Določilo protected

1

- poznamo tri načine vrste določil za dostop do razrednih komponent
  - public ... javne
  - private ... zasebne
  - protected ... zaščitene
- določilo `protected` pride do izraza pri dedovanju
  - razredi, ki dedujejo zaščiteno spremenljivko ali funkcijo, lahko do le-te direktno dostopajo
- izven razreda in izpeljanih razredov se določilo `protected` obnaša enako kot `private`

Pregled dostopa do razrednih komponent

2

DOSTOP:	znotraj razreda	v izpeljanih razredih	izven razredov - preko objektov
public	DA	DA	DA
protected	DA	DA	NE
private	DA	NE	NE

- spomnimo se, da lahko dostop vedno omogočimo preko javnih metod, v katerih imamo nadzor nad spremembami

Primer: razred CDelavec

3

- zapisali bomo razred CDelavec, ki bo predstavljal delavca
- zapisali bomo tudi razred CManager, ki je vrste delavec, s to razliko, da lahko ima fiksno plačo
- razreda sta zelo podobna, in ker velja: manager je vrste delavec, uporabimo dedovanje
  - vsebuje vse podatke in funkcionalnosti razreda CDelavec
  - dodamo nove spremenljivke in metode
  - ponovno definiramo tiste metode, ki se razlikujejo od zapisanih v nadrazredu

Primer: razred CManager

4

- delavec ima naslednje
  - lastnosti
    - ime ... string
    - tarifa ... koliko je plačan na enoto (npr. na uro)
  - obnašanja
    - konstruktorji
    - vrni/spremeni funkcije za lastnosti
    - izračun plače
- manager ima naslednje
  - lastnosti
    - ime ... string
    - tarifa ... koliko je plačan na enoto (npr. na uro, mesec)
    - fiksnaPlaca (bool)
  - obnašanja
    - konstruktorji
    - vrni/spremeni funkcije za lastnosti
    - izračun plače

manager je vrste delavec => uporabimo dedovanje

Primer: razred CDelavec (.h)

5

```
class CDelavec {
  protected:
    string ime;
    double tarifa;
  public:
    CDelavec(string aI, double aT);
    virtual ~CDelavec() { }
    string vrniIme() const { return ime; }
    double vrniTarifo() const { return tarifa; }
    void spremeniIme(string aI);
    void spremeniTarifo(double aT);
    double placa(int enote) const;
};
```

Primer: razred CDelavec (.cpp)

6

```
CDelavec::CDelavec(string aI, double aT) {
  spremeniIme(aI);
  spremeniTarifo(aT);
}

void CDelavec::spremeniIme(string aI) { ime = aI; }

void CDelavec::spremeniTarifo(double aT) {
  tarifa = 0;
  if (aT>=0)
    tarifa = aT;
}

double CDelavec::placa(int enote) const {
  return enote*tarifa;
}
```

Primer: razred CManager (.h)

7

```
class CManager : public CDelavec {
  protected:
    bool fiksnaPlaca;
  public:
    CManager(string aI, double aT, bool aF);
    virtual ~CManager() { }
    bool imaFiksnoPlaco() const { return fiksnaPlaca; }
    double placa(int enote) const;
};
```

plača se za managerja izračuna drugače, kot za običajnega delavca, zato to funkcijo ponovno definiramo, kljub temu, da je že definirana v nadrazredu

Primer: razred CManager (.cpp)

8

```
CManager::CManager(string aI, double aT, bool aF) :
CDelavec(aI,aT) {
  fiksnaPlaca = aF;
}

double CManager::placa(int enote) const {
  if (fiksnaPlaca)
    return tarifa;
  else
    return enote*tarifa;
}
```

do spremenljivke *tarifa* lahko direktno dostopamo, saj je v baznem razredu definirana kot zaščitena

Primer: uporaba prejšnjih dveh razredov

9

```
#include "CManager.h"
...

int main() {
  CDelavec del("Janez Novak", 4.5);
  CManager mng("John Doe", 3000, true);

  cout << "Delavec " << del.vrniIme() << " je plačan ";
  cout << del.vrniTarifo() << " na uro in dobi na mesec ";
  cout << del.placa(160) << "EUR." << endl;
  cout << "Manager " << mng.vrniIme() << " dobi na mesec ";
  cout << mng.placa(160) << "EUR." << endl;
  return 0;
}
```

Naloga za vajo

10

- Napišite razred CNadzornik, ki predstavlja nadzornika. Nadzornik je odgovoren za delavce v določenem oddelku in mora:
  - imeti podatek o imenu oddelka (string)
  - imeti funkciji vrni/spremeni oddelek
  - ima **vedno** fiksno plačo, ne glede na število opravljenih ur
  - imeti konstruktor, ki inicializira vse razredne spremenljivke
- iz katerega razreda boste izpeljali razred CNadzornik?

Dedovanje in polimorfizem

11

- primer polimorfizma smo že srečali pri prekrivanju operatorjev
  - polimorfizem – omogoča, da se objekti na isto sporočilo (funkcijo) odzovejo na sebi edinstven način
- objekt izpeljanega razreda je hkrati objekt baznega razreda
  - kjerkoli pričakujemo objekt baznega razreda, lahko ustvarimo objekt izpeljanega razreda
- kako v takem primeru vemo, katera funkcija se izvede – tista iz nadrazreda ali tista iz izpeljanega razreda?
  - odgovor – to povedo virtualne funkcije

Virtualne funkcije - primer

12

```
class CSesalec {
  public:
    CSesalec() {}
    ~CSesalec() {}
    void premakniSe();
    virtual void oglasiSe();
};

void CSesalec::premakniSe() {
    cout << "Premik sesalca za en korak." << endl;
}

void CSesalec::oglasiSe() {
    cout << "Sesalec se oglaša." << endl;
}
```

Virtualne funkcije - primer

13

```
class CPes : public CSesalec {
  public:
    CPes() {}
    ~CPes() {}
    void premakniSe();
    virtual void oglasiSe();
    void dajTaco();
};
```

```
void CPes::premakniSe() {
  cout << "Pes se zapodi." << endl;
}

void CPes::oglasiSe() {
  cout << "Hov hov." << endl;
}

void CPes::dajTaco() {
  cout << "Taca." << endl;
}
```

Virtualne funkcije - primer

14

uporaba nad statično ustvarjenim objektom:

```
CSesalec s = CPes();
s.premakniSe();
s.oglasiSe();
// s.dajTaco();
```

izpis na ekran:

```
Premik sesalca za en korak.
Sesalec se oglaša.
```

objekt tipa CPes je tudi tipa CSesalec, saj je iz tega razreda izpeljan, zato ga lahko priredimo kot objekt nadrazreda

s je tipa CSesalec, v katerem ni definirane funkcije dajTaco(), zato je ne moremo klicati

izvedejo se funkcije tipa, ki ga uporabimo v deklaraciji, v našem primeru razreda CSesalec

Virtualne funkcije - primer

15

uporaba nad dinamično ustvarjenim objektom:

```
CSesalec *s = new CPes();
s->premakniSe();
s->oglasiSe();
// s->dajTaco();
```

izpis na ekran:

```
Premik sesalca za en korak.
Hov hov.
```

objekt tipa CPes je tudi tipa CSesalec, saj je iz tega razreda izpeljan, zato lahko ustvarimo objekt tipa CPes

funkcija premakniSe() **ni** definirana kot **virtualna**, zato se izvede funkcija razreda iz deklaracije

s je kazalec na objekt tipa CSesalec, v katerem ni definirane funkcije dajTaco(), zato je ne moremo klicati

funkcija oglasiSe() **je** definirana kot **virtualna**, zato se izvede funkcija glede na dejanski tip objekta, ki smo ga ustvarili

Virtualne funkcije

16

- če je funkcija deklarirana kot virtualna, to pove:
  - da bo zelo verjetno ta razred bazni razred nekega drugega razreda
  - v izpeljanem razredu bo ta funkcija ponovno definirana, saj se bodo objekti tega tipa pri klicu te funkcije odzvali na svojstven način
  - kadar polimorfno obravnavamo objekte (objekt podrazreda obravnavamo kot objekt baznega razreda), se izvede funkcija iz dejanskega tipa objekta (tista iz podrazreda)
- virtualne funkcije delujejo na opisan način le nad kazalci na objekte in referencami na objekte
  - nad dinamično ustvarjenimi objekti, ne pa tudi nad statičnimi

Kaj pa obratno?

17

- IMELI SMO TOLE
  - objekt tipa podrazreda lahko obravnavamo kot objekt tipa nadrazreda
  - če vseeno želimo, da se izvedejo funkcije podrazreda, lahko to dosežemo s pomočjo dinamično ustvarjenih objektov in virtualnih funkcij
- KAJ PA OBRATNO?
  - objekt tipa nadrazreda želimo obravnavati kot objekt tipa podrazreda
  - to je mogoče s pretvarjanjem tipov

Pretvarjanje tipov

18

- poznamo več vrst pretvarjanja tipov
  - implicitno pretvarjanje
    - se izvede samodejno za združljive tipe

    PRIMER:
    ```
    int x = 3;
    double y = x; // int se pretvori v double
    ```
  - eksplicitno pretvarjanje
    - sami eksplicitno zapišemo, kot kateri tip želimo objekt obravnavati

    PRIMER:
    ```
    int x = 3;
    double y = (double)x / 2;
    ```

Eksplicitno pretvarjanje tipov

19

- eksplicitno pretvarjanje kazalcev na objekte

PRIMER:

```
CSesalec *x = new CPes();
// x->dajTaco();
((CPes *)x)->dajTaco();
```

x je kazalec na CSesalec, v katerem ni definirane funkcije dajTaco(), zato je ne moremo klicati

ta del kode pravi: *obravnavaj x kot kazalec na CPes*, tam je definirana funkcija dajTaco(), zato jo lahko kličemo

- TEŽAVA: C++ dopušča, da kazalec na nek tip eksplicitno pretvorimo v kazalec na poljuben drugi tip. Kadar to naredimo za nezdružljive tipe, se lahko program zruši, ali pa dobimo nepredvidljiv rezultat.

Kako se izognemo eksplicitnemu pretvarjanju?

20

- C++ pozna t.i. varna pretvarjanja s pomočjo naslednjih operatorjev:
  - dynamic_cast<tip> (objekt)
  - static_cast<tip> (objekt)
  - reinterpert_cast<tip> (objekt)
  - const_cast<tip> (objekt)
- pogledali si bomo dynamic_cast<tip>(objekt)
  - uporabimo ga lahko samo nad kazalci in referencami na objekte
  - rezultat operatorja je kazalec na pretvorjen tip, v primeru, ko se lahko dani objekt v celoti pretvori v nov tip
  - v nasprotnem primeru je rezultat kazalec NULL

Primer uporabe dynamic_cast<>

21

```
CSesalec* x = new CSesalec();
CPes* y = dynamic_cast<CPes*>(x);
if (y!=NULL)
     y->dajTaco ();
else
     cout << "Napaka";
```

s tem rečemo: *poskusi pretvoriti objekt x, ki je tipa CSesalec, v objekt tipa CPes*

v tem primeru izpiše "Napaka", saj je objekt tipa CSesalec nima vseh lastnosti in obnašanj kot objekti tipa CPes, zato ga ni mogoče v celoti pretvoriti