C++ programavimo kalba

Virtualios bazinės klasės,

funkcijų perkrovimas

(6 paskaita)

Virtualios bazinės klasės

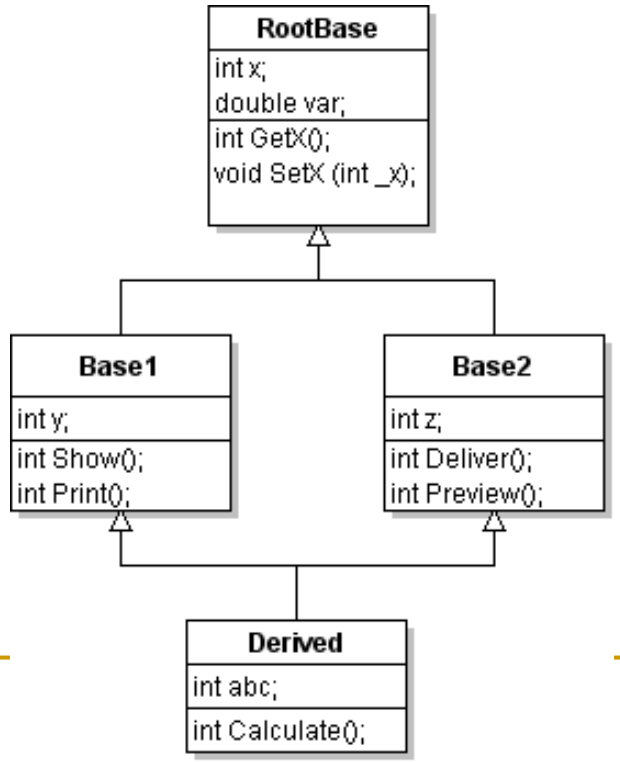
Formuojant sudėtingas klasių herarchijas bei susiduriant su daugybiniu paveldėjimu gali nutikti taip, kad išvestinė klasė paveldi du arba daugiau tos pačios bazinės klasės egzempliorių. Tokiu atveju kompiliatorius randa eilę klaidų, siejamų su nevienareikšmiu kreipimusi į klasės narius.

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

```
Class Derived: public Base1, public Base2
{ …. };
main ()
   Derived ob;
   ob.var = 5.0                  // blogai
   ob.SetX(0);                   // blogai
   int z = ob.GetX();            // blogai
   // ob.Base1::var = 5.0;       // gerai
   // ob.Base1::SetX(0);         // gerai
return 0; }
```

Virtualios bazinės klasės

Formuojant sudėtingas klasių herarchijas bei susiduriant su daugybiniu paveldėjimu gali nutikti taip, kad išvestinė klasė paveldi du arba daugiau tos pačios bazinės klasės egzempliorių. Tokiu atveju kompiliatorius randa eilę klaidų, siejamų su nevienareikšmiu kreipimusi į klasės narius.



Virtualios bazinės klasės

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

```
class RootBase
{   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
{ …. };

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

Virtualios bazinės klasės

Konstruktoriai, esant virtualioms paveldimoms klasėms **iškviečiami tokia tvarka:**

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

Jie turime keletą 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 galioja tos pačios taisyklės tik viskas vykdoma atvirkštine tvarka.

Abstrakčios bazinės klasės

Dažnai formuojant herarchines klasių struktūras, kuriamos klasės, kurios naudojamos tik tam, kad suformuotų loginę klasių struktūrą, tačiau realūs objektai iš tokių klasių niekad nekuriami.

Tokios klasės vadinamos **abstrakčiomis klasėmis**.

```
#include <iostream>
using namespace std;
class darbuotojas
{ char name[80];
  unsigned long id;
  public:
  void getdata()
    {cout<<“Vardas: ”; cin>>name;
     cout<<“id: ”; cin>>id; }
  void putdata() const
    {cout<<“Vardas: ”<<name<<endl;
     cout<<“id: ”<<id <<endl; } };
```

```
class vadovas :public darbuotojas
{ char pareigos[80];
  unsigned long alga;
  public:
   void getdata()
    {darbuotojas::getdata();
     cout<<“Pareigos: ”; cin>>pareigos;
     cout<<“Alga: ”; cin>>alga; }
   void putdata() const
     {darbuotojas::putdata();
     cout<<“Pareigos: ”<<pareigos <<endl;
     cout<<“Alga: ”<<alga <<endl; } };
```

Tęsinys

Realiai mums reikalingi tik tokie objektai, kurie apibrėžia konkretų darbuotoją su konkrečiomis pareigomis, todėl iš klasės darbuotojas, objektai nebus kuriami. Klasė *darbuotojas* vadinama abstrakčia.

```
class pardavejas :public darbuotojas
{ char pareigingas;
  public:
    void getdata()
    {darbuotojas::getdata();
     cout<<“Pareigingas? (T/N): ”;
     cin>>pareigingas; }
    void putdata() const
     {darbuotojas::putdata();
     cout<<“Pareigingas: ”<<pareigingas
     <<endl;
     }
};
```

```
void main()
 { vadovas AA;
    pardavejas BB;
    AA.getdata();
    AA.putdata();
    BB.getdata();
    BB.putdata();
 }
```

Perkrovimas (overloading)

Kodėl naudojamas perkrovimas ?

Funkcijų perkrovimas – tai vienas iš polimorfizmo tipų, naudojamas C++ kalboje. Jei keletas funkcijų turi tą patį vardą, jos laikomos perkrautomis (*overloaded*).

Perkrauti galima tik funkcijas, turinčias skirtingą argumentų skaičių arba jų tipą. Taip pat gali skirtis funkcijų tipas.

Argumentų tipas skiriasi, jei jiems apibrėžti naudojami skirtingi inicializatoriai.

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

```
# include <iostream.h>
// funkcijos abs perkrovimas
int abs (int a);
float abs (float b);
double abs (double c);
void main ()
{  int x = 255; float y = -1.2f; double  z = -15.0;
   cout << abs (x)<< endl;
   cout << abs (y)<< endl;
   cout << abs (z)<< endl;
}
```

```
int abs (int a)
{ return a<0? –a: a; }

float abs (float b)
{ return b<0? –b: b; }

double abs (double c)
{ return c<0? –c: c; }
```

Pavyzdys

```
# include <iostream.h>
// funkcijos repchr() perkrovimas

void repchr();
void repchr(char);
void repchr(char, int);

void main ()
{ repchr();
  repchr('=');
  repchr('+', 30);
}

void repchr()
{ for (int i=0; i<45; i++)
     cout<< '*';
  cout<<end;}

void repchr(char ch)
{ for (int i=0; i<45; i++)
     cout<< ch;
  cout<<end;}

void repchr(char ch, int n)
{ for (int i=0; i<n; i++)
     cout<< ch;
  cout<<end;}
```

Ekrane

```
*********************************************
=============================
++++++++++++++++++++++++
```

Funkcijų perkrovimo apribojimai

- Perkraunamos funkcijos privalo turėti skirtingus argumentų sąrašus;
- Negalimos perkrautos funkcijos, turinčios tuos pačius argumentų sąrašus ir skirtis tik grąžinamos reikšmės tipu.
- Klasės metodai nelaikomi perkrautais, jei vienas iš jų apibrėžtas kaip *static*, o kitas ne.
- *typedef* operatorius neįtakoja perkrovimo mechanizmui.

```
typedef char* ptr;
void SetVal (char * xx);
void SetVal (ptr xx);
```

Klaida, nes tai neperkrautos funkcijos (argumentų sąrašas tas pats)

Klasės metodų perkrovimo pavyzdys

```
class Lapas
{  private: int x; float y; char z;
   public:
   Lapas (int A, float B, char C) { x = A; y = B; z = C; }   // Konstruktorius
   Lapas() { x = 0; y = 0; z = 'z'; }                        // Konstruktorius
   void Rodo (char *);
     void Suma (int Sk)     { x = x + Sk; }
     void Suma (float Sk)  { y = y + Sk; }
     void Suma (char Sk)  { z = Sk; }
     void Suma (int A, char B) { x = x + 2.0 * A;  z = B + 4; }
};

void Lapas::Rodo(char *Eilute) {
cout << Eilute << " : ";
cout << " x= " << x << " y= " << y << " z= " << z <<endl; }
```

perkrautos funkcijos

Tęsinys

```
void main(void)
{
  Lapas A;
  Lapas B(5, 3.4, 'C');
  A.Rodo("Objektas A-1");
  B.Rodo("Objektas B-1");

  A.Suma(5);
  B.Suma('K');
  A.Rodo("Objektas A-2");
  B.Rodo("Objektas B-2");

  A.Suma((float)35.25);
  B.Suma(5, 'A');
  A.Rodo("Objektas A-3");
  B.Rodo("Objektas B-3");
  getch();
}
```

Klasės metodų perkrovimas

Perkrauti klasės metodai gali būti apibrėžti skirtingose klasės pasiekiamumo dalyse (pvz. vienas *private*, kitas *public* dalyje). Tačiau tai nekeičia metodų perkrovimui galiojančių taisyklių.

```
class AnyClass
{   private :
    int Func (int a)
      { …. }
    public:
    AnyClass();
    double Func (double b, char * c)
     { ….}
};
```

```
main()
{ AnyClass * ptr = new AnyClass;
  ptr -> Func(104);          //blogai
  ptr -> Func(12.2, “Eilute”);
  return 0;
};
```

Konstruktorių perkrovimas

Dažniausiai funkcijų perkrovimas naudojamas kuriant perkrautus konstruktorius. Taip daroma siekiant suteikti galimybę kurti objektus, naudojant skirtingus argumentų sąrašus.

Class Rect

```
{ private: int x,y,w,h;
  public: Rect()  {x=y=w=h=0; }
          Rect(int a, int b)  { x = a; y = b; w=h=10; }
          Rect(int a, int b, int c, int d)  { x = a; y = b; w = c; h=d; }
          Rect(const Rect&);
          Rect(const Rect&, int, int );
};
Rect::Rect(const Rect& rc)

{x=rc.x;   y=rc.y;
 w = rc.w; h = rc.h }
```

Konstruktorių perkrovimas

```
Rect::Rect( const Rect& rc, int _x, int _y)
{ x = _x;
  y = _y;
  w= rc.w;
  h = rc.h;
 }

void main ()
{ Rect ab;
  Rect bc(10, 20);
  Rect rc(3,4,5,6);
  Rect newrc(rc, 14, 34);
}
```

Kopijos konstruktorius

Kopijos konstruktorius naudojamas siekiant jau iš esamo objekto sukurti naują, kaip esamo kopiją.

Kintamųjų kopijos daromos, kai jie perduodami funkcijoms per argumentų sąrašą. Dėl tos priežasties C++ buvo sukurtas kopijos konstruktorius, kuris automatiškai generuojamas kuriant objektus.

Tokiu būdu panaudojant tokią sintaksę:

```
AnyClass obj1;
AnyClass obj2(obj1);
```

atliekamas objekto kopijavimas.

Argumentų sąraše naudojant rodykles į objektą, turime reikalą su tuo pačiu objektu, o ne jo kopija.

Perkrovimas ir nevienareikšmiškumas

Kompiliatorius funkcijas išrenka pagal geriausio atitikimo principą t.y.

1. Rasta tikslaus pavadinimo f-ja;
2. Rasta tikslių argumentų f-ja;

Jei po tokios atrankos, lieka daugiau nei viena funkcija, kompiliatorius generuoja klaidą ir perkrovimas laikomas **nevienareikšmiu**.

Nevienereikšmiškumas randamas tuomet, kai kviečiama perkrauta funkcija.

Norint valdyti perkrautų konstruktorių paiešką, C++ turi dar vieną galimybę t.y. **explicit** identifikatorių panaudojimą. Toks identifikatorius neleidžia konstruktoriui atlikti neišreikštinį duomenų keitimą.

```
class T
{ public: explicit T(int);
          explicit T(double);};
```

```
void f(T)   { };      //apibrėžiama funkcija
void g(int i);
{f(i); }    //blogai, neišreikštinis tipo
              pakeitimas
```

Perkrautos funkcijos adresas

Funkcijos adresas randamas tokiu būdu: rodyklei, kurios tipas atitinka funkcijos tipą priskiriama ieškoma funkcija tik be argumentų sąrašo t.y. skliaustai turi būti tušti ().

Pavyzdys:

```
int Func(int i, int j);
int Func(long l);
```

} *perkrautos funkcijos*

```
int (*ptr)( int,  int) = Func;    // randamas pirmos funkcijos adresas
ptr(10,20);                       // išviečiama funkcija
```

Turint perkrautas funkcijas, atranka atliekama pagal bendras taisykles t.y. ieškoma funkcija, kuri tiksliai atitinka užduota rodyklės funkcijos tipą. Jei atsiranda nevienereikšmiškumas, generuojama klaida.

Operatorių perdengimas (overloading)

Programavimo kalbose naudojami operatoriai pasižymi polimorfizmu (daugiavariantiškumu). Kaip pavyzdys gali būti operatorius **+**, kuris taikomas tiek *int*, tiek *float* tipo kintamiesiems, nors sudėties veiksmai procesoriaus registruose atliekami nevienodai.
Klasėse galima naujai apibrėžti operatorius, t.y. pakeisti jų veikimą. Toks naujas operatorių perorientavimas yra vadinamas **perdengimu.**
Operatorių aprašai nuo funkcijų skiriasi vartojamu žodžiu **operator**.

Operatoriaus perdengimo sintaksė *:*

operator Operatoriaus_simbolis (…);

Perdengimas draudžiamas operatoriams: . .* :: ?:

Operatorių perdengimas gali galioti lokaliai t.y. klasės ribose arba globaliai t.y. visoje programoje.
Operatoriai – tai funkcijos, todėl operatorių perkrovimas siejamas su funkcijų perkrovimu.

Operatorių perdengimas

```
#include <iostream.h>
#include <string.h>

class Katinas
{private:
  char *Vardas;
 public:
   Katinas(char *);
   void operator + (char *);
   void operator - (char );
   void Rodo();
 ~Katinas();
};
```

```
Katinas::Katinas(char *Eilute)
{  Vardas = new char[50];
    strcpy(Vardas, Eilute); }

void Katinas::Rodo()
  { cout << Vardas << endl; }

void Katinas::operator + (char * A)
{ strcat(Vardas, A); }

void Katinas::operator - (char C)
{ for(int i=0; *(Vardas+i) != '\0'; i++ )
   { if(*(Vardas + i) == C)
        for(int j=i; j<(strlen(Vardas)-1); j++)
        *(Vardas + j) = *(Vardas + j+1);
   } }
```

Operatorių perdengimas

```
Katinas::~Katinas()
{ delete Vardas; }

// Pagrindinė programa

void main(void) {
Katinas A ("Batuotas ir piktas");
A.Rodo();
A + " Katinas! ";
A.Rodo();
A - 'a';
A.Rodo();

}
```

Kas ekrane?