DRAUGIŠKOS FUNKCIJOS IR KLASĖS

Objektiniame programavime prieiga prie klasės duomenų rekomenduojama įgyvendinti klasės metodais. Tokia prieigos disciplina tarnauja klasių (objektų) inkapsuliacijos koncepcijos įgyvendinimui. Tačiau C++ kalboje realizuotas draugiškumo mechanizmas leidžia apeiti tokia griežtą prieigos discipliną. Draugiškomis klasei gali būti apibrėžiamos funkcijos arba kitos klasės.

Draugiškos funkcijos, nebūdamos klasės elementais (nariais), gauna prieigą net prie uždarų (private, protected) klasės (ar kelių klasių) duomenų. Draugiška funkcija turi būti apibrėžta toje klasėje, kuriai ji yra draugiška. Draugiška funkcija skelbiama matomoje (*public*) klasės dalyje prieš jos vardą nurodant bazinį žodį **friend**.

Kadangi draugiška funkcija nepriklauso klasės metodams, todėl iškviečiant ją nereikia nurodyti objekto vardo ar nuorodos į objektą prieš funkcijos vardą. Prieigą prie uždarų klasės duomenų draugiška funkcija gauna per tos klasės objektą, todėl toks objektas turi būti apibrėžtas funkcijoje arba perduotas jai parametrų sąraše.

Pavyzdys 3.1. Draugiška funkcija.

```
class AnyClass
{ int n, d;
   public:
   AnyClass (int x, int y) : n(x), d(y) { }
   friend bool draugas (AnyClass) ;
};
bool draugas (AnyClass obj)
  {
        if (! obj.n % obj.d )
        return false; else return true;
  }
main ()
  { AnyClass obj (12, 3);
    if (draugas(obj))
       cout<< "12 dalinasi is 3"<< endl;
    else
       cout<< "12 nesidalina is 3"<< endl;
    return 0; }
```

Funkcija gali būti vienos klasės metodu, o kitoje klasėje ji gali būti apibrėžta kaip draugiška funkcija. Funkcija taip pat gali būti draugiška

keliom klasėm. Tačiau, draugiška funkcija nėra paveldima, t.y. ji nėra draugiška išvestinėm klasėm.

DRAUGIŠKOS KLASĖS

C++ leidžia apibrėžti ne tik draugiškas funkcijas, bet ir klases. Draugiškai klasei suteikiama pilna prieiga prie duotosios klasės duomenų. Aprašant klasę ***B*** draugišką klasei ***A,*** reikia, kad klasės ***A*** apibrėžtyje būtų nurodyta, kad klasė ***B*** yra jai draugiška. Draugiška klasė skelbiama klasės matomoje (*public)* dalyje prieš jos vardą nurodant bazinį žodį **friend**.

Pavyzdys 3.2. Draugiška klasė.

```
class AA
{
      int n, d;
    public:
        AA (int x, int y) : n(x), d(y) { }
        void  increase() { n++; d = d*10; }
        friend class BB ;
};
class BB
{
      AA objA(3,5);
    public:
        void show()
        { cout<< “Pradinės reikšmės”<<objA.n<<“ “<<objA.d << endl;
          objA.increase();
         cout<< “Pakeistos reikšmės”<<objA.n<<“ “<<objA.d << endl; }
};
void main()
{
BB objB;
objB.show();
}
```

Draugiškos klasės pasižymi tokiomis savybėmis:

- draugiškumo santykis nėra abipusis, t.y. jei klasė ***B*** yra draugiška klasei *A*, tai nereiškia, kad klasė ***A*** draugiška klasei ***B***;
- draugiškos klasės nėra paveldimos, t.y., jei klasė ***B*** yra draugiška klasei ***A***, tai klasės, kurios yra gautos, kaip ***B*** išvestinės, nėra draugiškos klasei ***A***;

- draugiškumas taip pat nepaveldimas, t.y., jei ***B*** yra draugiška klasei ***A***, tai klasėse kurios yra gautos, kaip ***A*** išvestinės, klasė ***B*** nėra draugiška.

Tačiau dvi klasės gali pasiskelbti kad jos yra draugai. Tokie skelbimai turėtų atrodyti taip:

```
class B;      //Nepilnas klasės apibrėžimas
 class A
{
   friend class B;
};

class B
{
      friend class A;
};
```

KLASIŲ HIERACHIJOS KŪRIMO MECHANIZMAI

PAVELDĖJIMAS

C++ klasės gali paveldėti kitos klasės ar net kelių klasių duomenis ir metodus. Taip atsiranda galimybė sudaryti klasių hierarhiją ir ateityje panaudoti klasę kelėta kartų, šiek tiek ją modifikavus.

Klasė ***A***, kurią paveldi klasė ***B***, vadinama **bazinė klase** (arba protėviu). Klasė ***B***, kuri savo duomenis ir metodus papildo kitos klasės (***A***) duomenimis ir metodais vadinama **išvestine klase** (arba palikuonimi). Protėvio duomenys ir metodai pagal nutylėjimą paveldimi kaip *private*, t.y. juos gali naudoti tik palikuonio metodai. Jeigu norima tiesiogiai kreiptis iš išorės į protėvio metodus, reikia nurodyti prieigos specifikaciją *public*.

Išvestinės klasės apibrėžimas:

class Palikuonis : <prieigos specifikaciją> Bazinė_klasė_1,
<prieigos specifikaciją> Bazinė_klasė_2,
...
<prieigos specifikaciją> Bazinė_klasė_N
{ ……. };

Prieigos specifikacijos paskirtis – nustatyti išvestinės klasės paveldimumo lygį, t.y. kokius metodus ir duomenis gali paveldėti klasė. Tuo tarpu bazinė klasė neturi jokiu teisių į duomenis ir funkcijas, apibrėžtus išvestinėje klasėje:

Bazinės klasės prieigos sritys	Pasiekimas iš bazinės klasės	Pasiekimas iš išvestinės klasės	Pasiekimas iš išorinių funkcijų
public	taip	taip	taip
protected	taip	taip	ne
private	taip	ne	ne

Nepaveldimi tokie klasės (ir kiti) elementai:

- konstruktorius;
- kopijos konstruktorius;
- destruktorius;
- priskyrimo operatorius, kurį apibrėžia pats programuotojas;
- draugiškos klasės;

Paveldėjimo prieiga:

Bazinės klasės prieigos sritys	Pasiekimas bazinėje klasėje	Pasiekimas išvestinėje klasėje
public	public protected private	public protected nepasiekiami
protected	public protected private	protected protected nepasiekiami
private	public protected private	private private nepasiekiami

Paveldėjimo prieigos specifikacija nustato tokį lygį, iki kurio sumažinama prieiga išvestinėje klasėje (t.y. nustato išvestinės klasės elementų prieigą prie bazinės klasės elementų). Pvz., jei nustatyta paveldėjimo prieiga *private*, tai bazinės klasės atviri (*public*) elementai tampa uždarais *(private)* elementais išvestinėje klasėje. Tačiau tokius elementus, esant reikalui, galima padaryti atvirais skelbiant juos išvestinės klasės *public* dalyje (pvz. 3.3).

Bet kuriuo paveldėjimo atveju išvestinėje klasėje prieinami tik atviri arba apsaugotieji (*protected*) elementai bazinės klasės, uždari elementai lieka uždarais nepriklausomai nuo nurodytos paveldėjimo prieigos specifikacijos.

Pavyzdys 3.3. Paveldėjimo prieiga.

class Bazine

```
{
    int x,y;
   public:
     int GetX() {return x;}
     int GetY() {return y;}
};
class Isvestine : private Bazine
{
    public:
    Bazine::GetX();
    void showX()
        { cout<< GetX()<< endl;}
};
main()
{   Isvestine A;
    A.showX();
    return 0;
}
```

Išvestinė klasė turi apibrėžti savo konstruktorių, nes bazinės klasės konstruktorius nėra paveldimas. Išvestinės klasės konstruktorius turi priskirti pradines reikšmės tiek bazinės klasės duomenims, tiek saviems. Kadangi savi elementai gali naudoti bazinės klasės elementus, pastarieji turi būti inicializuoti ankščiau nei išvestinės klasės elementai. Jei turime klasę, kuri paveldi keletą bazinių klasių, tuomet svarbus ir bazinių konstruktorių užrašymo eiliškumas.

Konstruktoriaus apibrėžimo forma naudoja inicializacijos sąrašą, kuris susideda iš bazinės klasės konstruktoriaus:

<išvest_kl_konstr>(<parametrai>) : <bazinės_kl_konstr>(<parametrai>) {konstruktoriaus tekstas}

Išvestinės klasės destruktorius turi būti iškviečiamas ankščiau nei bazinės klasės, tik kai jis baigs savo darbą, kompiliatorius iškvies bazinės klasės destruktorių.

Pavyzdys 3.4. Konstruktorius ir destruktorius.

```
#include <iostream.h>
class Bazine
{
  public:
        Bazine()
```

```
        {
        cout<<“Sveiki, vykdomas bazinės klasės konstruktorius \n”;
        }
    ~Bazine()
        {
        cout<<“Vykdomas bazinės klasės destruktorius, viso gero \n”;
        }
};
class Isvestine : public Bazine
{
  public:
    Isvestine()
        {
        cout<<“Sveiki, vykdomas išvestinės klasės konstruktorius \n”;
        }
    ~Isvestine()
        {
        cout<<“Vykdomas išvestinės klasės destruktorius, iki \n”;
        }
};
main()
{
    Isvestine objektas;
    return 0;
 }
```

Paprastai, paveldėdama bazinės klasės elementus, išvestinė klasė dar papildo juos savais. Taip pat galima perrašyti (pakeisti) (angl. *override*) bazinės klasės elementus, pvz., perrašant bazinės klasės metodą, išvestinėje klasėje reikia pateikti jo prototipą ir apibrėžti iš naujo.

Pavyzdys 3.5. Bazinės klasės metodų perrašymas išvestinėje klasėje.

```
#include <iostream.h>
class Taskas
{
    protected:
        int x, y;
   public:
      Taskas(int a, int b) { x=a, y=b; }
      Taskas() { x=0, y=0; }                    //Konstruktorius pagal nutylėjimą
       int Gauti_x() { return x; }
       int Gauti_y() { return y; }
       void Nustatyti_x(int a) {x=a;}
       void Nustatyti_y(int b) {y=b;}

};
```

```
class TaskoIsvedimas : public Taskas
{
    public:
      TaskoIsvedimas(int a, int b) : Taskas (a,b) {}
       int Gauti_x() { return ++x; }
       int Gauti_y() { return ++y; }
       void Rodyti_x() {cout<< Gauti_x()<< ‘ ‘;}
       void Rodyti_y() {cout<< Gauti_y()<< ‘ ‘;}
};

main ()
{
    TaskoIsvedimas* ptr;
    ptr = new TaskoIsvedimas(1,2);          //Dinaminio objekto sukūrimas
    ptr -> Rodyti_x();
    ptr -> Rodyti_y();
    cout<< “\n”;
    delete ptr;                             //Objekto naikinimas
    return 0;
 }
```

C++ leidžia taikyti daugkartinį (daugybinį) paveldėjimą, kai išvestinė klasė paveldi daugiau nei iš vienos bazinės klasės. Kaip ir paprasto (viengubo) paveldėjimo atveju bazinių klasių konstruktoriai yra iškviečiami ankščiau nei išvestinių klasių konstruktoriai. Vienintelis būdas perduoti jiems parametrus – naudoti elementų inicializacijos sąrašą. Šiame sąraše bazinių klasių konstruktoriai išdėstomi tokia tvarka, kokia jie bus iškviečiami (t.y. tokia tvarka, kokia jie paveldimi).

Pavyzdys 3.6. Daugkartinis paveldėjimas: konstruktorių iškvietimas.

```
#include <iostream.h>
class Bazine_1
{
  public:
        Bazine_1()
        {
        cout<<"Sveiki, vykdomas klasės Bazinė_1 konstruktorius \n";
        }
~Bazine_1()
        {
        cout<<"Vykdomas klasės Bazinė_1 destruktorius, viso gero \n";
        }
```

```
} ;
class Bazine_2
{
  public:
        Bazine_2()
        {
        cout<<"Sveiki, vykdomas klasės Bazinė_2 konstruktorius \n";
        }
~Bazine_2()
        {
        cout<<"Vykdomas klasės Bazinė_2 destruktorius, viso gero \n";
        }
};
class Isvestine : public Bazine_1, public Bazine_2
{
  public:
        Isvestine()
        {
        cout<<"Sveiki, vykdomas išvestinės klasės konstruktorius \n";
        }
~Isvestine()
        {
        cout<<"Vykdomas išvestinės klasės destruktorius, iki \n";
        }  ;
};
main()
{
        Isvestine objektas;
        return 0;
        getchar();
}
```

VIRTUALINĖS BAZINĖS KLASĖS

Kai taikomas daugkartinis paveldėjimas galima situacija, kai išvestinė klasė paveldi du arba daugiau tos pačios bazinės klasės egzempliorių. Tokiu atveju atsiranda nevienareikšmiškumas, kai reikia kreiptis į klasės narius.

Pavyzdys 3.7. Daugkartinis paveldėjimas: nevienareikšmiškumas.

```
class Pr_Bazine
{
        int x;
   public:
```

```
        int Gauti_x() {return x;}
        void Nustatyti_x(int a) {x=a;}
        double var;
};
class Bazine_1: public Pr_Bazine
{ …. };
class Bazine_2: public Pr_Bazine
{ …. };
class Isvestine: public Bazine_1, public Bazine_2
{ …. };
main ()
{
  Isvestine ob;
  ob.var = 5.0
  ob.Nustatyti_x(0);
  int z = ob.Gauti_x();
  return 0;
}
```

Kompiliuojant tokia programa kompiliatorius nustatys nevienareikšmiškumą, kai yra kreipimąsi į elementus

```
  ob.var = 5.0
  ob.Nustatyti_x(0);
  int z = ob.Gauti_x();
```

Išvengti nevienareikšmiškumo galima užduodant matomumo praplėtimo operaciją:

```
  ob. Bazine_1::var = 5.0;
  ob.Bazine_1:: Nustatyti_x(0);
```

Tačiau tokiu atveju bazinė klasė **Pr_Bazine** bus įtraukta į klasės **Išvestine** kodą du kartus. Tam, kad būtų išvengta kodo dubliavimo naudojamos virtualinės bazinės klasės.

Pavyzdys 3.8. Virtualinė bazinė klasė.

```
class Pr_Bazine
{
        int x;
   public:
        int Gauti_x() {return x;}
        void Nustatyti_x(int a) {x=a;}
        double var;
```

```
};
class Bazine_1: virtual public Pr_Bazine
{ …. };
class Bazine_2: virtual public Pr_Bazine
{ …. };
class Isvestine: public Bazine_1, public Bazine_2
{ …. };
main ()
{
  Isvestine ob;
  ob.var = 5.0
  ob.Nustatyti_x(0);
  int z = ob.Gauti_x();
  return 0;
}
```

Esant virtualioms paveldimoms klasėms konstruktoriai iškviečiami tokia tvarka:

1. Virtualių bazinių klasių konstruktoriai
2. Nevirtualių bazinių klasių konstruktoriai.

Kai yra keletas virtualių bazinių klasių, jų konstruktoriai iškviečiami tokia eile, kaip surašyti paveldimumo eilėje, jei virtuali klasė yra nevirtualios klasės išvestinė, tuomet pirmas vykdomas nevirtualios bazinės klasės konstruktorius. Destruktorių iškvietimo tvarka –atvirkštinė konstruktorių iškvietimui.

FUNKCIJŲ PERKROVIMAS

Kaip jau minėta perkrovimas – tai vienas iš polimorfizmo atvejų. C++ kalboje naudojamas funkcijų ir operacijų (operatorių) perkrovimas. Funkcijų perkrovimas – kai eilė funkcijų turi tą patį vardą bet skirtingą skaičių argumentų arba jų tipą. Tokios funkcijos vadinamos perkrautomis (angl. *overloaded*).

Perkrovimas naudingas tuomet, kai siekiama supaprastinti programą t.y. funkcijoms, atliekančioms panašius veiksmus su skirtingais argumentų sąrašais, suteikti tokius pat pavadinimus. Kuri iš perkrautų funkcijų bus iškviesta, sprendžia kompiliatorius.

Pavyzdys 3.9. Funcijos *abs* perkrovimas.

```
# include <iostream.h>
int abs (int a);
float abs (float b);
double abs (double c);
void main ()
{  int x = 255; float y = -1.2f; double  z = 155.0;
   cout << abs (x)<< endl;
   cout << abs (y)<< endl;
   cout << abs (z)<< endl;
}

int abs (int x)
{
        return x < 0 ? -x:x;
}

float abs (float y);
{
        return y < 0 ? -y:y;
}

double abs (double z);
{
        return z < 0 ? -z:z;
}
```

Funciją *abs* galima būtų apskaičiuoti ir pasinaudojus bibliotekinėmis funkcijomis, tačiau, tokiu atveju, reikės skirtingų funkcijų, pvz., *abs(), fabs(), labs()* ir pan.