C++ programavimo kalba

Draugiškos funkcijos ir draugiškos klasės, paveldėjimas

(5 paskaita)

Draugiškos funkcijos

C++ galima apeiti vieną iš pagrindinių OOP savybių t.y. duomenų inkapsuliaciją panaudojant draugiškas funkcijas ir draugiškas klases.

Paprastai uždari klasės duomenys, pasiekiami per tos pačios klasės atvirus (public) metodus. Tačiau klasės duomenis galima pasiekti ir per išorinę draugišką funkciją.

Ta pati funkcija gali būti draugiška keletui klasių, todėl ji gali pasiekti keletos klasių duomenis.

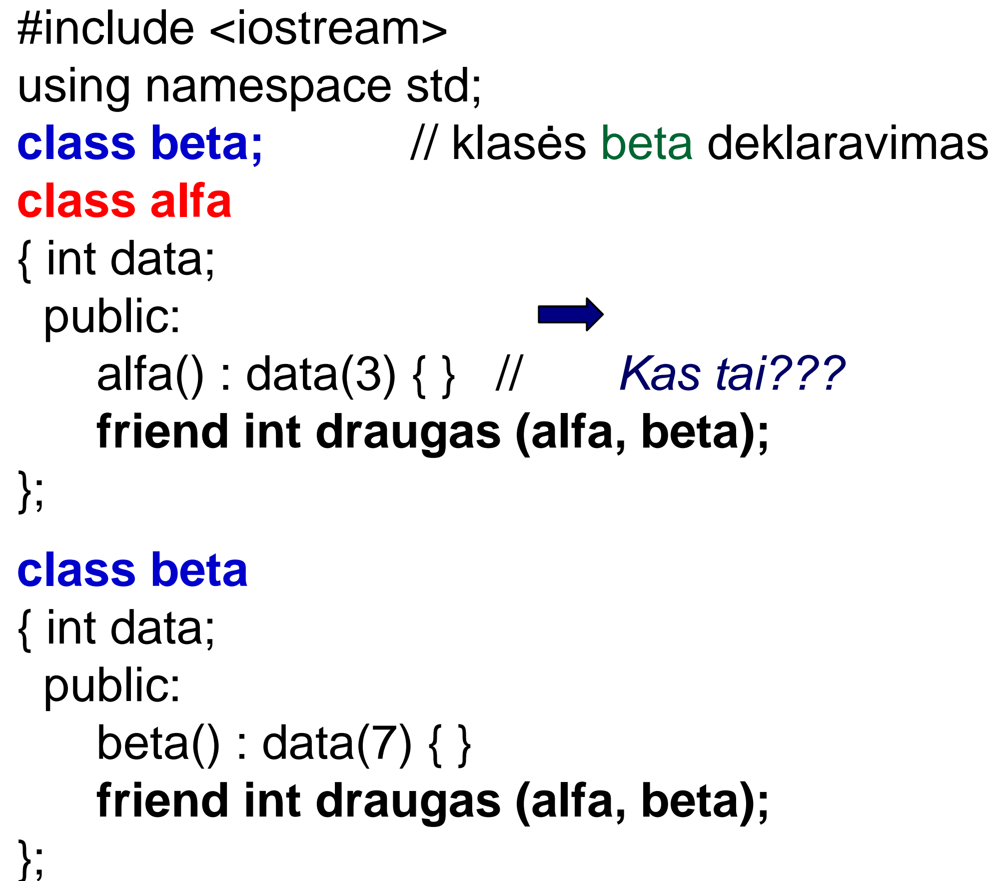
Draugiška funkcija turi būti apibrėžta klasėje, kuriai ji draugiška panaudojant atributą friend.

Draugiška funkcija skelbiama klasės dalyje *public*.

Draugiškos funkcijos pavyzdys

```
#include <iostream>
using namespace std;
class gama
{ int x, y;
  public:
      gama(int a, int b)
      { x = a;  y = b; }
      friend int draugas (gama);      // draugiška funkcija
 };
int draugas (gama AB)
{ return (AB.x - AB.y);
}

main ()
   gama obj(3,5);
   cout<< draugas(obj)<< endl;       // iškviečiama draugiška funkcija
   return 0; }
```

Draugiškos funkcijos pavyzdys



```
#include <iostream>
using namespace std;
class beta;           // klasės beta deklaravimas
class alfa
{ int data;
  public:
     alfa() : data(3) { }   //       Kas tai???
     friend int draugas (alfa, beta);
};
class beta
{ int data;
  public:
     beta() : data(7) { }
     friend int draugas (alfa, beta);
};
```

```
int draugas (alfa a, beta b)
{ return (a.data + b.data);
}

main ()
   alfa aa; beta bb;
   cout<< draugas(aa, bb)<< endl;
   return 0; }
```

Draugiškos funkcijos

Draugiškos funkcijos nepriklauso klasės metodams, todėl iškviečiant draugišką funkciją, nereikia prieš ją nurodyti objekto vardo ar nuorodos į objektą. Uždarus klasės duomenis draugiška funkcija pasiekia per klasės objektą, kuris **turi būti apibrėžtas funkcijoje arba perduotas per funkcijos argumentų sąrašą**.

Draugiška funkcija nepaveldima.

```
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 true; else return false; }
```

```
main ()
{ AnyClass obj (12, 3);
  if (draugas(obj))
     cout<< “12 dalinasi is 3”<< endl;
  else
     cout<< “12 nesidalina is 3”<< endl;
  return 0; }
```

Draugiškos funkcijos

Kartais funkcija gali būti vienos klasės metodu, o kitoje klasėje ji gali būti apibrėžta kaip draugiška funkcija.

```
class beta;                                  // nepilnas klasės apibrėžimas. Jis būtinas
class alfa
{   int data1, data2, ;
    public:  alfa() : data1(3), data2(4) { }
        int draugas (beta bb)
        { return (bb.b1 + bb.b2); }
};
class beta
{ int b1, b2;
    public:   beta() : b1(7) b2(8) { }
    friend int alfa::draugas (beta bb)          // draugiška funkcija iš klasės alfa
};
void main ()
   { alfa aa; beta bb;
   cout<< aa.draugas(bb)<< endl;  }
```

Draugiška klasė

C++ leidžia apibrėžti ne tik draugiškas funkcijas, bet ir klases. Draugiškai klasei suteikiamos teisės pilnai naudoti duotosios klasės duomenimis.

Tam tikslui reikia, kad klasėje **A** būtų skelbiama, kad klasė **B** yra draugiška. Tai daroma parašant atributą **friend** prieš draugišką klasę. Draugiška klasė skelbiama klasės dalyje *public*.

Taisyklės, taikomos draugiškoms klasėms*:*

- Negalioja abipusės draugystės savybė t.y. jei klasė **B** yra draugiška klasei **A**, tai nereiškia, kad klasė **A** draugiška klasei **B**.
- Draugiškos klasės nepaveldimos. Jei klasė **B** yra draugiška klasei **A**, tai klasėse, kurios yra gautos, kaip **B** išvestinės, nėra draugiškos klasei **A**.
- Draugiškumas nepaveldimas t.y. jei **B** yra draugiška klasei **A**, tai klasėse kurios yra gautos, kaip **A** išvestinės, klasė **B** nėra draugiška.

Draugiškos klasės pavyzdys

```
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(2,6);
  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(); }
```

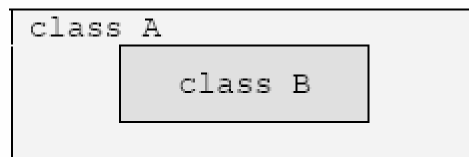
Draugiškos klasės pavyzdys

```
#include <iostream.h>
class alfa
{ int data1;
    public:
          alfa() : data1(99) { }
          friend class beta ;
};
class beta
{   public:
         void func1(alfa a)
           { cout<< "data1 reikšmė"<<a.data1<< endl;}
         void func2(alfa a)
           { cout<< "data1 reikšmė"<<a.data1++<< endl;}
};
void main()
{ alfa AA; beta BB;
  BB.func1(AA);
  BB.func2(AA); }
```

Paveldėjimas

Objektiniame programavime viena svarbiausių savybių, apibrėžiančių objektų tarpusavio sąveiką, - tai **paveldėjimas** (*angl. inheritance*).

C++ klasės gali paveldėti kitos klasės ar net kelių klasių duomenis ir metodus. Paveldėjimo vienas iš didžiausių privalumų – tai galimybė ir ateityje panaudoti klasę, šiek tiek ją modifikavus.



Klasė B, kurią paveldi klasė A, vadinama **protėviu** (bazinė klasė). Klasė A, kuri šalia savo duomenų ir metodų paveldi kitą klasę B, vadinama **palikuonimi** (išvestinė klasė). 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 paveldėjimo atributą *public*.

Išvestinės klasės apibrėžimas

Išvetinė klasė apibrėžiama:

```
class Palikuonis : [identifikatorius] Bazinė_klasė1, [identifikatorius] Bazinė_klasė2
{ ……. };
```

pvz. class Palikuonis : public Bazine
{…..};

[identifikatorius] gali turėti tokias reikšmes: *public, private, protected*. Jis nėra privalomas. Jei identifikatorius praleistas, laikoma, kad jo reikšmė *private*.

Identifikatoriaus paskirtis – nustatyti išvestinės klasės paveldimumo lygį, t.y. kokios srities metodus ar duomenis gali paveldėti išvestinė klasė.

Bazinė klasė neturi jokiu teisių į duomenis ir funkcijas, apibrėžtus išvestinėje klasėje.

Pavyzdys

```
#include <iostream>
using namespace std;
class Skaitliukas
{ protected: int x;
  public:
        Skaitliukas(): x(0)
        { }
        int Getx()
        { return x; }

       void plius()
       {x++;}
};
class MinusSkaitliukas: public Skaitliukas
{ public:
    int minus()
    { return x--;}
};
```

```
main ()
{ MinusSkaitliukas m1;
  cout<<m1.Getx()<<endl;

  m1.plius();
  cout<<m1.Getx()<<endl;
  m1.minus();
  cout<<m1.Getx()<<endl;

return 0; }
```

Paveldėjimas ir pasiekiamumas

Sritys bazinės klasėje	Pasiekimas iš bazinės klasės	Pasiekimas iš išvestinės klasės	Pasiekimas iš išorinių funkcijų
public	Taip	taip	taip
protected	taip	taip	ne
private	taip	ne	ne

Paveldėjimo lygiai

Paveldėjimo identifikatorius	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

Pavyzdys

```
class Base
{int x, y;
   public:
      int GetX() { return x;}
      int GetY() { return y;}
};
class Derived : private Base
{ public:
     void showX()
        { cout<< GetX()<< endl;}
};
```

```
main()
{   Derived A;
    A.showX();
    return 0;
}
```

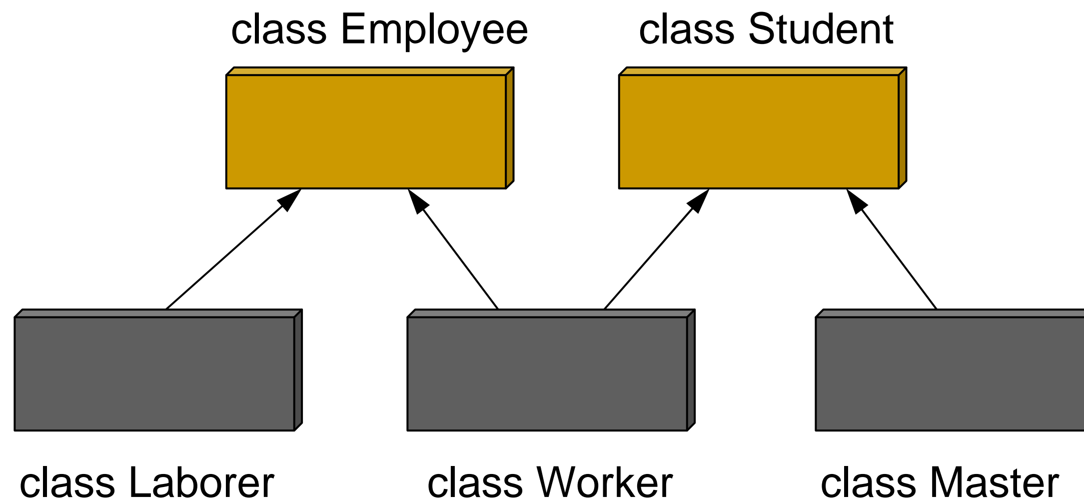
Pavyzdys

```
#include <iostream.h>

class A
{private:  int PrivateData;
 protected: int ProtectedData;
 public: int PublicData;
};

class B : public A
{public: void func()
   { int a;
     a= PrivateData;        // klaida
     a= ProtectedData;     // OK
     a= PublicData;         // OK
     }
};
```

```
class C : private A
{public: void func()
   { int a;
     a= PrivateData;        // klaida
     a= ProtectedData;     // OK
     a= PublicData;         // OK
     }
};
void main()
{ int a;   B objB;  C objC;
  a=objB.PrivateData;      // klaida
  a=objB.ProtectedData;  // klaida
  a=objB.PublicData;        // OK

  a=objC.PrivateData;      // klaida
  a=objC.ProtectedData;  // klaida
  a=objC.PublicData;        // klaida }
```

Nepaveldimi metodai, herarchija

- Konstruktorius
- Kopijos konstruktorius
- Destruktorius
- Priskyrimo operatorius, kurį apibrėžia pats programuotojas
- Draugiškos klasės



Išvestinės klasės konstruktorius

```
class Distance
{ int feet; float inches;
 public:  Distance() : feet(0), inches(0.0) { }
  Distance(int ft, float in) : feet(ft), inches(in) { }
  void showdist() const
       {cout<<feet<< "  "<<inches<,endl; }
};
class Sign: public Distance
{ char sig;
public: Sign() : Distance() { sig = '+'; }
Sign(int ft, float in, char sig1) : Distance (ft, in)
{sig = sig1; }
void showdist() const
{cout<<sig<< Distance::showdist(); }
};
```

```
main()
{ Sign alfa;
  Sign beta(11, 4.32, '+');
   alfa.showdist();
   beta.showdist();
   return 0;
}
```

Išvestinės klasės konstruktorius

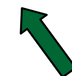
Išvestinė klasė turi apibrėžti savo konstruktorių, nes bazinės klasės konstruktorius nepaveldimas. Išvestinės klasės konstruktorius turi priskirti pradines reikšmės ir bazinės klasės duomenims, ir saviems ir **tai turi padaryti ankščiau nei savo klasės**. Jei turime klasę, kuri paveldi keletą bazinių klasių, tuomet svarbus ir bazinių konstruktorių užrašymo eiliškumas.

Apibrėžiant konstruktorių naudojama sintaksė:

Išvestinės_klasės_konstr(argumentai) : bazinės_klasės_konstr(argumentai1)
{konstruktoriaus tekstas}

Pvz.

```
Sign (int ft, float in, char sig1) : Distance (ft, in)
    {sig = sig1; }
```



Bazinės klasės konstruktoriaus kvietimas

Išvestinės klasės konstruktorius

```
#include <iostream>
using namespace std;
// Klasė Pirmas
    class Pirmas
    { int x;
      public:  Pirmas(int a) { x = a; }
               void Rodo()
               { cout<<"Pirmas::rodo()“<< x<<endl; }
};

// Klasė Antras
   class Antras: public Pirmas
   { int y;
    public:  Antras(int b, int c) : Pirmas(b)
             { y = c; }
             void Rodo()
             {cout<< "Antras::rodo()” << y << endl; }
};
```

```
void main ()
{ Antras B (4, 55);
  B.Rodo();
  B.Pirmas::Rodo();
}
```

Destruktorius

Nors destruktorius nėra paveldimas, uždarinėjant išvestinės klasės objektą, vistiek įvykdomas bazinės klasės destruktorius.

Išvestinės klasės destruktorius turi būti vykdomas ankščiau nei bazinės, priešingu atveju bus sunaikinti bazinės klasės duomenys.

Destruktoriaus vykdymo eiliškumu rūpinasi kompiliatorius t.y. programuotojo tai neliečia.

```
class Base
{ public: Base()
      { cout << "Bazinis konstruktorius"<<endl; }
      ~Base()
       {cout<<"Bazinis destruktorius \n";
       }
};
```

```
class Derived : public Base
{ public: Derived()
      { cout << "Išvestinis konstruktorius"<<endl;
      }
      ~Derived()
       {cout<<"Išvestinis destruktorius \n";
       }
};
```

Funkcijų pakeitimas (overriding)

Išvestinėje klasėje kuriant savus metodus, galima pakeisti jau egzistuojančius bazinės klasės metodus. Tą patį galima atlikti ir su duomenimis. Toks funkcijų pakeitimas atliekamas sukuriant to paties pavadinimo funkciją, turinčią ta patį argumetų sąrašą, kaip ir bazinėje klasėje.

Esant tokiai situacijai, gali iškilti neapibrėžtumų, kuri f-ja bus naudojama: iš bazinės ar išvestinės klasės.

Taisyklė: jei bazinėje ir išvestinėje klasėje egzistuoja tos pačios funkcijos, vykdoma funkcija iš išvestinės klasės (išvestinės klasės objektui). Jei turime bazinės klasės objektą, vykdoma bazinės klasės funkcija.

Norint išvestinės klasės objekte išsikviesti bazinės klasės objektą naudojama sintaksė:

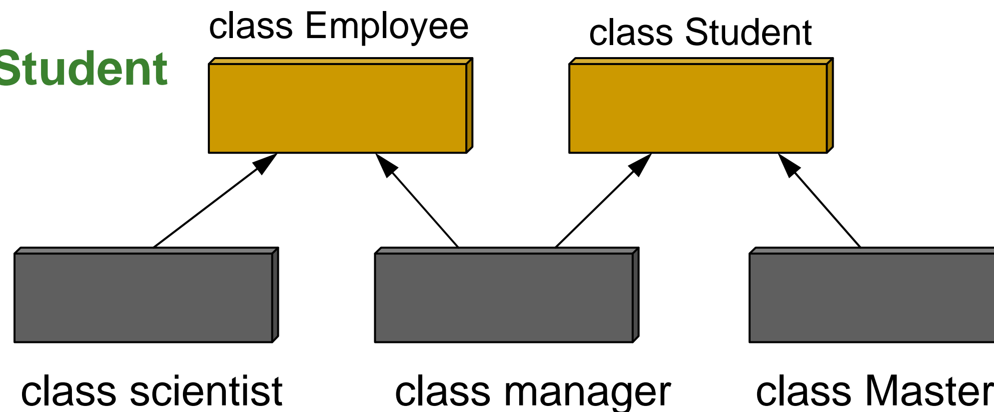
obj.baze::funkcija(...);

Pvz. **B.Pirmas::Rodo();** Pavyzdys pateiktas prieš 2 skaidres

Daugybinis paveldėjimas

Jei išvestinė klasė turi kelias bazines klases, turime daugybinio paveldėjimo atvejį.

```
class Student
{ };
class Employee
{ };
class manager : private Employee, private Student
{ };
class scientist : private Employee
{ };
class Master : private Student
{ };
```



Daugybinio paveldėjimo pavyzdys

```
#include <iostream.h>
class Coord
{int x,y;
   public:
   Coord(int x_init, int y_init) {x=x_init;  y=y_init)
   int GetX() { return x;}
   int GetY() { return y;}
};
class SaveMsg
{char Message[80];
 public:
      SaveMsg(char* msg) {strcpy(Message, msg); }
      void Show() { cout<< Message;}
};
class PrintMsg: public Coord, public SaveMsg
{ public: PrintMsg(int _x, int _y, char *msg): Coord(_x, _y), SaveMsg(*msg)
              { }
};
```

Daugybinio paveldėjimo pavyzdys

```
void main()
{ PrintMsg *ptr;
  ptr=new PrintMsg(10, 5,”Centras”);
  ptr -> Show();
  cout << “Koordinate X=“ << ptr -> GetX() << endl;
  cout << “Koordinate Y=“ << ptr -> GetY() << endl;
}
```

Atminties išskyrimo operatorius:

Sintaksė

ptr = **new** DuomTipas[dydis]

Pvz. char ptr1=new char[10];   float ptr2= new float[20];

Atminties atlaisvinimo operatorius:

delete [ ] ptr; //jei ptr masyvas   arba   **delete ptr**; // jei ptr vienas objektas