O::m2(int);
  }
};
void f(){
  I i;
  int p=i.a;     // I::a
  int q=i.O::a; // O::a
  i.m1();        //! GRESKA
  i.O::m1();     // O::m1()
  i.m1(p);       // I::m1(int)
  i.O::m1(q);    // O::m1(int)
  i.m2();        // O::m2()
  i.m2(p);       // I::m2(int)
  i.O::m2(q);    // O::m2(int)
}
```

Uvoz konstruktora

- Konstruktori osnovne klase se ne nasleđuju
- Može se izvršiti uvoz (svih) konstruktora osnovne klase
  using *ime_klase*::*ime_klase*
  - na taj način se generišu konstruktori izvedene sa istim potpisima
  - generisani konstruktori izvedene klase imaju prazno telo
  - oni samo pozivaju konstruktore osnovne sa istim potpisom
- Primer:

```
class A { public: A(int i){…} };
class B: public A {public: using A::A; }
B b(1); // poziva se A::A(1)
```

Izvođenje struktura

- Strukture su ravnopravne sa klasama
  - moguće je izvođenje strukture iz klase ili iz strukture
    - podrazumeva se javno
  - moguće je i izvođenje klase iz strukture
    - podrazumeva se privatno
- Primer:

```
struct OS { };
class OK { };
class IKS: OS{ };  // privatno izvodjenje
class IKK: OK{ };  // privatno izvodjenje
struct ISK: OK{ }; // javno izvodjenje
struct ISS: OS{ }; // javno izvodjenje
```

- Unije (**union**) se ne mogu izvoditi, niti se iz unije može izvoditi

Konstruktori i destruktori

- Prilikom kreiranja objekta izvedene klase,
  poziva se konstruktor te klase, ali i konstruktor osnovne klase
- Prilikom uništavanja objekta izvedene klase,
  poziva se destruktor te klase, ali i destruktor osnovne
- Prenos parametara konstruktoru osnovne klase:
  - u listi inicijalizatora konstruktora izvedene klase
    moguće je navesti i inicijalizator osnovne klase
  - inicijalizator osnovne klase se sastoji od imena osnovne klase
    i argumenata poziva konstruktora osnovne klase
- Nije moguće vršiti inicijalizaciju pojedinih nasleđenih atributa

Primer inicijalizacije osnovne klase

```
class Osnovna {
  int bi;
public:
  Osnovna(int);  // konstruktor osnovne klase
};
Osnovna::Osnovna (int i) : bi(i)
{/*...*/}

class Izvedena : public Osnovna {
  int di;
public:
  Izvedena(int);
};
Izvedena::Izvedena (int i) : Osnovna(i),di(i+1)
{/*...*/}
```

Redosled konstrukcije

- Pri kreiranju objekta izvedene klase redosled akcija je sledeći:
  - inicijalizuje se podobjekat osnovne klase
    - pozivom odgovarajućeg konstruktora osnovne klase
    - za višestruko izvođenje - po redosledu navođenja osnovnih klasa u definiciji izvedene, ne u listi inicijalizatora
  - inicijalizuju se specifični atributi
    - po redosledu navođenja u definiciji klase, ne u listi inicijalizatora
    - klasni tipovi pozivom njihovih odgovarajućih konstruktora
    - ugrađeni tipovi na osnovu inicijalizatora
    - nizovi objekata po rastućim vrednostima indeksa podrazumevanim konstruktorom
  - izvršava se telo konstruktora izvedene klase
- Pri uništavanju objekta izvedene klase redosled poziva destruktora je uvek obratan

Primer redosleda

```
class A {
public:
  A() {cout<<"Konstruktor A."<<endl;}
  ~A() {cout<<"Destruktor A."<<endl;}
};

class O {
public:
  O() {cout<<"Konstruktor O."<<endl;}
  ~O() {cout<<"Destruktor O."<<endl;}
};

class I : public O {
  A a;
public:
  I() {cout<<"Konstruktor I."<<endl;}
  ~I() {cout<<"Destruktor I."<<endl;}
};

int main () { I d; }
```

Izlaz:

```
Konstruktor O.
Konstruktor A.
Konstruktor I.
Destruktor I.
Destruktor A.
Destruktor O.
```

Konverzija

- Objekat javno izvedene klase je i objekat osnovne klase
- Posledice su:
  - pokazivač na objekat izvedene klase se može implicitno konvertovati u pokazivač na objekat osnovne klase
  - pokazivač na objekat osnovne klase se može samo eksplicitno konvertovati u pokazivač na objekat izvedene klase
  - isto važi i za reference
  - objekat osnovne klase se može inicijalizovati objektom izvedene klase
  - objektu osnovne klase može se dodeliti objekat izvedene klase
- Objekat privatno/zaštićeno izvedene klase nije i objekat osnovne klase
  - pokazivač na objekat takve klase se može implicitno konvertovati u pokazivač na objekat osnovne klase samo unutar izvedene klase

Pojam polimorfizma

- Potrebno je projektovati klasu geometrijskih figura `Figura`
  - sve figure treba da imaju metod `crtaj()`
- Iz klase geometrijske figure se izvode klase kruga, kvadrata, trougla itd.
- Izvedena klasa treba da realizuje metod crtanja na sebi svojstven način
- U nekom delu programa sve figure koje se nalaze na crtežu treba nacrtati
- C++ omogućava da figure jednostavno iscrtamo na sledeći način:

```
void crtanje (int br_figura, Figura** niz_figura){
  for (int i=0; i<br_figura; i++)
    niz_figura[i]->crtaj();
}
```

- Napomena: figurama se pristupa preko niza pokazivača na figure

Decentralizacija odgovornosti

- Bitno u prethodnom primeru: decentralizacija odgovornosti
  - program koji crta sve figure ne zna kako se crta pojedina figura
- Svaki objekat će "prepoznati" kojoj izvedenoj klasi pripada, iako mu se obraćamo kao objektu osnovne klase
- Polimorfizam: svaki objekat izvedene klase izvršava metod onako kako je to definisano u njegovoj izvedenoj klasi, iako mu se pristupa kao objektu osnovne klase
- Polymorphism = Poly (više) + Morph (oblik)
  - isti metod ima više "oblika"
  - tip objekta za koji se zove metod određuje koji metod se poziva
  - mehanizam je potpuno dinamički – klasa objekta se određuje pri izvršavanju programa, a ne pri prevođenju

Virtuelni (polimorfni) metodi

- Virtuelni metodi:
  - metodi osnovne klase koji se u izvedenim klasama mogu redefinisati, a ponašaju se polimorfno
  - metod u izvedenoj nadjačava (eng. *override*) metod osnovne klase
- Polimorfna klasa:
  - klasa koja sadrži barem jedan virtuelni (polimorfni) metod
- Virtuelni metod se u osnovnoj klasi deklariše:
  - pomoću ključne reči `virtual` na početku deklaracije
- Prilikom deklarisanja virtuelnih metoda u izvedenim klasama:
  - ne mora se stavljati reč `virtual`
- Pozivom preko pokazivača/reference na osnovnu klasu izvršava se onaj metod koja pripada klasi pokazanog objekta

Redefinisanje virtuelnih metoda

- Deklaracija virtuelnog metoda u izvedenoj klasi
  - mora da se potpuno slaže sa deklaracijom istog u osnovnoj klasi
  - potpuno slaganje znači:
    - da se podudaraju broj i tipovi argumenata, kao i tip rezultata
  - izuzetno, ako je tip rezultata referenca/pokaz. na osnovnu klasu
    - redefinisan metod može da vraća referencu/pokazivač na javno izvedenu klasu iz date osnovne
- Neslaganje potpisa istoimenog virtuelnog metoda – sakrivanje met.
- Samo razlika u tipu rezultata (osim navedene) – greška
- Virtuelni metod osnovne klase ne mora da se redefiniše u svakoj izvedenoj klasi
  - u izvedenoj klasi u kojoj nije redefinisan, važi nasleđen virtuelni metod iz osnovne klase

Primer virtuelnog metoda

```
class ClanBiblioteke {
protected:
  Racun r;
public:
  virtual Racun platiClanarinu () // virtuelni metod osnovne klase
    { return r-=CLANARINA; }
};

class PocasniClan : public ClanBiblioteke {
public:
  Racun platiClanarinu ()         // virtuelni metod izvedene klase
    { return r; }
};

int main () {
  ClanBiblioteke *clanovi[100];
  //...
  for (int i=0; i<brojClanova; i++)cout<<clanovi[i]->platiClanarinu();
}
```

Pozivanje virtuelnih metoda

- Virtuelni mehanizam se aktivira samo ako se objektu pristupa indirektno (preko reference ili pokazivača):

```
class Osnovna { public: virtual void f(); };
class Izvedena : public Osnovna { public: void f(); };

void g1(Osnovna b)   { b.f(); }
void g2(Osnovna *pb) { pb->f(); }
void g3(Osnovna &rb) { rb.f(); }

int main () {
  Izvedena d;
  g1(d);                                  // poziva se Osnovna::f
  g2(&d);                                 // poziva se Izvedena::f
  g3(d);                                  // poziva se Izvedena::f
  Osnovna *pb=new Izvedena; pb->f();      // poziva se Izvedena::f
  Osnovna &rb=d; rb.f();                  // poziva se Izvedena::f
  Osnovna b=d; b.f();                     // poziva se Osnovna::f
  delete pb; pb=&b; pb->f();              // poziva se Osnovna::f
}
```

Modifikatori `override` i `final`

- Implicitna virtuelnost metoda u izvedenoj klasi nije robusna
- Kao modifikator (iza liste parametara) može da se navede `override`
  - eksplicitno se iskazuje da metod nadjačava odgovarajući virtuelni metod
  - prevodilac prijavljuje grešku ako nije moguće nadjačavanje
    - ako u osnovnoj klasi ne postoji virtuelni metod sa istim potpisom i tipom rezultata
- Modifikator `final` sprečava nadjačavanje metoda u izvedenoj klasi
  - dozvoljen samo za (eksplicitno ili implicitno) virtuelni metod
- Modifikatori `override` i `final` nisu rezervisane reči
- Modifikatori `override` i `final` nisu deo potpisa metoda
- Moguća je kombinacija oba modifikatora `override` i `final`

Modifikatori - primer

```
class A { public:
  virtual void vm1();
  virtual void vm1(int);
  void m1();
};
class B: public A {public:
  void vm1() override;
  void vm1(int) override final;
  void vm1(float) override; // ! GRESKA – nema A::vm1(float)
  void vm2() override; // ! GRESKA – nema A::vm2()
  void m1() override;  // ! GRESKA – A::m1() nije virtuelna
  void m2() final;     // ! GRESKA – B::m2() nije virtuelna
};
class C: public B { public:
  void vm1(int); // ! GRESKA – B::vm1(int) je konacna
};
int override=1; int final=2; // U redu, nije preporucljivo
```

Modifikatori `explicit` i `new`

- Za izvedene klase sa modifikatorom `explicit`
  - modifikator `override` je obavezan za nadjačani metod
  - za član koji sakriva ime iz osnovne, a nije `override`, obavezan je `new`
- Klase sa modifikatorom `explicit` su robusnije
- Primer:

```
class A { public:
  virtual void vm1();
  virtual void vm2();
  void m();
  int x,y;
};
class B explicit: public A { public:
  void vm1() override;
  void vm1(int) new;
  void vm1(float);      // ! GRESKA – nema new
  void vm2();           // ! GRESKA – nema ni override ni new
  void m();             // ! GRESKA – nema new
  int x new;
  int y;                // ! GRESKA – nema new
};
```

Dinamičko vezivanje

- Dinamičko vezivanje (engl. *dynamic binding*):
  - mehanizam koji obezbeđuje da se metod koji se poziva određuje:
    - po tipu objekta,
    - ne po tipu pokazivača ili reference na taj objekat
- Odlučivanje koji će se virtuelni metod (iz koje klase) pozvati
  - obavlja se u toku izvršavanja programa - dinamički
  - bitna je razlika u odnosu na mehanizam preklapanja imena funkcija
    - mehanizam preklapanja imena funkcija je statički

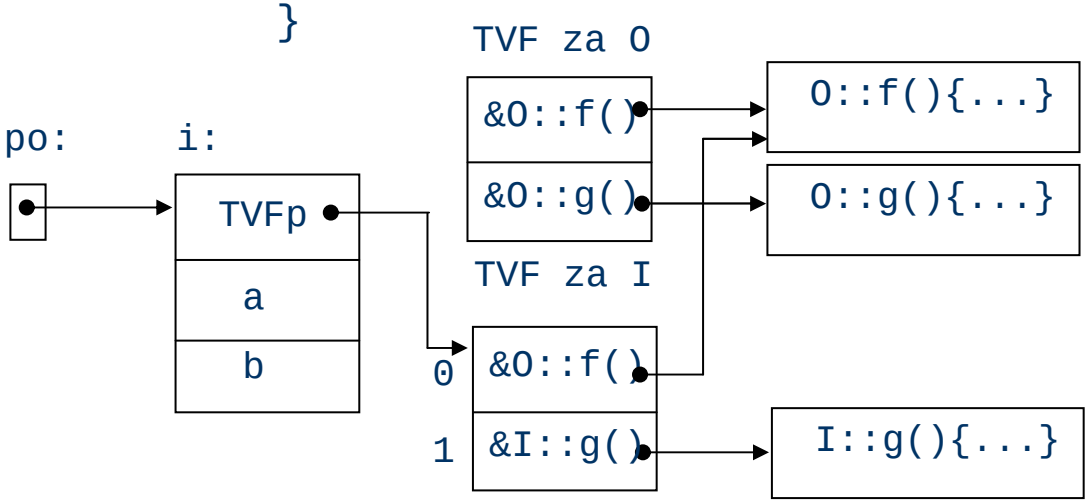
Implementacija virtuelnih poziva

- Za jednostruko izvođenje mehanizam je sledeći:
  - za virtuelne (kao i za nevirtuelne) metode svake nove klase
    - jedinstven kod metoda
  - za svaku polimorfnu klasu:
    - tabela pokazivača na virtuelne metode te klase (tabela virtuelnih funkcija - TVF)
  - svaki objekat polimorfne klase sadrži pokazivač na TVF

Primer – virtuelni mehanizam

```
class O{
public:
  int a;
  virtual void f();
  virtual void g();
}
class I:public O{
  int b;
public:
  void g();
}
```

```
int main(){
  I i;
  O *po=&i;
  po->g(); // (*(po->TVFp[1]))(po)
  // ...
}
```



Osobine polimorfnih metoda

- Virtuelni (polimorfni) metodi ne mogu biti statički
- Ako je potreban statički metod sa polimorfnim ponašanjem
  - ovaj metod treba da poziva nestatički virtuelni metod za neki objekat
  - pokazivač ili referenca na objekat čiji se metod poziva može da se prenese kao argument poziva statičkog metoda
- Primer:

```
class X{
public:
   virtual void m()const{};
   static void s(const X&);
};
void X::s(const X& x){ x.m();}
```

- Globalne prijateljske funkcije ne mogu biti polimorfne
  - ako je potrebno polimorfno ponašanje – sličan recept kao za statičke metode
- Virtuelni metodi mogu biti prijatelji drugih klasa

Virtuelni destruktor

- Konstruktor ne može biti virtuelna funkcija
  - jer se poziva pre nego što se objekat kreira
- Destruktor može da bude virtuelna funkcija
  - ako je destruktor virtuelna funkcija,
    tek u vreme izvršenja se odlučuje koji destruktor se poziva
- Kada se uništava dinamički kreirani objekat (pomoću `delete p`)
  - destruktor osnovne klase se uvek izvršava
  - ili kao jedini ili posle destruktora izvedene klase
- Kada neka klasa ima neku virtuelnu funkciju,
  verovatno i njen destruktor (ako ga ima) treba da bude virtuelan
- Unutar destruktora izvedene klase
  - ne treba pozivati destruktor osnovne klase (implicitno se poziva)

Primer virtuelnog destruktora

```
class OVD { public: virtual ~OVD(); };
class IVD: public OVD { public: ~IVD(); };

class OnVD{ public: ~OnVD(); };
class InVD: public OnVD { public: ~InVD(); };

void oslobodi (OVD *pb) { delete pb; }
void oslobodi (OnVD *pb) { delete pb; }

int main () {
  OVD *pb=new OVD; oslobodi(pb);      // ~OVD()
  IVD *pd=new IVD; oslobodi(pd);      // ~IVD(), ~OVD()
  OnVD *pbn=new OnVD; oslobodi(pbn);  // ~OnVD()
  InVD *pdn=new InVD; oslobodi(pdn);  // ~OnVD()
}
```

Nizovi i izvođenje

- U jeziku C++ niz objekata nije objekat
- Niz objekata izvedene klase nije jedna vrsta niza objekata osnovne klase
- Ako se niz objekata izvedene klase prenese funkciji kao niz objekata osnovne klase, može doći do greške
- Primer:

```
class Osnovna { public: int bi; };
class Izvedena : public Osnovna { public: int di; };
void f(Osnovna *b, int i) { cout<<b[i].bi; }
int main () {
  Izvedena d[5];
  d[2].bi=77;
  f(d,2);              // nece se ispisati 77
}
```

  - objekti osnovne klase su manji od objekata izvedene klase
  - funkcija `f` smatra da je dobila niz objekata osnovne klase
  - kada joj se prosledi niz objekata izvedene klase, nema načina da to odredi

Prenos zbirki objekata funkcijama

- Ako se računa sa nasleđivanjem, pravilo:
  - u programu ne treba koristiti nizove objekata, već nizove pokazivača na objekte

```
void f(Osnovna **b, int i) { cout<<b[i]->bi; }
int main () {
  Osnovna b1;  Izvedena d1,d2;
  Osnovna *b[5];                  // b se moze konvertovati u Osnovna**
  b[0]=&d1; b[1]=&b1; b[2]=&d2; d2.bi=77;
  f(b,2);                         // ispisace se 77
}
```

- Nije dozvoljena konverzija `Izvedena**` u `Osnovna**`
- Za prethodni primer nije dozvoljeno:

```
int main () {
  Izvedena *d[5];                 // d je tipa Izvedena**
  f(d,2);                         // ! GRESKA: pokusana konverzija
                                  //           Izvedena** u Osnovna**
}
```

Apstraktni metodi i klase

- Apstraktan (čist virtuelni) metod:
  - virtuelni metod koji nije definisan za osnovnu klasu
- Deklaracija apstraktnog metoda u osnovnoj klasi:
  - umesto tela stoji `=0`
- Klasa koja sadrži barem jedan apstraktni metod naziva se *apstraktnom klasom* (engl. *abstract class*)
- Apstraktna klasa ne može imati primerke (objekte), već se iz nje samo mogu izvoditi druge klase
- Mogu da se formiraju pokazivači i reference na apstraktnu klasu
- Pokazivači i reference na apstraktnu klasu mogu da ukazuju samo na objekte izvedenih konkretnih klasa

Primer apstraktne klase

```
class Osnovna {
public:
  virtual void cvf () =0;  // apstraktni metod
  virtual void vf ();      // virtuelni metod
};

class Izvedena : public Osnovna {
public:
  void cvf();
};

int main () {
  Izvedena izv, *pi=&izv;
  Osnovna osn;      // ! GRESKA: Osnovna je apstraktna klasa
  Osnovna *po=&izv;
  po->cvf();        // poziva se Izvedena::cvf()
  po->vf();         // poziva se Osnovna::vf()
}
```

Apstraktni destruktor

- Kada treba sprečiti stvaranje objekata klase čiji su metodi konkretni
  - deklariše se apstraktni destruktor
- Apstraktni destruktor mora biti definisan i za osnovnu klasu
  - definicija mora biti van tela klase
- Primer:

```
class O {
public:
  virtual ~O()=0;
};
O::~O(){}
class I:public O {};
O o; // ! GRESKA
I i; // U redu
```

Generalizacija i konkretizacija

- Apstraktna klasa predstavlja generalizaciju izvedenih klasa
- Primer:
  - sve geometrijske figure treba da mogu da odrede svoju površinu
  - u osnovnoj klasi `Figura` se ne zna kako se uopšteno izračunava površina
  - svaka izvedena klasa iz klase `Figura` će znati kako se izračunava površina
- Klasa koja se izvodi iz apstraktne je jedna konkretizacija te klase
- Ako se u izvedenoj klasi iz apstraktne ne navede definicija nekog apstraktnog metoda iz osnovne klase i izvedena klasa je apstraktna
- Apstraktna klasa može imati konstruktor
  - iako se ne mogu konstruisati objekti apstraktne klase
  - on će se pozivati kao konstruktor osnovne klase pri konstrukciji objekata konkretnih klasa koje se izvode iz apstraktne klase

Problem konverzije naniže

- Problem:
  - preko pokazivača/reference na osnovnu klasu nije moguće pristupiti članovima izvedene klase
  - samo ako znamo tačno kog tipa je objekat na koji ukazuje pokazivač/referenca na osnovnu klasu
    - možemo eksplicitno konvertovati pokazivač na osnovnu u pokazivač na izvedenu klasu
  - ovakva konverzija se naziva konverzijom naniže *(downcast)*
  - konverzija naniže nije bezbedna
- Rešenje:
  - da se uradi konverzija pokazivača/reference naniže ali da se pri tome proveri njena ispravnost
    - postiže se operatorom za dinamičku konverziju tipa

Operator dinamičke konverzije

- Izrazi za dinamičku konverziju tipa:
  `dynamic_cast``<izvedena klasa*>(pokazivač)`
  `dynamic_cast``<izvedena klasa&>(referenca)`
  - konvertuju pokazivač/referencu na polimorfnu osnovnu klasu u pokazivač/referencu na izvedenu klasu
- Ako pokazivač ne pokazuje na objekat izvedene klase u koju se vrši konverzija, ili njene potklase:
  - rezultat je `nullptr`
- Ako referenca ne upućuje na objekat izvedene klase u koju se vrši konverzija, ili njene potklase:
  - baca se izuzetak `bad_cast` (definisan u `<typeinfo.h>`)

Primer dinamičke konverzije

```
class A {public: virtual void vm(){}}; // polimorfna klasa
class B: public A{/*...*/};
class C: public A{/*...*/};
int main(){
  A *pa=new B();
  B *pb=dynamic_cast<B*>(pa);
  C *pc=dynamic_cast<C*>(pa); //pc==nullptr
  B &rb=dynamic_cast<B&>(*pa);
  C &rc=dynamic_cast<C&>(*pa); // bad_cast
}
```

Dinamičko određivanje tipa

- Jezik C++ omogućava da se u vreme izvršavanja odredi tip izraza
  `typeid``(`*`izraz`*`)`
  `typeid``(`*`tip`*`)`
  - *`izraz`*:
    - može biti proizvoljnog tipa:
      - standardnog/ korisničkog kao i jednostavnog/složenog
    - vrednost se ne izračunava, nema ni bočnih efekata
    - ako ukazuje na objekat polimorfne klase,
      rezultat se odnosi na dinamički tip operanda, inače na statički
    - ako je u izrazu **`typeid`**`(*p) p==nullptr` ➔ izuzetak `bad_typeid`
  - *`tip`*: proizvoljan tip
  - rezultat: **`const`** `type_info&`
- Introspekcija ili refleksija: način da program saznaje o sebi

Klasa `type_info`

- `type_info` je tip iz `<typeinfo>` i sadrži podatke o tipu
- Metodi klase `type_info`:

```
bool operator==(const type_info&) const;
bool operator!=(const type_info&) const;
bool before(const type_info&) const;
    // uređuje dva objekta tipa type_info (zavisi od prev.)
const char* name()const;
    // vraća neko logično ime tipa (zavisi od prevodioca)
```

- Tip `type_info` nema javne konstruktore
  - ne mogu se kreirati podaci ovog tpa
  - razlog zašto se ne mogu prenositi podaci ovog tipa po vrednosti
- Operator dodele za tip `type_info` je takođe privatan

Primer određivanja tipa

```
class O {};      // nepolimorfna klasa
class I: public O{public: virtual void f(){}};
class II: public I{};
int main(){
  O *po=new I;
  I *pi=new II;                          //Izlaz (Borland):
  cout<<(typeid(*po)==typeid(I))<<endl;  //0
  cout<<(typeid(*pi)==typeid(II))<<endl; //1
  cout<<typeid(O).name()<<endl;          //O
  cout<<typeid(*po).name()<<endl;        //O
  cout<<typeid(*pi).name()<<endl;        //II
  cout<<typeid(po).name()<<endl;         //O *
  cout<<typeid(pi).name()<<endl;         //I *
  int m[100][20];
  cout<<typeid(m).name()<<endl;          //int[100][20]
}
```

Zloupotreba određivanja tipa

- Objektno-orijentisane programe treba koncipirati tako:
  - da se pišu polimorfni metodi koji decentralizuju odgovornosti
    - odgovornosti se delegiraju pojedinim apstrakcijama
- Određivanje tipa krije zamku:
  - kada se odredi tip objekta on se može porediti sa klasama
    - za dati tip objekta se poziva nepolimorfni metod date klase
  - polimorfizam se zamenjuje selekcijom
  - program ponovo postaje sa centralizovanom odgovornošću
  - smanjuje se mogućnost jednostavnog održavanja koda

Višestruko nasleđivanje

- Višestruko nasleđivanje (engl. *multiple inheritance*):
  - kada klasa direktno nasleđuje osobine više osnovnih klasa
  - roditeljske klase nisu jedna drugoj vrsta
- Primer:
  - konj je životinja, ali je i prevozno sredstvo
  - pri tome ni životinja nije vrsta prevoznog sredstva, ni obrnuto
- Klasa se deklariše kao naslednik više klasa
  tako što se u zaglavlju navode osnovne klase
  - ispred svake osnovne klase treba da stoji reč `public`,
    da bi izvedena klasa nasleđivala prava pristupa članovima

```
class I:public O1, public O2, public O3 {/*...*/};
```

Konstrukcija i destrukcija

- Pravila o nasleđenim članovima važe i ovde
- Konstruktori svih osnovnih klasa se izvršavaju pre
  - konstruktora članova izvedene klase i
  - konstruktora izvedene klase
- Konstruktori osnovnih klasa se pozivaju po redosledu deklarisanja
- Destruktori svih osnovnih klasa se izvršavaju posle
  - destruktora izvedene klase i
  - destruktora članova izvedene klase
- Destruktori osnovnih klasa se pozivaju obrnutim redom

Problem "dijamant strukture"

- Problem:
  - kada su osnovne klase pri višestrukom izvođenju, izvedene iz iste roditeljske klase
- Primer:

```
class B {int i; /*...*/};
class X : public B {/*...*/};
class Y : public B {/*...*/};
class Z : public X, public Y {/*...*/};
```

- Svaka od klasa X i Y ima po jedan primerak članova klase B
  - objekat `Z z;` će imati dva skupa članova klase B
  - njih je moguće razlikovati pomoću operatora `::` `z.X::i` ili `z.Y::i`
- Konstruktor osnovne klase B se izvršava dva puta

Virtuelne osnovne klase

- Ako ovo nije potrebno, klasu B, pri izvođenju iz nje, treba deklarisati kao virtuelnu osnovnu klasu:

```
class B {int i; /*...*/};
class X : virtual public B {/*...*/};
class Y : virtual public B {/*...*/};
class Z : public X, public Y {/*...*/};
```

- Sada klasa Z nasleđuje samo jedan skup članova klase B
  - nema dvoznačnosti u pristupu članovima nasleđenim iz B
- Konstruktor klase B se poziva samo jednom
- Klasa B mora biti virtuelna osnovna i za X i za Y
  - ako je samo za jednu virtuelna - ostaju dva skupa članova i 2x konstruktor
- Redefinicija metoda m( ) definisanog u klasi B
  - ili u klasi X ili u klasi Y – za z.m( ) se poziva data redefinicija (nije problem)
  - i u klasi X i u klasi Y – z.m( ) je dvoznačno (problem)

Redosled konstrukcije

- Konstruktori virtuelnih osnovnih klasa
  se pozivaju pre konstruktora nevirtuelnih osnovnih klasa
- Precizan redosled izvršavanja konstruktora:
  - konstruktori virtuelnih osnovnih klasa
    - po dubini grafa izvođenja (najpre koreni)
    - sleva-udesno na istom nivou grafa
  - konstruktori nevirtuelnih osnovnih klasa
  - konstruktori atributa
  - konstruktor izvedene klase
- Konstruktori virtuelnih osnovnih klasa se pozivaju samo jednom

Primer redosleda konstrukcije

```
#include <iostream.h>
class B {
public: B(){cout<<" B";}
};

class X : virtual public B {
public: X(){cout<<" X";}
};

class Y : virtual public B {
public: Y(){cout<<" Y";}
};

class Z : public X, public Y {
public: Z(){cout<<" Z"<<endl;}
};

int main(){Z z;}
```

```
// Originalni redosled:       // B X Y Z

// Da je stajalo (jedna po jedna zamena):
class X: public B {…          // B B X Y Z

class Y: public B {…          // B X B Y Z

class Z: public X,
         virtual public Y {…  // B Y X Z

class Z: virtual public X,
         virtual public Y {…  // B X Y Z
```