9-10-11 PASKAITOS

Turinys:
Funkcijos: virtualios, grynai virtualios, draugiškos, statinės.
Klasės: abstrakčios, draugiškos.
Dinaminė informacija apie objektų tipus.

Virtualios funkcijos

Tokios funkcijos leidžia realizuoti polimorfizmą – vieną iš pagrindinių OOP principų. Polimorfizmo idėja: tarkim, turim jau matytas bazinę klasę *Shape* ir ir jos išvestines klases *Quad, Rectangle* ir *Triangle*; visos jos turi metodą *draw( )*. Kelias figūras sugrupuojam į rodyklių masyvą ir norim visas jas nubraižyti metodu *draw( )*:

```
. . .
Shape* parray[ 100 ];
for( int i = 0; i < n; i++ ) parray->draw( );
. . .
```

Taigi, masyve yra įvairios figūros, o skirtingi jų braižymo metodai *draw( )* kviečiami lygiai taip pat – t.y. galim užmiršti klasių skirtumus ir kodą rašyti vien tik baziniam duomenų tipui. Antras tokio stiliaus pranašumas – labai lengva išplėsti kodą, pavyzdžiui, nauju figūros tipu: reikia tik parašyti atitinkamą figūros klasę ir jos metodą *draw( )*, jos objektą įdėti į rodyklių masyvą, o daugiau jokių keitimų nereikės. Polimorfizmui realizuoti reikia dviejų dalykų: kad klasės būtų susietos paveldimumu panašia schema kaip šiame pavyzdyje, ir kad *draw( )* bazinėje klasėje būtų virtuali funkcija.

Prieiga prie metodų ir virtualių funkcijų rodyklėmis

Prieiga prie metodų:

```
#include <iostream>
using namespace std;
//
class Base{
        public:
                void show( ){
                        cout<<"Base\n";
                }
};
//
class Derived1: public Base{
        public:
                void show( ){
                        cout<<"Derived1\n";
                }
};
```

```
//
class Derived2: public Base{
        public:
                void show( ){
                        cout<<"Derived2\n";
                }
};
//
int main( ){
        Derived1 d1;
        Derived2 d2;
        Base* p;
        p = &d1;
        p->show( );
                p = &d2;
                p->show( );
                        return 0;
}
```

Rezultatai:

```
        Base
        Base
```

Šioje programoje taikytas "kylantysis tipų pertvarkymas": bazinio tipo rodyklę galima nukreipti į išvestinių tipų objektų atminties sritį; tam išvestiniai tipai pertvarkomi į bazinį tipą.
Programos rezultatai rodo, kad kompiliatorius nežiūri į turinį atminties srities, į kurią rodo rodyklė – metodą renkasi pagal rodyklės tipą. Taip polimorfizmo realizuoti negalima. Toks kreipinio į metodą ir metodo kūno susiejimas vadinamas "statiniu saistymu".
Programos bazinę klasę pakeitus taip:

```
. . .
class Base{
        public:
                virtual void show( ){
                        cout<<"Base\n";
                }
};
//
. . .
```

rezultatai yra:

```
        Derived1
        Derived2
```

Kompiliatorius ir šiuo atveju nežino, kokį metodą reikia kompiliuoti konkrečiu atveju, todėl visada kompiliuojamas bazinės klasės metodas. Koks metodo kūnas bus susietas

su konkrečiu kreipiniu *show( )*, paaiškės tik programos vykdymo metu, kai bus aišku, į kokią atminties sritį nutaikyta rodyklė. Tai – "vėlyvasis saistymas" arba "dinaminis saistymas". Toks saistymas reikalauja daugiau kompiuterio išteklių, nei statinis saistymas.
Klasė, kurioje yra bent viena virtuali funkcija, dar vadinama polimorfine.

Grynai virtualios funkcijos. Abstrakčios klasės

Dažnai (taip pat ir nagrinėtuoju atveju) bazinė klasė reikalinga tik kaip išvestinių klasių kūrimo įrankis; jos objektų neturi būti. Objektų kūrimą uždrausti galima į bazinę klasę įdedant "grynai virtualią funkciją":

```
. . .
class Base{
        public:
                virtual void show( ) = 0;
};
. . .
```

Taip pakeitus programą, rezultatai liktų tie pat.
Kaip matyti, grynai virtuali funkcija išvis neturi kūno (bet jį leidžiama rašyti). Funkcijos prieskyra nuliui – tik sutartinė sintaksė grynai virtualiai funkcijai žymėti.
Klasė, turinti bent vieną grynai virtualią funkciją – "abstrakti klasė".

Draugiškos funkcijos ir klasės

Tai – įrankis prieigai prie klasės laukų iš funkcijų.
Jei funkcija nėra klasės metodas, ji neturi prieigos prie klasės *private* laukų. Tai nepatogu, jei funkcija turi dirbti su kelių klasių laukais.
Pavyzdys: išorinė funkcija sumuoja dviejų klasių privačius laukus. Kad ji turėtų prieigą prie jų, ji skelbiama draugiška – *friend*, ir jos prototipai įdedami į abi klases.

```
#include <iostream>
using namespace std;
//
class B;     // 1 pastaba
//
class A{
        private:
                int data;
        public:
                A( ): data( 0 ){ }
                friend int friendlyfunc( A, B  );   // 2;  prototipas
};
//
class B{
        private:
                int data;
```

```
        public:
                B( ): data( 10 ){ }
                friend int friendlyfunc( A, B );   // 2;  prototipas
};
//
int friendlyfunc( A a, B b ){
        return (a.data + b.data );
}
//
int main( ){
        A a;
        B b;
        cout<< "Sum is: "<<friendlyfunc( a, b )<<endl;
        return 0;
}
```

Pastabos:

1. Kadangi draugiškos funkcijos prototipo eilutėje žemiau yra sutinkamas tipas *B*, prieš tai reikia parodyti, jog tai – klasė.
2. Kur dėti draugiškos funkcijos prototipą – nesvarbu: gali būti arba *private*, arba *public* sekcijoje.

Tokį mechanizmą reikėtų taikyti rezervuotai, nes pati draugiškų funkcijų idėja prieštarauja OOP principams (duomenų inkapsuliavimo principui).

Draugiškos klasės: visi draugiškos klasės metodai turi prieigą prie kitos klasės, kurioje yra draugiškos klasės skelbimas, *private* laukų.
Pavyzdys:

```
#include <iostream>
using namespace std;
//
class A{
        private:
                int data;
        public:
                A( ): data( 10 ){ }
                friend class B;   // skelbimas
};
//
class B{
        public:
                void m( A a ){
                        cout<<"Data: "<<a.data<<endl;  // yra prieiga
};
//
int main( ){
        A a;
        B b;
        b.m( a );
```

```
        return 0;
}
```

Bus:

```
        Data: 10
```

Statinės funkcijos

Statiniai gali būti ne tik laukai (bendri visiems klasių objektams), bet ir metodai (dažniausiai – dirbantys su statiniais laukais). Juos taip pat įprasta kviesti tik klasės vardu.
Pavyzdys: klasė, suskaičiuojanti, kiek sukurta jos objektų, ir visiems objektams suteikianti numerį. Perrašomas destruktorius, kuris naikindamas objektą sumažina objektų skaičių ir apie tai spausdina pranešimą. Kartu taps aiški destruktorių vykdymo tvarka.

```
#include <iostream>
using namespace std;
//
class C{
        private:
                static int total;  //  objektų skaičius
                int id;             //  objekto numeris
        public:
                C( ){
                        total++;
                        id = total;
                }
                ~C( ){
                        total--;
                        cout<<"Destroyed object No: "<<id<<endl;
                }
                static void schowTotal( ){
                        cout<<"Number of objects: "<<total<<endl;
                }
                void schowId( ){
                        cout<<"Number of object: "<<id<<endl;
};
//
int C::total = 0;
//
int main( ){
        C c1;
        C::showTotal( );
        C c2, c3;
        C::schowTotal( );
                c1.showId( );
                c2.showId( );
                c3.showId( );
```

```
        cout<<"End of program\n";
        return 0;
}
```

Rezultatai:

```
Number of objects: 1
Number of objects: 3
Number of object: 1
Number of object: 2
Number of object: 3
End of program
Destroyed object No: 3
Destroyed object No: 2
Destroyed object No: 1
```

Taigi, objektai naikinami atvirkščia tvarka nei sukurti. Destruktoriai kviečiami automatiškai, kai objektas netenka į jį nutaikytos rodyklės. Šiuo konkrečiu atveju destruktoriaus galima neperrašyti – jokių prasmingų veiksmų į jį neįdėta. Tačiau kompiliatoriaus sukurti destruktoriai nenaikina objektų arba objektų laukų, sukurtų dinamiškai su *new* operacija. Taigi jei koks laukas objekte alokuotas dinamiškai, būtina perrašyti destruktorių ir jame su *delete* atmintį išlaisvinti – antraip *heap* atminties dalis bus prarasta – „memory leak“ klaida.

Dinaminė informacija apie objektų tipus

Informaciją teikia RTTI mechanizmas (Run-time Type Identification): programos vykdymo metu galima nustatyti objekto tipą ir kai kurą kitą informaciją, pavyzdžiui, klasės vardą. Taip pat leidžia keisti paveldimumu susijusių klasių tipus.
Programoje būtinas antraštinis failas <*typeinfo*> .
Dirbant MS DeveloperStudio programavimo terpėje, RTTI būtina suaktyvinti rankiniu būdu: terpėje reikia rinktis

<nuostatas> *Project* -> *Settings* <dabar kortelę> -> *C\C++* -> *Category* -> <dabar pasirinkti> -> C++ Language -> <automatiškai bus pažymėtas žymimasis laukelis varnele> *Enable RTTI*

Dinaminis tipų pervedimas: *dynamic_cast<naujasis tipas>(senasis tipas)* .
Informacija apie tipą: *typeid(objektas).name( )* .

Kartu programoje žemiau demonstruojamas raktažodis *this*: rodyklė į šiuo metu apdorojamą objektą. Todėl prieičiai prie einamojo objekto lauko reikia konstrukcijos *this->laukas*. *this* dažniausiai taikomas grąžinant einamąjį objektą iš funkcijos:
*return *this;* arba kai be tokio raktažodžio kyla vardų konfliktas (šis pavyzdys iliustruotas programos metoduose-konstruktoriuose).

```
#include <iostream>
#include <typeinfo>  // 1 pastaba
using namespace std;
//
class Base{
        protected:
                int b;
        public:
                Base( ): b( 0 ){ }
                Base( int b ) { this->b=b; }    // 2
                virtual void fictitious( ){ }      // 3
                 void show( ){
                        cout<<"Base: b= "<<b<<endl;
                }
};
//
class Derived: public Base{
        private:
                int d;
        public:
                Derived( int b,int d ){  this->d=d; this->b=b; }  //  2
                void show( ){
                        cout<<"Derived: b= "<<b<<" d= "<<d<<endl;
                }
};
//
int main( ){
        Base* pBase = new Base( 10 );
        pBase->show( );
        Derived* pDerived = new Derived( 21,22 );
                pBase = dynamic_cast<Base*>( pDerived );  // 4
                pBase->show( );
                        pBase = new Derived( 31,32 );
                        pBase->show( );
                        pDerived = dynamic_cast<Derived*>( pBase );  // 5
                        pDerived->show( );
                                pBase = pDerived;  // 6
                                pBase->show();
                                        return 0;
}
```

Pastabos:

1. Toks antraštinis failas būtinas RTTI mechanizmui.
2. Raktažodis *this*. Kartais taikomas toks programavimo stilius: metode-konstruktoriuje argumentų vardai renkami tokie pat, kaip ir laukų vardai. Todėl be *this* kiltų vardų konfliktas: nebūtų aišku, kas yra laukas, o kas – argumentas. *this* inicializavimo sąraše būti negali.
3. Reikia, kad klasė būtų polimorfinė – turi būti bent vienas virtualus metodas, todėl įtrauktas šis fiktyvus *virtual* skelbiamas metodas.

4. Pagrindinėje funkcijoje demonstruojami įvairūs tipų pertvarkymai paveldimumo schemoje – kylantysis (visada nepavojingas, galimas ir be RTTI) bei besileidžiantysis (pavojingas pertvarkymas; galimas tik veikiant RTTI). Šioje eilutėje yra išreikštinis (operatoriumi *dynamic_cast*) kylantysis pertvarkymas iš išvestinio tipo *pDerived* į bazinį *pBase*. Lygiai tą pat galima padaryt ir neišreikštiniu pertvarkymu *pBase = pDerived;* (kaip 6-a pastaba pažymėtoje eilutėje).
5. Besileidžiantis išreikštinis pertvarkymas. Neišreikštinis toks pertvarkymas – negalimas.

Rezultatai:

Base: b = 10
Base: b = 21
Base: b = 31
Derived: b = 31 d = 32
Base: b = 31