Objektno programiranje - nadaljevanje

Center za heterogeno procesiranje

Primer uporabe destruktorja:

```
int main()
{
    Kompleksno* x;
        x = new Kompleksno(1.0,1.0);    // klic konstruktorja
    za ustvarjanje
//dinamičnega objekta
    cout << "x: ";
    x->izpisi();
    delete x;                                   // klic destruktorja
    return 0;
```

Klic metod in dostop do podatkov objektov, ki so dinamično generirani

Za dostop do metod ali podatkov v objektih, ki so dinamično generirani (generirani so z uporabo kazalcev) uporabljamo operator ->.

V našem primeru smo uporabili ta operator v **x ->izpisi().** Enakovreden zapis je tudi: **(*x).izpisi()**;

Konstantni objekti

To so objekti, ki jim ne moremo spremeniti vrednosti podatkov.

Primer:

const Kompleksno i(0.0,1.0);

Konstantne metode

V konstantnem objektu lahko kličemo samo konstantne metode, to so tiste metode, ki ne spremenijo podatkov v objektu. Primer takšne metode je metoda izpisi:

```
void Kompleksno::izpisi() const
  { cout << '(' << realni << ", "
  << imaginarni << ')';
```

Konstantni podatki

Poleg konstantnih metod lahko definiramo v razredu tudi konstantne podatke.

Primer:

```
class Kompleksno {
public:
    Kompleksno();                          // privzeti konstruktor
    Kompleksno(double, double);            // konstruktor

private:
    double realni;       // realni del
    double imaginarni;   // imaginarni del
     const double pi;
};
```

Konstruktor kličemo na naslednji način:

```
Kompleksno::Kompleksno(double r=0.0, double i=0.0)
: realni(r), imaginarni(i), pi(3.14)
{
}
```

Kazalec this

Pri vsakem klicu metode se vanjo kot argument prenese še naslov objekta nad katerim ta metoda deluje. Naslov objekta, ki je klical metodo lahko uporablja tudi programer.

Primer:

```
class Preizkus {
public:
   Preizkus(int = 0);    // privzet konstruktor
   void izpisi() const;
private:
   int x;
};
```

Kazalec this

```
Preizkus::Preizkus(int a) { x = a; } // konstruktor
 void Preizkus::izpisi() const
{
   cout << "         x = " << x << endl
        << "  this->x = " << this->x << endl
        << "(*this).x = " << (*this).x << endl;
}
main()
{
   Preizkus a(12);
    a.izpisi();
    return 0;
}
```

Program izpiše:

```
   x = 12
  this->x = 12
(*this).x = 12
```

Prekrivanje operatorjev (operator overloading)

Določene operatorje želimo prekriti. Npr. operacije + nimamo definirane za kompleksna števila.

Prekrivanje operatorjev

```
class Kompleksno {
public:
   Kompleksno(double = 0.0, double = 0.0);
   // prekrivanje operatorja za setevanje
     Kompleksno operator+(const Kompleksno &) const;
   // prekrivanje operatorja za odstevanje
     Kompleksno operator-(const Kompleksno &) const;
     // prekrivanje operatorja za prirejanje
     Kompleksno &operator=(const Kompleksno &);
     void izpisi() const;
private:
   double realni;        // realni del
   double imaginarni;    // imaginarni del
};
```

Prekrivanje operatorjev

```
// konstruktor
Kompleksno::Kompleksno(double r, double i)
{
   realni = r;
   imaginarni = i;
}
// Prekrivni operator sestevanja
Kompleksno Kompleksno::operator+(const Kompleksno
   &operand2) const
{
   Kompleksno vsota;
   vsota.realni = realni + operand2.realni;
   vsota.imaginarni =  imaginarni +
operand2.imaginarni;
   return vsota;
}
```

Prekrivanje operatorjev

```
// Prekrivni operator odstevanja
Kompleksno Kompleksno::operator-(const Kompleksno &operand2)
   const
{
   Kompleksno razlika;
   razlika.realni = realni - operand2.realni;
   razlika.imaginarni = imaginarni –
operand2.imaginarni;
   return razlika;
}

// Prekrivni operator =
Kompleksno& Kompleksno::operator=(const Kompleksno &desno)
{
   realni = desno.realni;
   imaginarni = desno.imaginarni;
   return *this;   // omogoca veckratno prirejanje
}
```

Prekrivanje operatorjev

```
// Prikazi objekt Kompleksno v
  obliki: (a, b)
void Kompleksno::izpisi() const
  { cout << '(' << realni << ", "
   << imaginarni << ')';
 }
```

Prekrivanje operatorjev – glavna funkcija

```
main()
{
   Kompleksno x, y(4.3, 8.2), z(3.3, 1.1);
   cout << "x: ";
   x.izpisi();
   cout << endl << "y: ";
   y.izpisi();
   cout << endl << "z: ";
   z.izpisi();
   x = y + z;
   cout << endl << endl << "x = y + z:" << endl;
   x.izpisi();
   cout << " = ";
   y.izpisi();
   cout << " + ";
   z.izpisi();
```

Prekrivanje operatorjev glavna funkcija - nadaljevanje

```
x = y - z;
   cout << endl << endl << "x = y - z:" << endl;
   x.izpisi();
   cout << " = ";
   y.izpisi();
   cout << " - ";
   z.izpisi();
   cout << endl;
   return 0;
}
```

Program izpiše:

```
x: (0, 0)
y: (4.3, 8.2)
z: (3.3, 1.1)

x = y + z:
(7.6, 9.3) = (4.3, 8.2) + (3.3,
  1.1)

x = y - z:
(1, 7.1) = (4.3, 8.2) - (3.3, 1.1)
```

Vmesnik razreda

Vmesnik razreda predstavljajo vsi javni elementi razreda. Uporabnik nekega razreda mora poznati le njegov vmesnik, podrobnosti o implementaciji pa zanj niso pomembne. Tako je programska koda neodvisna od implementacije.

Prijateljske funkcije

Do privatnih metod ali podatkov razreda lahko dostopajo samo prijateteljske funkcije razreda (friend funkcije), čeprav niso člani razreda.

Primer – prijateljske funkcije

```
class Stevec {
   friend void doloci_x(Stevec &, int);
 // opredelitev friend funkcije
public:
   Stevec() { x = 0; }        // konstruktor
   void izpisi() const { cout << x << endl; }
 // izhod
private:
   int x;  // podatkovni element razreda
};

void doloci_x(Stevec &c, int vred)
{
   c.x = vred;  // dovoljeno: doloci_x je friend
// funkcija Stevca
}
```

Primer – prijateljske funkcije – glavna funkcija

```
main()
{
   Stevec objekt;

   cout << "objekt.x po uvedbi: ";
   objekt.izpisi();
   cout << "objekt.x po klicu friend funkcije
             doloci_x: ";
   doloci_x(objekt, 8);  // dolocitev x s friend funkcijo
   objekt.izpisi();

   return 0;
}
```

Program izpiše:

```
objekt.x po uvedbi: 0
objekt.x po klicu friend funkcije doloci_x: 8
```

DEDOVANJE

Z uporabo **dedovanja** iz obstoječih razredov tvorimo nove razrede.

Takšno tvorjenje razredov imenujemo **izpeljava** in razrede, ki tako nastanejo **izpeljani razredi** ali **podrazredi**. Izpeljani razred ima torej svoj **nadrazred**, tj. razred, iz katerega je izpeljan. Dedovanje je lahko **enkratno** ali **večkratno** – v tem primeru je razred izpeljan iz dveh ali več nadrazredov.

DEDOVANJE

Izpeljani razred **podeduje** vse elemente svojega nadrazreda. K tem elementom pa lahko doda še svoje elemente (podatke in metode) inponovno definira metode iz svojega nadrazreda. Zato lahko izpeljane razrede razumemo kot posebne primere svojih nadrazredov. Tako izpeljani razredi predstavljajo **specializacijo**, nadrazredi pa **posplošitev**.

Ker izpeljani razred podeduje vse elemente svojega nadrazreda, je objekt izpeljanega razreda hkrati tudi objekt nadrazreda. Obratno pa ne velja, kajti objektu nadrazreda manjkajo

Primer izpeljave razreda student iz razreda oseba:

Razred student predstavlja sprecializacijo razreda oseba.

K podatkom o določeni osebi bomo torej dodali podatke,ki določajo, da je ta oseba pravzaprav študent. Zaradi enostavnosti smo izbrali podatek "smer", ki predstavlja smer študija posameznega študenta.

```
#include<iostream.h>

enum spol {moski, zenski};

class oseba
{
    public:
        oseba(char im[], char p[], spol s, char dat_r[]);
        oseba() {};
        void Vpisi();
        void Izpisi();
    private:
        char ime[15];
        char priimek[15];
        spol spol_o;
        char datum_r[10];
};
```

```
class student : public oseba
{
    public:
        void Vpisi();
        void Izpisi();
    private:
        char smer[20];
};
```

Beseda **public**, ki je zapisana pri izpeljavi nam pove, da so javni ("public") elementi nadrazreda (oseba) javni tudi v izpeljanem razredu (student).

```
oseba :: oseba(char im[], char p[], spol s, char dat_r[])
{

    for (int i=0; i<15; i++)
      ime[i] = im[i];

    for (i=0; i<15; i++)
      priimek[i] = p[i];

    spol_o = s;

    for (i=0; i<10; i++)
      datum_r[i] = dat_r[i];
}
```

```
void oseba :: Vpisi()
{
    int sp;
    cout << endl;
    cout << " Vpisi ime: ";
    cin  >> ime;
    cout << " Vpisi priimek: ";
    cin  >> priimek;
    cout << " Vpisi spol 1 - zenski, 2 - moski: ";
    cin  >> sp;

    if (sp == 1)
        spol_o = zenski;
    else
        spol_o = moski;

    cout << " Vpisi datum rojstva: ";
    cin  >> datum_r;
}
```

```
void student :: Vpisi()
{
   cout << " Vpisi smer: ";
   cin  >> smer;
}
```

```
void oseba :: Izpisi()
{
    cout << endl << " Ime: " << ime;
    cout << endl << " Priimek: " << priimek;
    if (spol_o == zenski)
        cout << endl << " Spol: Zenski";
    else
        cout << endl << " Spol: Moski";

    cout << endl << " Datum rojstva: " << datum_r << endl;
}
void student :: Izpisi()
{
    cout << " Smer: " << smer << endl;
}
```

```
int main()
{
    student luka;
    luka.oseba::Vpisi();
    luka.Vpisi();
    luka.oseba::Izpisi();
    luka.Izpisi();
    return 0;
}
```

Zaščiteni elementi

Poleg javnih in privatnih elementov lahko v razredih definiramo tudi zaščitene elemente z določilom **protected**.

Zaščiteni elementi so vmesna stopnja med javnimi in privatnimi elementi. Do **javnih elementov** nekega razreda lahko z objektom tega razreda dostopa kdorkoli v programu.

Do **privatnih elementov** nekega razreda lahko dostopajo samo ta razred in njegovi prijatelji.

Do **zaščitenih elementov** nekega razreda lahko dostopajo enako kot do privatnih elementov ta razred in njegovi prijatelji, poleg tega pa še vsi izpeljani razredi in prijatelji teh razredov.

Tipi izpeljav

Izpeljavo, v kateri uporabimo ključno besedo public, kot npr. v primeru:

class student : public oseba

imenujemo **javna izpeljava**. Poleg te ima jezik C++ še **zaščiteno** in **privatno** izpeljavo. Ta dva tipa izpeljav določimo s ključnima besedama protected in private:

class student : protected oseba

in

class student : private oseba

Tipi izpeljav

Zaščitena in privatna izpeljava v izpeljanih razredih omejita dostop do zaščitenih in javnih elementov nadrazreda. Dostop do elementov nadrazreda je predstavljen v naslednji tabeli:

Tipi izpeljav

tip elementa	**Tip izpeljave**	**Tip izpeljave**	**Tip izpeljave**
	public	***protected***	***private***
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Tipi izpeljav

Podatki v tabeli povedo, kakšen je dostop do elementa nadrazreda v izpeljanih razredih, če imamo podan tip elementa in tip izpeljave. Pri **zaščiteni izpeljavi** se javni elementi nadrazreda obravnavajo kot zaščiteni elementi, pri **privatni izpeljavi** pa se javni in zaščiteni elementi obravnavajo kot privatni elementi.

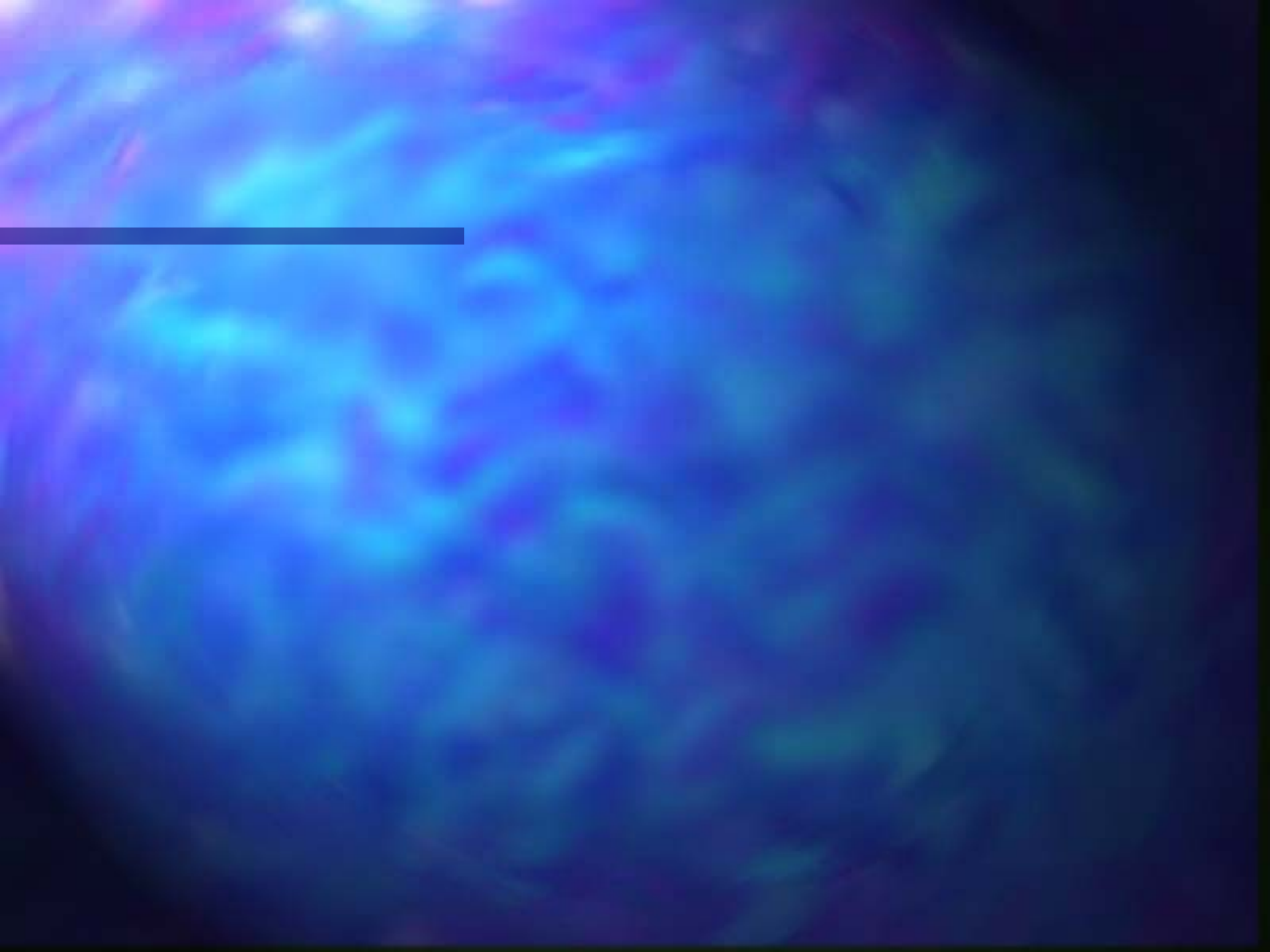
POLIMORFIZEM

Dedovanje in **polimorfizem** sta dve najpomembnejši lastnosti objektnega programiranja. Prvo smo že spoznali, druga pa označuje princip, da lahko različni objekti razumejo isto sporočilo in se nanj odzovejo vsak na svoj način. Obe lastnosti sta osnova objektnega programiranja in omogočata hitrejši razvoj in lažje vzdrževanje programske kode.

Polimorfizem - primer

Imejmo poleg razreda `student`, iz razreda `oseba` izpeljana še razreda `profesor` in `asistent`. Oba naj imata po en dodaten element: profesor naj ima predmet, ki ga poučuje, asistent pa predmet, pri katerem ima vaje. Vsak od njiju ima svojo inačico metode izpiši, ki izpiše podatke tega objekta.

Oglejmo si primer, ko želimo vnašati in izpisovati več objektov tipa študent, asistent ali profesor. Za shranjevanje bomo uporabili polje kazalcev. Ker ne vemo kakšne elemente bomo gradili (ta podatek bo znan šele v izvajanju), bomo shranjevali kazalce na skupni



```
#include  "student.h"
#include  "asistent.h"
#include  "profesor.h"
#include  <iostream.h>

const int max_oseb = 4;

int main()
{
   oseba* ptr_polje_oseb[max_oseb];     //polje za shranjevanje oseb
   oseba* ptr_začasna;

   cout << "Vnesi " << max_oseb << " osebe !" << endl;

   char tip_osebe[];
   char ime[vel_ime];
   char naslov[vel_naslov];
   char smer[vel_smer];
   char vaje[vel_vaje];
```

```
char predmet[vel_predmet];


   for (int i = 0;  i  < max_oseb;  i++)
   {
    cout <<  endl << endl << "Vnesi tip osebe (S, A, P):  ";
    cin >> tip_osebe;
    cout  << "Vnesi ime           : ";
    cin << ime;
    cout  << "Vnesi naslov        : ";
    cin >> naslov;

    if  (tip_osebe  ==  'S')
      {
         cout << "Smer studenta          : ";
         cin >> smer;
         ptr_zacasna  =  new student (ime, naslov, smer);
      }
```

```
else  if  (tip_osebe  ==  'A')
      {
         cout << "Predmet asistenta      : ";
         cin >> vaje;
         ptr_zacasna  =  new asistent (ime, naslov, vaje);
      }
     else  if  (tip_osebe  ==  'P')
      {
         cout << "Predmet profesorja     : ";
         cin >> vaje;
         ptr_zacasna  =  new profesor (ime, naslov, predmet);
      }
      else
        {
         cout  << "Napacen tip osebe!" << endl;
         cout  << "Ustvarilo bom objekt razreda oseba!";
         ptr_zacasna  =  new oseba (ime, naslov);
        }
       ptr_polje_oseb[i]  = ptr_zacasna;
  }
```

```
  //izpis vnesenih oseb
  for (i = 0;  i < max_oseb;  i++)
  {
    ptr_polje_oseb[i] ->izpisi();
    cout << endl;
  }


  //brisanje polja kazalcev
  for (i = 0;  i < max_oseb;  i++)
     delete ptr_polje_oseb[i];

   return 0;
}
```

Razložimo program! Najprej v zanki preberemo tip osebe ter ime in naslov osebe. Glede na tip osebe preberemo še smer, vaje ali predmet in ustvarimo objekt ustreznega razreda s pomočjo operatorja `new`. Ta nam vrne kazalec na objekt (`student*` ali `asistent*` oz. `profesor*`). Ta kazalec shranimo v spremenljivko `ptr_zacasna`, ki pa je tipa `oseba*`. Kako lahko to storimo? To je mogoče zato, ker je vsak študent (oz. asistent ali profesor) tudi oseba, zato lahko napravimo pretvorbo kazalca: npr. `student*` v `oseba*`. Pretvorba se napravi implicitno pri operatorju `new`. Pretvorbo bi lahko zahtevali tudi eksplicitno z naslednjim zapisom:

```
ptr_zacasna = (oseba*)new student(ime,naslov,smer);
```

Kadar kot tip osebe vnesemo nepravilno vrednost, ustvarimo objekt tipa oseba. Ko smo vnesli vse osebe, jih še izpišemo in nato zbrišemo (sproščanje dinamično

Prikaz delovanja programa:

```
Vnesi 4 osebe!

Vnesi tip osebe (S, A, P):   S
Vnesi ime        :  Janez
Vnesi naslov        :  Mladinska
Smer studenta       :  Anglescina

Vnesi tip osebe (S, A, P):   A
Vnesi ime        :  Metka
Vnesi naslov        :  Slovenska
Predmet asistenta      :  Nemscina

Vnesi tip osebe (S, A, P):   P
Vnesi ime        :  France
Vnesi naslov        :  Ljubljanska
Predmet profesorja     : Računalnistvo
```

```
Vnesi tip osebe (S, A, P):   k
Vnesi ime        :  Marija
Vnesi naslov         :  Betnavska
Napacen tip osebe!
Ustvaril bom objekt razreda oseba!


Ime osebe    :  Janez
Naslov           :  Mladinska

Ime osebe    :  Metka
Naslov           :  Slovenska

Ime osebe    :  France
Naslov           :  Ljubljanska

Ime osebe    :  Marija
Naslov       : Betnavska
```

Polimorfna redefinicija

Če si ogledamo izpis programa opazimo, da so se za vsak vnešeni objekt izpisali samo podatki, ki so zapisani v razredu `oseba` in ne vsi podatki študenta, asistenta in profesorja. Zakaj?

Ko smo objekte shranjevali, smo kazalec pretvorili v tip `oseba*`. Ko kličemo metodo `izpisi`, zato kličemo metodo `oseba::izpisi`. Vidimo, da navadna redefinicija metode v tem primeru ne zadošča. Čeprav imamo samo kazalce na razred `oseba`, želimo, da vsak objekt ohrani sebi lastno obnašanje ob klicu metode `izpisi`. Uporabiti moramo **polimorfno redefinicijo**.

Polimorfna redefinicija

Polimorfno redefinicijo omogočimo z definicijo **virtualne** metode. Metodo `izpisi` moramo v razredu `oseba` definirati kot virtualno, kar storimo na naslednji način:

```
virtual void izpisi() const;
```

Ta definicija pove, da naj izpeljani objekt kliče svojo funkcijo `izpisi`, čeprav bi bil klic izvršen preko kazalca na nadrazred (`oseba*`). Ob takšni definiciji metode `izpisi` v razredu `oseba`, bo prejšnji primer izpisal naslednje:

```
Vnesi 4 osebe!

Vnesi tip osebe (S, A, P):   S
Vnesi ime          :  Janez
Vnesi naslov         :  Mladinska
Smer studenta        :  Anglescina

Vnesi tip osebe (S, A, P):   A
Vnesi ime          :  Metka
Vnesi naslov         :  Slovenska
Predmet asistenta       :  Nemscina

Vnesi tip osebe (S, A, P):   P
Vnesi ime          :  France
Vnesi naslov         :  Ljubljanska
Predmet profesorja   : Računalnistvo
```

```
Vnesi tip osebe (S, A, P):   k
Vnesi ime          :  Marija
Vnesi naslov         :  Betnavska
Napacen tip osebe!
Ustvaril bom objekt razreda oseba!

Ime osebe    :  Janez
Naslov          :  Mladinska
Smer            : Angleščina

Ime osebe    :  Metka
Naslov          :  Slovenska
Vaje pri        : Nemščina

Ime osebe    :  France
Naslov          :  Ljubljanska
Predmet         : Računalništvo

Ime osebe    :  Marija
Naslov       : Betnavska
```

Polimorfna redefinicija

Problem se pojavi tudi pri brisanju dinamično alociranih objektov. Pri klicu operatorja `delete` se bo klical destruktor za razred `oseba` namesto destruktor ustreznega objekta. Zato moramo tudi destruktor definirati kot virtualno metodo:

```
virtual ~oseba();
```

Abstraktni razredi

Včasih imamo hierarhijo razredov, v kateri imamo nadrazred, za katerega vemo, da ne bomo uporabljali objektov tega razreda. Tak razred hočemo imeti v hierarhiji samo zaradi definicije polimorfnega obnašanja nekaterih funkcij. Take razrede imenujemo **abstraktni razredi**. Primeri takih abstraktnih razredov so lahko na primer `oblika`, `telo`, `lik`, ...

Razred napravimo abstrakten tako, da definiramo vsaj eno izmed njegovih metod kot **čisto virtualno** (*pure virtual function*). Funkcija je čisto virtualna, kadar za definicijo vsebuje še inicializacijo = 0.

Čisto virualne funkcije

```
virtual void izracunaj_ploscino() const  =  0;
```

Funkcijo `izracunaj_ploscino` lahko napravimo čisto virtualno za razred `lik`. Funkcija omogoča polimorfno definicijo funkcije za vsakega izmed izpeljanih razredov, hkrati pa definira razred `lik` kot abstraktni razred, ki ga ne moremo uporabiti za tvorjenje objektov. To tudi pomeni, da moramo funkcijo `izracunaj_ploscino` redefinirati v vsakem razredu, ki ga izpeljemo iz razreda `lik` – kar pa je tudi smiselno, saj v razredu `lik` nimamo podatkov za izračun ploščine.

Čisto virtualne funkcije moramo redefinirati v vseh izpeljanih razredih, navadne virtualne pa ne. Kadar izpeljani razred nima redefinirane navadne virtualne funkcije, se bo klicala funkcija nadrazreda.

Večkratno dedovanje

Razred je lahko izpeljan tudi iz več kot enega razreda. V tem primeru govorimo o večkratnem dedovanju. Definirajmo razred `student_asistent`, ki naj predstavlja osebe, ki so hkrati študentje in asistentje.

```
#include  "student.h"
#include  "asistent.h"

class  student_asistent  :  public student,  public
    asistent
{
   public:
    student_asistent (char i[],  char n[],  char s[],
    char v[]);
    void  izpisi()  const;
};
```

V konstruktorju razreda `student_asistent` sedaj podamo ime, naslov, poleg tega pa še smer in predmet. Oglejmo si definicijo konstruktorjev in metode `izpisi()`.

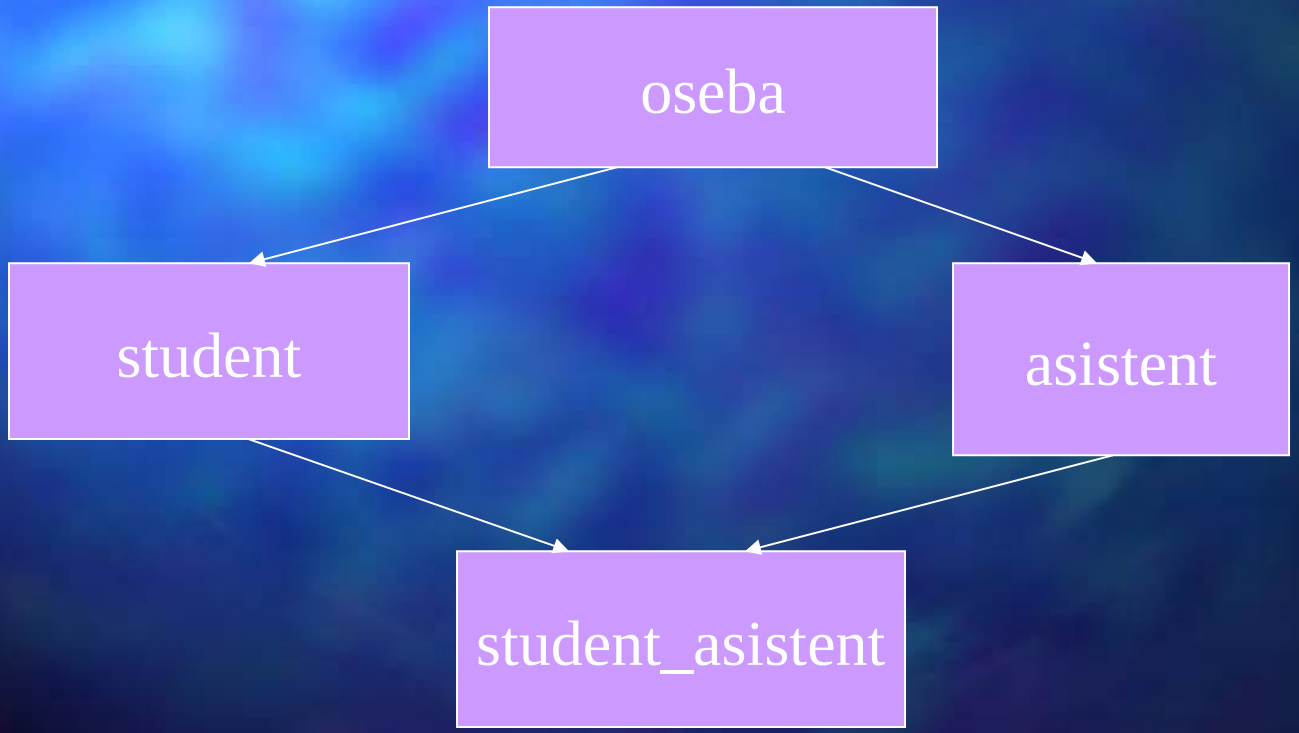
```
#include  <iostream.h>

student_asistent :: student_asistent (char i,  char n,  char s,
    char v)
    : student (i, n, s),  asistent ("---", "+++" , v)  {}

void student_asistent :: izpisi()  const
{
    student::izpisi();
    cout << endl;
    asistent::izpisi();
}
```

Večkratno dedovanje

Z večkratnim dedovanjem dobi razred več primerkov nadrazredov:



Večkratno dedovanje

V našem primeru je razred `student_asistent` izpeljan iz razredov `student` in `asistent`, ta dva pa sta izpeljana iz razreda `oseba`. Objekt tipa `student_asistent` v tem primeru vsebuje dva objekta tipa `oseba`, ki sta med seboj neodvisna in lahko vsebujeta različne podatke. Različne podatke v obeh objektih tipa oseba lahko dosežemo tako, da konstruktorjema nadrazredov `student` in `asistent` podamo različne argumente:

: student (i, n, s), asistent ("---", "+++" , v)

```
#include  <iostream.h>
#include  "student_asistent.h"
#include  "profesor.h"

int main()
{
    student  sosolec("Janez", "Ljubljanska 12", "Računalništvo");
    cout  << "Student sosolec" << endl;
    sosolec.izpisi();
    cout << endl << endl;

    student_asistent  sosed("Peter", "Mariborska 23",
     "Programska oprema",  "Programiranje ");
    cout  << "Student-asistent sosed"  << endl;
    sosed.izpisi();
    cout  << endl;

    return 0;
}
```

Program izpiše:

```
Student sosolec
Ime osebe   : Janez
Naslov      : Ljubljanska 12
Smer        : Računalništvo

Student-asistent sosed
Ime osebe   : Peter
Naslov      : Mariborska 23
Smer        : Programska oprema
Ime osebe   : ---
Naslov      : +++
Vaje        : Programiranje
```

Večkratno dedovanje

Če v razredu `student_asistent` ne definiramo metode za izpis `izpisi()`, je klic `sosed.izpisi()` napačen, saj prevajalnik ne ve, katero od metod `izpisi()` naj kliče; tisto, ki je v razredu `student` ali tisto, ki je v razredu `asistent`. V tem primeru moramo sami navesti nadrazred, katerega metoda za izpis se naj kliče: `student::izpisi()` ali `asistent::izpisi()`.

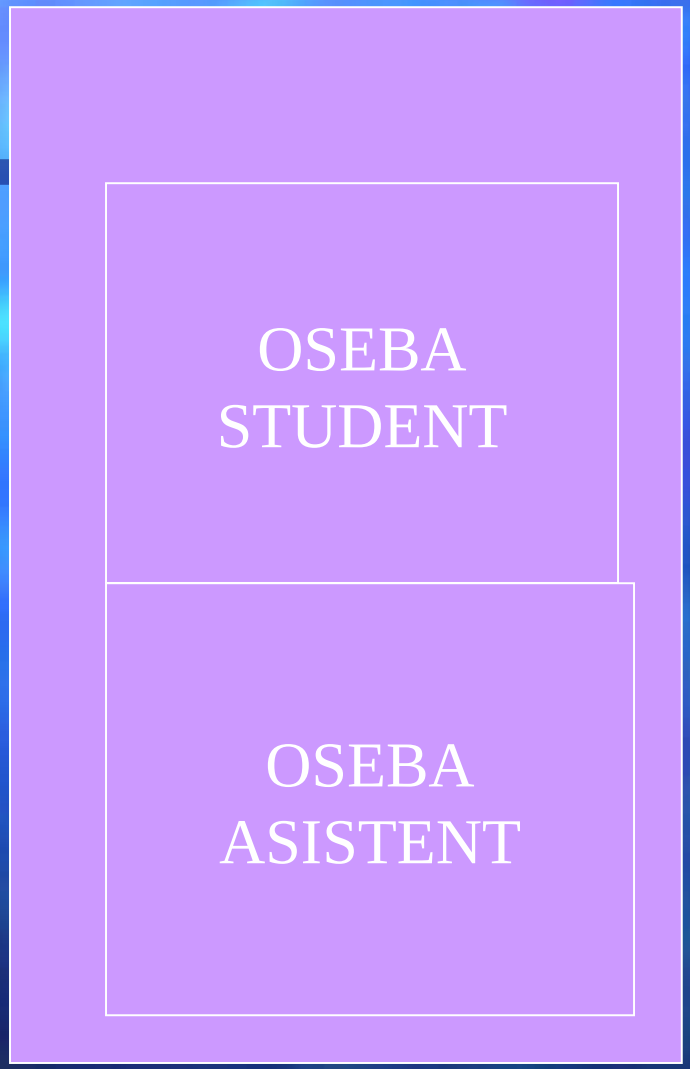
Virtualni nadrazredi

Ugotovili smo, da se v razredih pri večkratnem dedovanju podatki podvajajo. Včasih je to potrebno, v večini primerov pa se želimo nepotrebnemu podvajanju podatkov izogniti. V ta namen lahko definiramo **virtualne nadrazrede** (*virtual base classes*).

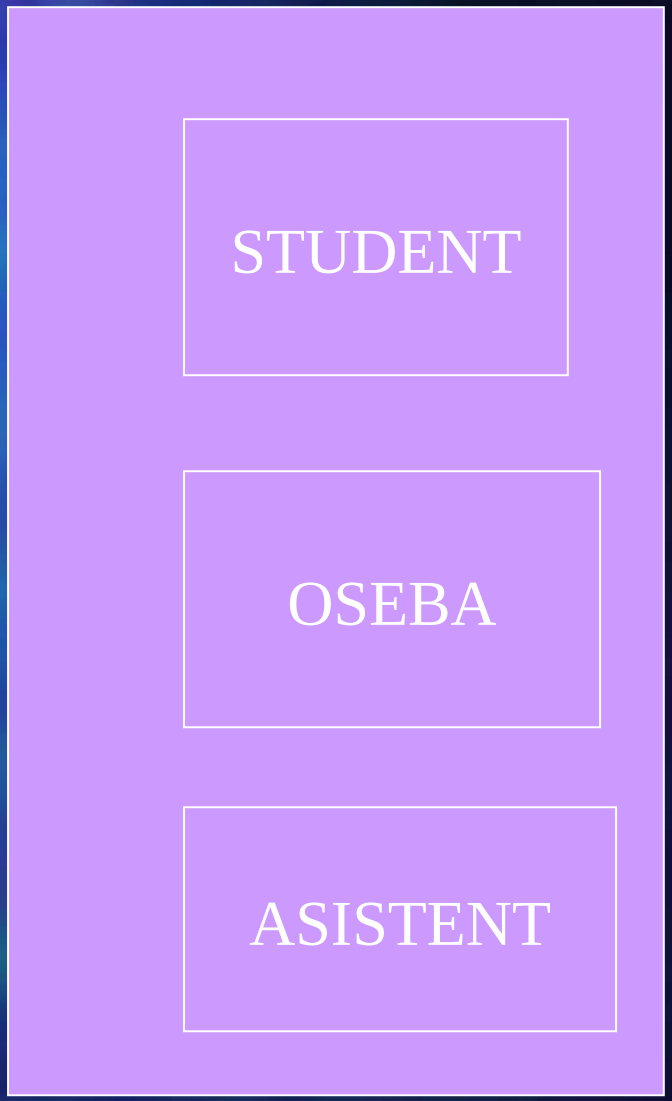
Kot virtualni nadrazred definiramo tisti razred, za katerega predvidevamo, da se bo uporabljal kot nadrazred za večkratno dedovanje. V našem primeru definiramo kot virtualni nadrazred, razred oseba. Zato zapišemo izpeljavo tazredov student, asistent in profesor na naslednji način:

class student :: public virtual oseba

Takšno izpeljavo imenujemo tudi virtualna izpeljava. Poglejmo grafično predstavitev objekta razreda student_asistent pri navadni in virtualni izpeljavi:



Navadna izpeljava



Virtualna izpeljava

VIRTUALNA IZPELJAVA

Spremeniti pa je potrebno tudi konstruktor razreda student_asistent:

```
student_asistent ::
student_asistent (char i,  char n,
char s,  char v)
: oseba(i, n),  student (i, n, s),
asistent ("---", "+++" , v)  {}
```