Osnovi računarskih sistema - Nasleđivanje

Milena Vujošević i Jelena Tomašević

Čas 1

Vežbe—— 10 12 2004

1.1 Nasleđivanje

Nasleđivanje je jedan od osnovnih mehanizama u C++. Nasleđivanje omogućava da se nova klasa opiše uz pomoć neke postojeće klase. Nova klasa će preuzeti sve što joj odgovara iz stare klase i promeniti ili dopuniti preostalo. Nasleđivanje omogućava korišćenje već napisanog kôda na jednostavan i prirodan način.

1.2 Sintaksa nasleđivanja

```
class imeKlase: lista_izvodjenja_klase
```

Lista izvođenja klase predstavlja niz klasa koje ova klasa nasleđuje sa opisom načina tog nasleđivanja, dakle

```
vrsta_nasledjivanja ime_klase_koja_se_nasledjuje
```

Elementi liste su razdvojeni zarezima. Vrste nasleđivanja mogu biti **private, protected i public.**

Primer 1

```
class A
{...};
class B: public A
{...};
class C: protected B
{...};
class D
{...};
class E: public A, private D
{...};
```

Klasa koja nasleđuje neku drugu klasu naziva se **izvedena** klasa ili **podklasa**. Klasa koju ta klasa nasleđuje je njena **bazna** klasa ili nadklasa. Bazna klasa mora biti definisana u trenutku prevođenja. Ako je A bazna klasa za B, a B bazna klasa za C onda kažemo da je A **posredna bazna klasa** za C. Bazne i izvedene klase formiraju **hijerarhiju klasa**.

Postoje **višestruko** i **jednostruko** nasleđivanje. Na početku ćemo razmatrati samo jednostruko nasleđivanje, dakle *izvedenu klasu definišemo samo uz pomoć jedne bazne klase.* Ako je A posredna bazna klasa za C, i dalje je u pitanju jednostruko nasleđivanje, višestruko nasleđivanje u listi izvođenja ima više od jedne klase.

Ako je potrebno samo deklarisati izvedenu klasu onda se to čini kao i ranije. Dakle:

```
class B;
```

a ne:

```
class B: public A;
```

1.3 Kako izgleda objekat izvedene klase?

Objekat izvedene klase se sastoji iz nestatičkih članova bazne klase i nestatičkih članova izvedene klase. Deo objekta izvedene klase koji sam za sebe predstavlja objekat bazne klase zvaćemo **podobjektom** bazne klase.

Primer 2

```
//Ovo je bazna klasa
class A {
 public:
 int a;
 int MetodA() {...}

private:
 int x,y;
 };

// Klasa B nasledjuje klasu A.
// Nasledjivanje je javno.
// Klasa B je izvedena klasa.
class B: public A {
 public:
 int b;
```

```
 int MetodB() {...}

private: int i, j;
};

int f(B& b) {...}
```

Od čega se sastoji klasa B?

Klasa B ima podatke članove a, b, x, y, i, j.

Klasa B sadrži metode članice MetodA() i MetodB(). Svi podaci i metodi koji su nasleđeni iz klase A čine **podobjekat bazne klase A**.

Kakva je vidljivost podataka i metoda u odnosu na korisnika klase?

Podaci a i b su javni podaci, dok su x, y, i, j privatni. Metode MetodA(), MetodB() su javne metode.

To je zato što je nasleđivanje javno(public). *Kada je nasleđivanje javno to znači da svi nasleđeni članovi zadržavaju isti nivo pristupa.*

Ako je vrsta nasleđivanja zaštićena (**protected**), tada oni članovi koji su bili javni (public) ili zaštićeni (protected) postaju zaštićeni. Privatni članovi bazne klase uopšte ne postoje u izvedenoj klasi, tj. iako se fizički (na nivou implementacije) nasleđuju, logički možemo reći da se ne nasleđuju (ne možemo im pristupiti iz izvedene klase).

Ako je vrsta nasleđivanja privatna (**private**) tada sve što je nasleđeno postaje privatno.

Razmotrimo šta je dostupno metodu MetodA(), šta je dostupno metodu MetodB(), a šta je dostupno spoljašnjoj funkciji f (korisniku klase B)?

MetodA() je javni metod bazne klase A i za njega ne važe nikakva specijalna nova pravila.

MetodB() je javni metod izvedene klase B i on može pristupiti javnom podatku a i metodu MetodA(), kao i svojim privatnim članovima i i j. Međutim, MetodB() ne može da pristupi podacima x i y jer su oni privatni podaci klase A.

Funkcija f može da pristupa javnim podacima a i b, i javnim metodama `MetodA()` i `MetodB()`.

Da je nasleđivanje bilo privatno, tada bi f mogla da pristupa samo podatku b i metodu `MetodB()`, podatak a i `MetodA()` bi u tom slučaju bili privatni pa time nedostupini spoljnoj funkciji `f`.

1.4 Zaštićeni članovi klase

Šta ako želimo da klasa B može da pristupi i svim privatnim podacima klase A?
Ako bi smo privatne podatke klase A proglasili za javne, time bi svako mogao da im pristupi a to nije ono što želimo. Želimo da samo izvedena klasa može da pristupi njenim podacima ali da za sve ostale stanje ostane kao što je do sada i bilo. To se ostvaruje tako što se u klasi A podaci članovi deklarišu kao zaštićeni odnosno **protected** a ne kao privatni.

```
//Ovo je bazna klasa
class A {
public:
 int a;
 int MetodA() {...}

protected:
 int x,y;
};

// Klasa B je izvedena klasa.
class B: public A {
 public:
 int b;
 int MetodB()
 {
 //Sada ovaj metod moze da
 //pristupa podacima x i y
 //klase A jer su oni
 //protected
 ...
 }

private:
 int i, j;
};


int f(A& a) {
//ova funkcija i dalje ne moze
//da pristupa podacima x i y
//za nju je stanje isto kao da su
```

```
//x i y private
... }
```

Da li klasu formirati nasleđivanjem ili umetanjem?
Zavisi od vrste problema, nekada treba koristiti jedno a nekada drugo rešenje. Nasleđivanje se primenjuje ako i samo ako klasa B predstavlja **specijalni slučaj** klase A tj. ako je A **generalizacija** za B.

Koja je razlika u korišćenju sledećih klasa?

```
class A {
 public:
 int a;
 int p() { return _p; }
 int MetodA()
{...}

private:
 int _p;
};

class B {
 public:
 A a;
 int x;
//...
};
```

ili

```
class B : public A {
public: int x;
//...
}
```

Razlika je velika. Pored novih mogućnosti koje koncept nasleđivanja pruža a koje nisu moguće prilikom rada sa umetnutim klasama, jedna od lako uočljivih razlika je u korišćenju objekata iz klase B:

```
B b;
```

Pristup podatku _p iz ove klase je u prvom slučaju:

```
b.a.p() //nije dozvoljeno b.a._p (_p je privatan clan u a)
```

- a u drugom slučaju je:

```
b.p()
```

1.5 Konstruktori i destruktori

U okviru izvršavanja konstruktora izvedene klase najpre se poziva konstruktor bazne klase. Prilikom uništavanja objekta, prvo se poziva destruktor izvedene klase i onda on automatski poziva destruktor bazne klase.

Primer 3

```
#include<iostream>
using namespace std;

class Zivotinja
{
public:
    Zivotinja(char* s, short bg) : ime(s), broj_godina(bg)
        {cout<<"Konstruktor zivotinje"<<endl;}
    ~Zivotinja()
        {cout<<"Destruktor zivotinje"<<endl;}

protected:
  char* ime;
  short broj_godina;
};

class Macka : public Zivotinja
{
public:
    Macka(char* s, short bg, short t):Zivotinja(s,bg),tezina(t)
        {cout<<"Konstruktor macke"<<endl;}
    ~Macka()
        {cout<<"Destruktor macke"<<endl;}
private:
    short tezina;
};
int main()
{
Zivotinja z("zivotinja", 3);
Macka m("maca", 2, 3);
```

```
return 0;
}

Izlaz iz programa:
Konstruktor zivotinje
Konstruktor zivotinje
Konstruktor macke
Destruktor macke
Destruktor zivotinje
Destruktor zivotinje
```

1.6 Skrivanje, predefinisanje i preopterećivanje

Ako metod u izvedenoj klasi ima isto ime kao neki metod iz bazne klase onda metod iz izvedene klase skriva metod iz bazne klase. Ovo važi čak i ako se ne slažu po tipu.

Primer 4 `class A { public: void m(char*) {...} };`

```
class B : public A {
 public:
  int m (int)
  {
//odavde se ne moze pozvati m("abc")osim sa A::m("abc");
... }
};
```

Ukoliko u baznoj i izvedenoj klasi imamo metod koji ima isto ime, broj i tipove argumenata (uključujući i const i tip rezultata), onda kažemo da je izvedena klasa zapravo predefinisala metod iz bazne klase (**overriding**). Potrebno je razlikovati pojam predefinisanja (overrideing) od pojma preopterećivanja (overloading). **Preopterećivanje** označava davanje istog imena većem broju metoda tj. funkčija a **predefinisanje** označava kreiranje metode u izvedenoj klasi sa istim imenom i istim potpisom. **Potpis metode** čine ime, broj i tip argumenata. Potpis ne uključuje povratni tip.

Mogući su neki neočekivani rezultati. Ako klasa A ima metod f nad kojim je izvršeno preopterećivanje i B vrši predefinisanje nad tim metodom, B će sakriti sve metode iz A sa ovim imenom.

Primer 5
`class A`

```
{
public:
    int f()const
    {
    //...
    }
    int f(int x)const
    {
    //preopterecivanje prethodne metode
    //...
    }
protected:
    int i,j;
};
class B : public A
{
public:
    int f()const
    {
    //predefinisanje metode f iz klase A
    //...
    }
};
int main()
{
    A a;
    B b;
    a.f();
    a.f(x);
    b.f();
   // b.f(x); greska, predefinisan metod f
   // bez argumenata je sakrio metod f iz A
    b.A::f(x);//ok
}
```

1.7 Pravila

1. Objekat izvedene klase ima pristup protected članovima samo svog podobjekta svoje bazne klase. Npr.

```
class A {
protected:
    int x;
```

```
//...
};

class B : public A {
public:
    int f(A a){ ...
//odavde se moze pozvati x ali ne moze a.x
}
...
};
```

2. **Prijateljstvo se ne nasledjuje**:
   Ako je A bazna klasa klase B i C je prijateljska (friend) klasa klase A onda C nije prijateljska klasa klase B (osim ako B ne deklarise suprotno). Isto važi i za prijateljske funkcije.

3. **Konstruktori, destruktori i operator dodele se ne nasleđuju:**
   Ako je A bazna klasa klase B i A ima konstruktor sa jednim argumentom onda ga B ne nasleđuje. Ne može se pozvati konstruktor klase B sa jednim argumentom ako nije definisan u klasi B - bez obzira na konstruktore bazne klase. Isto važi za destruktor i operator dodele (funkcijuju operator =).

1.8 Virtuelne funkcije

1.8.1 Statičko vezivanje

Statičko vezivanje je "odlučivanje" koja će metoda biti pozvana u vreme prevođenja. Naime, pretpostavimo da postoje dve metode sa istim imenom u istoj klasi koje se razlikuju po broji i/ili tipu argumenata. Odluka o tome koja će od ove dve metode biti pozvana može se doneti u fazi prevođenja i to tako što se izvrši poređenje tipova argumenata. Ukoliko postoji dvosmislenost onda prevodilac javlja grešku. Takođe, ako ne postoji metod sa datim imenom u izvedenoj klasi onda se on traži u baznoj klasi.

Na osnovu ovog može se steći utisak da je to dovoljno i da se sve može razrešiti statički. Međutim, programski jezik C++ omogućava dodatnu fleksibilnost tj **dinamičko vezivanje**. Dinamičko vezivanje je "odlučivanje" koja će metoda biti pozvana u vreme izvršavanja programa.

C++ omogućuje pokazivačima na baznu klasu da dobiju vrednost pokazivača na objekte izvedenih klasa. Dakle, može se napisati

sledeće:
`A* pok_bazna=new B;`
Kreira se objekat tipa B i vraća se pokazivač na taj objekat koji se dodeljuje pokazivaču na A. Ovaj pokazivač zatim može se koristiti za pozivanje bilo kog metoda iz A. Isto važi i za reference.

Možemo primetiti da je dozvoljeno dodeljivanje objektu bazne klase objekta izvedene klase. U tom slučaju se odbacuje sve ono što je dodato u odnosu na baznu klasu. Obrnuta operacija nije dozvoljena jer deo objekta ostaje neinicijalizovan.

Primer 6 *Ako imamo baznu klasu Životinja i ako iz nje izvedemo klasu Mačka, tada se pokazivaču na tip Životinja može dodeliti adresa nekog objekta klase Mačka.* ***Obrnuto ne važi****, tj pokazivaču na tip Mačka ne može se dodeliti adresa nekog objekta klase Životinja. To je zato što je Mačka istovremeno i Životinja, ali Životinja ne mora biti Mačka (može da bude i na primer Pas).*
Preko pokazivača na Životinju moguće je pristupiti metodama koje se nalaze u klasi Životinja, metode koje su specifočne za klasu Mačka nije moguće pozvati preko ovog pokazivača.

Primer 7 *Želimo da metodi koji vrše predefinisanje u klasi B budu ispravno pozvani umesto odgovarajućih metoda klase A. To se u ovom primeru neće desiti.*

```
#include <iostream>
using namespace std;

class A {
 public:
    int x;

    A(int c) : x(c) {}

    void metodA()
        { cout << "Ovo je metod klase A: " << x << "\n"; }
};

class B : public A {
 public:
    B(int c) : A(c) {}

    void metodA()
        { cout << "Ovo je metod klase B: " << x << "\n"; }
```

```
};

main() {
    A* niz[2];
    niz[0] = new A(1);
    niz[1] = new B(2);

    niz[0]->metodA();
    niz[1]->metodA();

    delete niz[1];
    delete niz[0];
}

Izlaz iz ovog programa:
 Ovo je metod klase A: 1
 Ovo je metod klase A: 2
```

1.8.2 Dinamičko vezivanje

Ukoliko u izvedenim klasama jedne bazne klase imamo metode koje su predefinisale neke metode bazne klase onda bi bilo poželjno da prevodilac prepozna na koju smo izvedenu klasu mislili.

Primer 8 *Ako imamo baznu klasu Životinja i ako iz nje izvedemo klasu Mačka, klasu Pas i klasu Konj, tada, na primer možemo formirati niz pokazivača na klasu Životinja kojima u zavisnosti od situacije možemo dodeliti da pokazuju na različite Mačke, Pse ili Konje. Ako su izvedene klase predefinisale neku metodu f klase Životinja, želimo da pozivom te metode uz pomoć pokazivača na Životinju bude pozvana odgovarajuća predefinisana metoda i to iz klase Mačka ukoliko pokazivač pokazuje na Mačku, iz klase Pas ukoliko pokazivač pokazuje na Psa ili iz klase Konj, ukoliko pokazivač pokazuje na Konja.*

Da bi se pozivi metoda razrešavali dinamički neophodno je da koristimo pokazivač ili referencu na objekat izvedene klase. Tada zapravo možemo da biramo da li da se vezivanje vrši statički ili dinamički. Da bi se vršilo dinamičko vezivanje neophodno je da odgovarajuće metode deklarišemo kao **virtuelne**. To se postiže navođenjem ključne reči *virtual* na početku deklaracije metode.

```
class A {
 public:
```

```
  virtual int VirtMetod();
//...
};

class B : public A {
 public: int
  VirtMetod(); //redefinicija
//...
};
```

Nije neophodno navesti ključnu reč virtual u izvedenoj klasi, ali nije ni greška. Ako imamo:

```
B b;
A *a = &b;
a->VirtMetod(); //?!
```

postavlja se pitanje da li će poslednjim redom biti pozvana metoda klase A ili B? Odgovor je da ako se ne navede ključna reč virtual onda će objekat klase B biti tumačen kao objekat klase A i biće pozvana metoda klase A tj. izvršiće statičko vezivanje. Ukoliko se međutim navede ključna reč virtual, onda će se pozvati metoda iz klase B, jer će u trenutku izvršavanja promenljivoj a biti pridružena adresa objekta klase B, tj izvršiće se dinamičko vezivanje. Ovakav mehanizam (pozivanje metode iz odgovarajuće klase preko pokazivača ili reference na baznu klasu) poznat je kao **virtuelni mehanizam**.

Primer 9 *Virtuelni mehanizam ne funkcioniše za prenos po vrednosti jer se tada izvodi kopiranje samo dela objekta čime se dobija objekat drugog (baznog) tipa.*

```
#include <iostream>
using namespace std;

class A
{
private:
    //  privatni podatak se NE vidi u metodima
    //  klasa naslednica
    int p;

protected:
    //  zasticeni podatak se vidi u metodima
    //  klasa naslednica ali ne van njih
```

```
    int z;

public:
    int x;

    void metodA()
        {
        cout << "A::metodA - " << x << endl;
        }

    void metodX()
        {
        cout << "A::metodX" << endl;
        }

    virtual void metodY()
        {
        cout << "A::metodY" << endl;
        }
};

class B : public A
{
public:
    void metodB()
        {
        cout << "B::metodB - " << x << endl;
        }

    void metodX()
        {
        cout << "B::metodX" << endl;
        }

    void metodY()
        {
        cout << "B::metodY" << endl;
        }
};

//Prenos po vrednosti
void f( A a )
{
```

```
    a.metodA();
    a.metodX();
    a.metodY();
}

//Prenos po referenci
void fr( A& a )
{
    a.metodA();
    a.metodX();
    a.metodY();
}

//Prenos preko pokazivaca
void fp( A* a )
{
    a->metodA();
    a->metodX();
    a->metodY();
}

main()
{
    A a;
    a.x = 5;

    cout << a.x << endl;
    a.metodA();
    a.metodX();

    cout << endl;
    B b;
    b.x = 7;
    cout << b.x << endl;
    b.metodA();
    b.metodB();
    b.metodX();
    b.A::metodX();

    cout << endl;
    f(a);
    f(b);
```

```
    cout << endl;
    fr(a);
    fr(b);

    cout << endl;
    fp(&a);
    fp(&b);

    cout << endl;
    a = b;
    a.metodA();
    a.metodX();

    return 0;
}

/* Izlaz iz programa:

5
A::metodA - 5
A::metodX

7
A::metodA - 7
B::metodB - 7
B::metodX
A::metodX

A::metodA - 5
A::metodX
A::metodY
A::metodA - 7
A::metodX
A::metodY

A::metodA - 5
A::metodX
A::metodY
A::metodA - 7
A::metodX
B::metodY

A::metodA - 5
```

```
A::metodX
A::metodY
A::metodA - 7
A::metodX
B::metodY

A::metodA - 7
A::metodX

  */
```

Napomene:

1. Virtuelna funkcija mora biti nestatička članica klase.
2. Konstruktori i operator new ne mogu biti virtuelni.

1.8.3 Konstruktor i destruktor

1. Ako je bar jedan metod virtuelan onda i destruktor treba da bude virtuelan
2. Konstruktor ne može da bude virtuelan.

Primer 10 *Ilustracija razlike pozivanja virtuelnih i ne virtuelnih metoda.*

```
#include <iostream>
using namespace std;

class A {
 protected:
    int _vrednost;

public:
    A( int n )
        : _vrednost(n)
        {}

    int vrednost() const
        { return _vrednost; }

    virtual void ispis( ostream& ostr ) const
        { ostr << vrednost(); }
};
```

```
// Klasa B ne definise ispis
class B : public A {
 public:
    B( int n )
        : A( n )
        {}

    void promena( int n )
        { _vrednost = n; }
};

// Klasa C ce predefinisati ispis
//Ispis iz C ima uglaste zagrade
class C : public A {
 public:
    C( int n )
        : A(n)
        {}

    void ispis( ostream& ostr ) const
        { ostr << '[' << vrednost() << ']'; }
};

// Funkcija provera kao prvi argument ima
// referencu na baznu klasu
void provera( const A& x, char* ime ) {
    cout << ime << ".vrednost() = " << x.vrednost() << endl;
    cout << ime << ": ";
    x.ispis(cout);
    cout << endl;
}

main() {
    A a(3);
    provera(a,"a");

    B b(3);
    b.promena(6);
    provera(b,"b");

    C c(7);
    provera(c,"c");
```

```
    //poziv  ispisa iz A
    cout << "c: ";
    c.A::ispis(cout);
    cout << endl;

    //Ispis iz C
    cout << "c(2): ";
    c.ispis(cout);
    cout << endl;

    return 0;
}
/*
Izlaz iz programa:
a.vrednost() = 3
a: 3
b.vrednost() = 6
b: 6
c.vrednost() = 7
c: [7]
c: 7
c(2): [7]
*/
```

Primer 11 *Ilustracija redosleda pozivanja destruktora(korišćenjem ne virtuelnog destruktora).*

```
#include <iostream>
using namespace std;

class X {
 public:
    ~X()
        { cout << "Destruktor klase X\n"; }
};

class A {
 public:
    int x;

    A(int c) : x(c) {}
```

```
    ~A()
        { cout << "Destruktor klase A " << x << "\n"; }

    virtual void metod()
        { cout << "Ovo je glavni metod klase A: " << x << "\n"; }
    void metodA()
        { cout << "Ovo je metod klase A: " << x << "\n"; }
};

class B : public A {
 public:
    X q;

    B(int c) : A(c) {}

    ~B()
        { cout << "Destruktor klase B " << x << "\n"; }

    void metod()
        { cout << "Ovo je glavni metod klase B: " << x << "\n"; }
    void metodB()
        { cout << "Ovo je metod klase B: " << x << "\n"; }
};


class C : public A {
 public:
    C(int c) : A(c) {}

    ~C()
        { cout << "Destruktor klase C " << x << "\n"; }
};

main() {
    A* niz[3];
    niz[0] = new A(1);
    niz[1] = new B(2);
    niz[2] = new C(3);

    niz[0]->metod();
    niz[1]->metod();
    niz[2]->metod();
```

```
    delete niz[0];
    delete niz[1];
    delete niz[2];
    return 0;
}
/*
Izlaz iz programa:
Ovo je glavni metod klase A: 1
Ovo je glavni metod klase B: 2
Ovo je glavni metod klase A: 3
Destruktor klase A 1
Destruktor klase A 2
Destruktor klase A 3
*/
```

Primer 12 *Ilustracija redosleda pozivanja destruktora(korišćenjem virtuelnog destruktora).*

```
#include <iostream>
using namespace std;

class X {
 public:
    ~X()
        { cout << "Destruktor klase X\n"; }
};

class A {
 public:
    int x;

    A(int c) : x(c) {}

    virtual ~A()
        { cout << "Destruktor klase A " << x << "\n"; }

    virtual void metod()
        { cout << "Ovo je glavni metod klase A: " << x << "\n"; }
    void metodA()
        { cout << "Ovo je metod klase A: " << x << "\n"; }
};

class B : public A {
```

```
 public:
    X q;

    B(int c) : A(c) {}

    ~B()
        { cout << "Destruktor klase B " << x << "\n"; }

    void metod()
        { cout << "Ovo je glavni metod klase B: " << x << "\n"; }
    void metodB()
        { cout << "Ovo je metod klase B: " << x << "\n"; }
};


class C : public A {
 public:
    C(int c) : A(c) {}

    ~C()
        { cout << "Destruktor klase C " << x << "\n"; }
};

main() {
    A* niz[3];
    niz[0] = new A(1);
    niz[1] = new B(2);
    niz[2] = new C(3);

    niz[0]->metod();
    niz[1]->metod();
    niz[2]->metod();

    delete niz[0];
    delete niz[1];
    delete niz[2];
    return 0;
}
/*
Izlaz iz programa:
Ovo je glavni metod klase A: 1
Ovo je glavni metod klase B: 2
Ovo je glavni metod klase A: 3
```

```
Destruktor klase A 1
Destruktor klase B 2
Destruktor klase X
Destruktor klase A 2
Destruktor klase C 3
Destruktor klase A 3
*/
```

Može se uočiti da će se korišćenjem ne virtuelnog destruktora osloboditi memorija koju je zauzimao samo podobjekat koji odgovara baznoj klasi, objekta izvedene klase a ostatak će biti trajno izgubljen u memoriji. Zato je neophodno da destruktor bude virtuelan.

1.9 Apstraktne klase

Postoje situacije u kojima virtuelna funkcija u baznoj klasi ne može da uradi nešto što bi imalo smisla. Tada se sve zapravo odradi u izvedenim klasama. Takva virtuelna funkcija zove se **čisto virtuelna funkcija** i deklariše se tako što se iza naslova doda = 0. Klasa koja sadrži bar jednu čisto virtuelnu funkciju zove se **apstraktna klasa**. Ukoliko se pokuša sa kreiranjem objekta apstraktne bazne klase, prevodilac će prijaviti grešku. Moguće je međutim koristiti pokazivač na apstraktnu klasu.

Primer 13 *Ilustracija apstraktne klase.*

```
#include <iostream>
using namespace std;

class Zivotinja {
    char* _ime;

public:
    Zivotinja( char* s )
        : _ime(s)
        {}

    virtual ~Zivotinja()
        {}

    char* ime() const
        { return _ime; }
```

```
    virtual int brojNogu() const = 0;
    virtual bool leti() const = 0;
    virtual char* kaziZdravo() const = 0;
};

// I ovo je apstraktna klasa jer nije predefinisala
// metod kaziZdravo()!
class Sisar : public Zivotinja {
public:
    Sisar( char* s )
        : Zivotinja(s)
        {}

    int brojNogu() const
        { return 4; }

    bool leti() const
        { return false; }
};

class Pas : public Sisar {
 public:
    Pas( char* s )
        : Sisar(s)
        {}

    char* kaziZdravo() const
        { return "AvAvvv"; }
};

class Slon : public Sisar {
 public:
    Slon( char* s )
        : Sisar(s)
        {}

    char* kaziZdravo() const
        { return "Juhuuuu"; }
};

class Delfin : public Sisar {
 public:
```

```
    Delfin( char* s )
        : Sisar(s)
        {}

    int brojNogu() const
        { return 0; }

    char* kaziZdravo() const
        { return "Zviiizz"; }
};

class Ptica : public Zivotinja {
 public:
    Ptica( char* s )
        : Zivotinja(s)
        {}

    int brojNogu() const
        { return 2; }

    bool leti() const
        { return true; }

    char* kaziZdravo() const
        { return "Ciju-ci"; }
};

class Kokoska : public Ptica {
 public:
    Kokoska( char* s )
        : Ptica(s)
        {}

    bool leti() const
        { return false; }

    char* kaziZdravo() const
        { return "Kokoda"; }
};

void provera( const Zivotinja& x ) {
    cout << x.ime() << endl;
    cout << (x.leti() ? "leti" : "ne leti") << endl;
```

```
    cout << "ima " << x.brojNogu() << " nogu(e)" << endl;
    cout << "kaze: " << x.kaziZdravo() << endl;
    cout << endl;
}

main() {
    Zivotinja* zivotinje[] = {
        new Pas("Bili"),
        new Slon("Cira"),
        new Delfin("Joca"),
        new Ptica("Kiki"),
        new Kokoska("Koka")
        };

    for( int i=0; i<sizeof(zivotinje)/sizeof(Zivotinja*); i++ )
        provera( *zivotinje[i] );

    for( int i=0; i<sizeof(zivotinje)/sizeof(Zivotinja*); i++ )
        delete zivotinje[i];

    return 0;
}
/*
Izlaz iz programa:

Bili ne leti ima 4 nogu(e) kaze: AvAvvv

Cira ne leti ima 4 nogu(e) kaze: Juhuuuu

Joca ne leti ima 0 nogu(e) kaze: Zviiizz

Kiki leti ima 2 nogu(e) kaze: Ciju-ci

Koka ne leti ima 2 nogu(e) kaze: Kokoda
*/
```

Sadržaj