C++ programavimo kalba

Polimorfizmas ir virtualios funkcijos

(8 paskaita)

Kam reikalingos virtualios funkcijos?

C++ virtualumas suprantamas, kaip polimorfizmo ir paveldėjimo savybių apjungimas.

Iki šiol nagrinėtos perkrautos funkcijos ar operatoriai buvo nesiejami su klasių hierarchine struktūra.

Virtualios funkcijos naudojamos tuomet, kai tuo pačiu pavadinimu egzistuoja funkcijos tiek bazinėje tiek ir išvestinėse klasėse.

Virtuali – **reiškia neegzistuojanti realybėje**, o tai suprantama, kad kviečiant funkciją iš vienos klasės, realiai vykdoma kitoje klasėje esanti funkcija.

Pavyzdys:

```
shape* ptr_array [100];          // 100 rodyklių masyvas į skirtingus objektus
for (int i=0; i<N; i++)
      ptr_array[ i ] -> draw( );  // ta pati funkcija draw() kviečiama iš
                                     skirtingų objektų
```

Problema…nr.1

```
BankAccount bretta;      // bazinės klasės objektas
Overdraft ophelia;       // išvestinės klasės objektas
bretta.ViewAcct();       // BankAccount::ViewAcct()
ophelia.ViewAcct();      // Overdraft::ViewAcct()
```

→ Vienareikšmiškai apibrėžta

```
BankAccount bretta;
BankAccount * bp = &bretta;   // rodyklė į BankAccount objektą
bp->ViewAcct();               // BankAccount::ViewAcct()
bp = &ophelia;                // BankAccount rodyklė į Overdraft objektą
bp->ViewAcct();               // kuris ViewAcct metodas bus kviečiamas?
```

↓ Atsakymas

Pagal nutylėjimą, C++ naudoja rodyklės arba nuorodos tipą, kad nuspręstų, kokią f-ją pasirinkti. ***Todėl bus naudojama BankAccount::ViewAcct()***

Problema…nr.2

Tačiau kompiliatorius kompiliavimo metu dažnai nežino, su kokiu objektu bus surišta nuoroda ar rodyklė. Pavyzdžiui:

```
cout << "Enter 1 for Brass Account, 2 for Brass Plus Account: ";
int kind;
cin >> kind;
BankAccount * bp;
if (kind == 1)
    bp = new BankAccount;
else if (kind == 2)
    bp = new Overdraft;
bp->ViewAcct();              // neaišku, kuri funkcija bus iškviesta
```

Apibrėžimai

Jei programos kompiliavimo metu, kompiliatorius pririša rodyklę ar nuorodą prie konkretaus objekto ir taip nustato, kurią iš perkrautų funkcijų naudoti, toks pririšimas vadinamas ankstyvuoju (*early binding*) arba statiniu (*static binding*). Kartais dar sakoma, jog tai **statinio polimorfizmo** atvejis.

```
// static binding
BankAccount *bp;
Overdraft ophelia;
bp = &ophelia;              // BankAccount rodyklė į Overdraft objektą
bp->ViewAcct();             // kviečiame BankAccount::ViewAcct()
```

Jei programos kompiliavimo metu, kompiliatorius nepririša rodyklės ar nuorodos prie konktetaus objekto ir palieka galimybę programos vykdymo metu nuspęsti, kurio objekto metodą iškviesti, toks atvejis vadinamas vėlyvuoju (*late binding)* arba dinaminiu pririšimu (*dynamic binding).* Sakoma, jog turime **dinaminio polimorfizmo** atvejį.

Dinaminio pririšimo (dynamic binding) aktyvizavimas

Dinaminis pririšimas aktyvizuojamas tik klasės metodams. Norint tai atlikti **bazinėje funkcijoje perkrauta funkcija apibrėžiama kaip virtuali** (naudojamas raktinis žodis **virtual**).

Jei metodas apibrėžtas kaip virtualus, jis toks išlieka visose paveldėtose (išvestinėse) klasėse.

Vienam virtualiam metodui raktinis žodis naudojamas tik vieną kartą ir tik bazinėje klasėje.

Pavyzdys

```
// bankacct.h – BankAccount klasė su virtualiomis funkcijomis
#ifndef _BANKACCT_H_
#define _BANKACCT_H_

class BankAccount  {
private:  const int MAX = 35;
          char fullName[MAX];
          long acctNum;
          double balance;

public:   BankAccount (const char *s = "Nullbody", long an = -1, double bal = 0.0);
          void Deposit (double amt);
          virtual void Withdraw (double amt);        // virtualus metodas
          double Balance() const;
          virtual void ViewAcct() const;             // virtualus metodas
};
```

Pavyzdys (tęsinys)

```
class Overdraft : public BankAccount {
private:  double maxLoan;
          double rate;
          double owesBank;

public:   Overdraft (const char *s = "Nullbody", long an = -1, double bal = 0.0,
                     double ml = 500, double r = 0.10, owes = 1.0);
          Overdraft (const BankAccount & ba, double ml = 500, double r = 0.1,
                     owes = 1.0);
          void ViewAcct() const;
          void Withdraw(double amt);
          void ResetMax(double m) { maxLoan = m; }
          void ResetRate(double r) { rate = r; }
          void ResetOwes() { owesBank = 0; }
};

#endif
```

Pavyzdys (tęsinys)

```
int main() {
    BankAccount * baps[10];
    char name[MAX];
    long acctNum;  double balance;  int acctType;
for (int i = 0; i < 10; i++) {
   // su cin >> nuskaitomi name, acctNum, balance, acctType

if (acctType == 2)
            baps[i] = new Overdraft(name, acctNum, balance, max, rate, ow );
else
           { baps[i] = new BankAccount(name, acctNum, balance);
                if (acctType != 1)
                     cout << "Įvestas skaičius interpretuojamas kaip 1 \n"; }
}
for (i = 0; i < 10; i++)
    { baps[i]->ViewAcct();       // dėl šios eilutės reikėjo apibrėžti virtualius metodus
       cout << endl; }
 return 0; }
```

Kodėl naudojamos du polimorfizmo tipai?

Dinaminis polimorfizmas leidžia pakeisti klasės metodus – tai patogu kuriant didelius programinius projektus (>1000 eilučių).

Statinio polimorfizmo naudojimas duoda didesnį programos vykdymo efektyvumą. Programos vykdymo metu nevykdomas funkcijų apibrėžimas iš naujo bei jų atranka, taip taupomas programos vykdymo laikas.

Taisyklės:

• Virtualios funkcijos naudojamos tose paveldimose klasėse, kur reikalingas funkcijų apibrėžimas iš naujo.

• Statinės funkcijos negali būti apibrėžtos virtualiomis.

• Virtuali funkcijos prototipas visose klasėse turi būti identiškas t.y. sutapti parametrų sąrašas bei grąžinamos reikšmės tipas.

Virtualios ir nevirtualios funkcijos pavyzdys

```
#include <iostream>
using namespace std;
class Base
{ public:  virtual void VirtFunc()
           { cout << “Base::VirtFunc()\n”; }
           void show()
           {cout << “Base klases funkcija \n”; }
};
class Derived: public Base
{ public:  void VirtFunc()
           { cout << “Derived::VirtFunc()\n”; }
           void show()
           { cout << “Derived klases funkcija \n”;}
};
```

Virtualios ir nevirtualios funkcijos pavyzdys

```
void main ()
{Derived A_isvestine;
 Derived *ptr_isvestine = &A_isvestine;

 Base  *ptr_base = &A_isvestine;

 ptr_base->VirtFunc();          // virtualios funkcijos iškvietimas iš Derived
 ptr_base->show();              // nevirtuali funkcija iš Base


 ptr_isvestine->VirtFunc();     // virtualios funkcijos iškvietimas iš Derives
 ptr_isvestine->show();         // nevirtuali funkcija iš Derived
}
```

Virtualių metodų pavyzdys

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

class Figura {
 protected:
   int width, height;
 public:
   void set_values (int a, int b)
     { width=a; height=b; }
   virtual int area ( )
     { return (0); }
 };

class Staciakampis: public Figura {
 public:
   int area ()
     { return (width * height); }
 };

class Trikampis: public Figura {
 public:
   int area ()
     { return (width * height / 2); }
 };
```

Tęsinys

```
int main () {
  Staciakampis rect;
  Trikampis trgl;
  Figura poly;
  Figura * ppoly1 = &rect;
  Figura * ppoly2 = &trgl;
  Figura * ppoly3 = &poly;

  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);

  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  cout << ppoly3->area() << endl;

 return 0;}
```

```
int main () {
  Staciakampis rect;
  Trikampis trgl;
  Figura poly;
  Figura * ppoly [3];
  ppoly [0] = &rect;
  ppoly [1] = &trgl;
  ppoly [2] = &poly;

  for (int i = 0; i< 3; i++)
  {  ppoly [i]->set_values (4,5);
     cout << ppoly[i]->area() << endl;
  }
delete ppoly;
return 0;
}
```

Virtualus destruktorius

Konstruktoriai negali būti virtualūs, tačiau destruktoriai gali. Kai šalinamas išvestinės klasės objektas, o destruktorius yra apibrėžtas kaip virtualus, vykdomi tiek išvestinės tiek ir bazinės klasės destruktoriai.

Priešingu atveju iškviečiamas tik išvestinės klasės destruktorius ir atmintyje lieka nepanaikinti bazinio objekto duomenys.

```
class Base
{ public: ~Base()
          // virtual ~Base()     taip turėtų būti
          {cout<< “Base destroyed \n”; }
};
class Derived: public Base
{public: ~Derv()
          {cout<< “Derived destroyed \n”; }
};
```

```
int main()
{Base *pBase = new Derived;
delete pbase;
return 0;
}
```

Daugybinis paveldėjimas, virtualios bazinės klasės

Siekiant išvengti dvigubo bazinės klasės įtraukimo, naudojamos virtualios bazinės klasės. Tam prieš paveldimumo identifikatorių rašomas žodis *virtual*.

```
class IndBase
{   int x;
    public:
    int GetX() {return x; }
    void setX (int _x) { x=  _x ;}
    double var;
};
Class Base1: virtual public Indbase
{ …. };
Class Base2: virtual public Indbase
{ …. };
```

```
Class Derived: public Base1, public Base2
{ …. };

main ()
   Derived ob;
   ob.var = 5.0
   ob.SetX(0);
   int z = ob.GetX();
return 0; }
```

Abstrakčios klasės ir *pure virtual* funkcijos

Egzistuoja klasės, kurios nėra naudojamos objektams kurti, o tik tam, kad sudarytų reikiamą klasių hierarchijos medį ir užtikrintų logišką paveldimumą. Tokios klasės vadinamos abstrakčiomis (*abstract class*).

Pavyzdžiui: bazinė klasė *forma* ir klasės *trikampis*, *apskritimas*, *keturkampis* ir t.t

Akstrakti klasė turi bent vieną virtualią funkciją, o tiksliau tikrą virtualią (*pure virtual*). Išvestinės klasės turi būtinai panaudoti abstrakčios klasės virtualias funkcijas, priešingu atveju jos taip pat tampa abstrakčiomis (dėl paveldimumo).

Tikrai virtualia funkcija vienareikšmiškai pasako, kad klasė, kuriai ji priklauso yra abstrakti ir objektas iš tokios klasės negali būti sukurtas.

Tikrai virtutalios funkcijos sintakse:

```
virtual <funkcijos_pavadinimas> (argumentai) = 0;
```

Pure virtual funkcijos pavyzdys

```
class Base
{ int x;
  public: Base( int xx) {x = xx;}
  virtual int GetX()      {return x;}
  virtual void PrintX () = 0;
};
class Derived: public Base
{ public: Derived(int xx): Base (xx)
              { }
              int GetX() { return 0;}
              void PrintX()
              {cout<< "x= " <<GetX()<<"  \n"; }
};
```

Pavyzdys

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

class Figura {
 protected:
   int width, height;
 public:
   void set_values (int a, int b)
     { width=a; height=b; }
   virtual int area ( ) = 0;

 };

class Staciakampis: public Figura {
 public:
   int area ()
     { return (width * height); }
 };
```

```
class Trikampis: public Figura {
 public:
   int area ()
     { return (width * height / 2); }
 };
```

Tęsinys

```
int main () {
  Figura * ppoly [3];
  ppoly [0] = new Staciakampis;
  ppoly [1] = new Trikampis;
  ppoly [2] = new Figura;

  for (int i = 0; i< 3; i++)
  {  ppoly [i]->set_values (4,5);
     cout << ppoly[i]->area() << endl;
  }
delete ppoly;
return 0;
}
```