Lucrarea 5 Ierarhii de clase

Scopul lucrării: Proiectarea si folosirea ierarhiilor de clase folosind limbajul de programare C++.

Desfăsurarea lucrării: Se vor scrie programe în cadrul carora se vor defini si utiliza ierahii de clase.

- Mostenire simpla;
- Constructori si destructori în cadrul unei ierarhii de clase;
- Suprascrierea functiilor membre;
- Mostenirea multipla;
- Utilizarea modificatorilor de acces în cadrul unei ierarhii de clase.

Se vor utiliza paradigmele de abstractizare a datelor si mostenire a claselor, aplicatiile având fisiere header pentru declararea claselor si fisiere sursa pentru implementarea functiilor membre. Se va crea separat un fisier sursa pentru testarea claselor dezvoltate. Se vor identifica si proiecta modulele corespunzatoare pentru rezolvarea fiecarei aplicatii.

Reutilizarea claselor utilizând mostenirea simpla

Mostenirea reprezinta mecanismul prin care se pot crea obiecte noi, din cele existente; noile obiecte preiau (mostenesc) metode si date de la obiectul parinte, putând sa modifice alte metode (suprascrie). O clasa poate mosteni proprietati (date si metode) de la una sau mai multe clase, mecanismul având numele de mostenire simpla, respectiv mostenire multipla.

Prin mostenire se creaza o ierarhie de clase, iar acest fapt va avea influente în modul de folosire a functiilor constructor si destructor apelate la instantierea/distrugerea obiectelor din clase frunza ale ierarhiei (clase derivate).

Puteti folosi ca exemplu clasa Persoana si clasa derivata Student. Student are toate elementele clasei Persoana si in plus elemente de genul: string facultate, string grupa, float medieAniStudiu, metode de genul modificaGrupa(), modificaFacultate()

Sintaxa prin care se specifica faptul ca o clasa mosteneste (este derivata) o alta clasa, in limbajul C++, este urmatoarea:

specificator_de_clasa nume_de_clasa: specificator_acces nume_clasa_da_baza {

membri (de tip data sau functie) proprii clasei

} ;

unde:

- specificator_de_clasa poate fi **class** sau **struct** (nu se pot deriva clase noi din union);

- specificator_acces poate fi **private** sau **public**; daca specificator_acces este absent, atunci valoarea sa implicita este **private** pentru class, respectiv **public** pentru struct. O clasa derivata mosteneste toti membrii clasei de baza, dar poate accesa numai membrii **protected** si **public**. Membrii **private** ai clasei de baza nu sunt disponibili si nu pot fi mosteniti la urmatorul nivel din ierarhie; în schimb, membrii protected si public pot fi transmisi în jos, în cadrul ierarhiei cât timp derivarea este de tip public.

Exemplu 5.1

Scrieti o clasa From cu un membru variabila name. Scrieti un getter/setter pentru name. Folositi clasa. Derivati o clasa Rectangle din Form. Clasa Rectangle are membrii witdh si height. Scrieti pentru fiecare un getter si un setter. Pentru Rectangle, setati un name si un width si afisati-le.

Codul aferent Exemplului 5.1 este prezentat mai jos.

```
#include <iostream>

using namespace std;

class Form {
protected:
   string name;
public:
   string getName(){
      return this->name;
   }
   void setName(string name){
      this->name = name;
   }
};

class Rectangle:public Form {
protected:
   int width;
   int height;
public:
   void setWidth(int width){
      this->width = width;
   }
   int getWidth(){
      return this->width;
   }
};

int main()
{
   Form *f = new Form();
   f->setName("forma1");
   cout<<"forma: "<<f->getName()<<endl;

   Rectangle *r = new Rectangle();
   r->setWidth(23);
   r->setName("rectangle 1");
   cout<<"rectangle name: "<<r->getName()<<" width:"<<r->getWidth()<<endl;
   return 0;
}
```

Exercitii

E5.1 in clasa Form declarati private: string color; si un getter/setter, private sau protected. Accesati-le din Rectangle sau prin r si verificati daca sunt disponibili.

E5.2 Refactorizati Exemplul 5.1 si exercitiul anterior astfel incit sa folositi fisiere header.

Constructori si destructori în cadrul unei ierarhii de clase

Modul de declarare, implementare si utilizare a functiilor constructor si destructor sunt valabile si pentru clasele derivate. Pentru construirea unui obiect al clasei derivate, se creaza un obiect al clasei de baza si se apeleaza constructorul clasei de baza; apoi se creaza elementele proprii clasei derivate si se apeleaza constructorul clasei derivate. La distrugerea unui obiect al clasei derivate, este apelat întâi destructorul clasei derivate, apoi cel al clasei de baza.

Exemplul 5.2

Scrieti un constructor pentru clasa Form cu parametrul name si un constructor pentru clasa Rectangle cu parametrii string name, int width, int height. In exemplul de mai jos, observati codul scris cu rosu si modul in care se apeleaza constructorul clasei base (Form).

Codul aferent Exemplului 5.2 este prezentat mai jos.

```
#include <iostream>

using namespace std;

class Form {
private:
   string color;

protected:
   string name;
   void setColor(string color)   {
      this->color = color;
   }
public:
   Form(string name){
      this->name = name;
   }
   string getName(){
      return this->name;
   }
   void setName(string name){
      this->name = name;
   }
};

class Rectangle:public Form {
protected:
   int width;
   int height;
public:
   Rectangle(string name, int width, int height) :Form(name){
      this->width = width;
      this->height = height;
   }
   void setWidth(int width){
      this->width = width;
   }
   int getWidth(){
      return this->width;
   }
};

int main()
{
   Form *f = new Form("forma1");
   cout<<"forma: "<<f->getName()<<endl;

   Rectangle *r = new Rectangle("rectangle 1", 1, 2);
   cout<<"rectangle name: "<<r->getName()<<" width:"<<r->getWidth()<<endl;
   return 0;
}
```

Exercitii

Exercitiul 5.3 Definiti destructori pentru clasele Form si Rectangle cu mesaje specifice afisate in consola. Apelati destructorul pentru f si r. Observati ce se intimpla si care este ordinea de apelare a acestora.

Redefinirea functiilor membru în cadrul unei ierarhii de clase

Clasele Rectangle si Square mostenesc metoda showMessage din clasa de baza Form. In cadrul fiecarei clase, metoda showMessage are alta implementare. Compilatorul de C++ alege metoda membru apelata pe baza tipului obiectului apelant. Implementarea noua din clasa derivata nu înlocuieste implementarea

din clasa de baza, ci se adauga ei. Clasa derivata are la dispozitie si implementarea din clasa de baza, pe care o poate folosi utilizând operatorul de rezolutie ::.

Exemplul 5.3

Scrieti o clasa Form cu un membru variabila name si o metoda showMessage. Construiti clasele derivate Rectangle si Square si in fiecare dintre ele implementati o metoda showMessage. Instantiati pe rind fiecare dintre cele 3 clase, Form, Rectangle, Square si apelati metoda showMessage corespunzatoare.

Codul aferent Exemplului 5.3 este prezentat mai jos.

```
#include <iostream>

using namespace std;

class Form {
private:
   string color;

protected:
   string name;
   void setColor(string color)   {
      this->color = color;
   }
public:
   Form(string name){
      this->name = name;
   }
   string getName(){
      return this->name;
   }
   void setName(string name){
      this->name = name;
   }
   void showMessage(){
      cout<<"mesaj din Form"<<endl;
   }
};

class Rectangle:public Form {
protected:
   int width;
   int height;
public:
   Rectangle(string name, int width, int height) :Form(name){
      this->width = width;
      this->height = height;
   }
   void setWidth(int width){
      this->width = width;
   }
   int getWidth(){
      return this->width;
   }
   void showMessage(){
      Form::showMessage(); // operator :: folosit pentru a apela showMessage din clasa de baza
      cout<<"mesaj din Rectangle"<<endl;
   }
};

class Square:public Form {
protected:
   int side;
public:
   Square(string name, int side) :Form(name){
      this->side = side;
   }
   void setSide(int side){
      this->side = side;
   }
   int getSide(){
      return this->side;
   }
   void showMessage(){
      cout<<"mesaj din Square"<<endl;
```

```
    }
};

int main()
{
    Form *f = new Form("forma1");
    cout<<"forma: "<<f->getName()<<endl;
    f->showMessage();

    Rectangle *r = new Rectangle("rectangle 1", 1, 2);
    cout<<"rectangle name: "<<r->getName()<<" width:"<<r->getWidth()<<endl;
    r->showMessage();

    Square *s = new Square("square 1", 20);
    cout<<"square name: "<<s->getName()<<" side:"<<s->getSide()<<endl;
    s->showMessage();
    return 0;
}
```

Tema de casa

T.5.1 Modificati exemplul anterior si scrieti o metoda calculArie(), membru al clasei Form, care calculeaza aria figurii geometrice. Ce ar trebui sa returneze ? Pentru fiecare clasa extinsa, supraincarcati metoda calculArie.

T.5.2 Analog cu T.5.1, scrieti o metoda perimetru.

Reutilizarea claselor utilizand mostenirea multipla

O clasa poate mosteni mai multe clase numite clase de baza. Un exemplu este urmatorul:

```
class A {

//....

};

class B {

//....

};

class C: public A, public B {

//....

};
```

Clasa C mosteneste membrii de tip public si protected ai claselor A si B rezultând un obiect din clasa C ce contine doua sub-clase.

Exemplul 5.4

Definiti o clasa Persoana si una Sofer cu membrii variabile, specifici (nume, respectiv seriePermis). Fiecare clasa are cite o metoda, cu nume diferit, pentru afisarea informatiilor.

Codul aferent Exemplului 5.4 este prezentat mai jos.

```
#include <iostream>
```

```
using namespace std;

class Persoana {
protected:
    string nume;
public:
    Persoana(string nume){
        this->nume = nume;
    }
    void afisareInformatiiPersoana(){
        cout<<"din Persoana, nume:"<<this->nume<<endl;
    }
};

class Sofer {
protected:
    string seriePermis;
public:
    Sofer(string seriePermis){
        this->seriePermis = seriePermis;
    }
    void afisareInformatiiSofer(){
        cout<<"din Sofer seriePermis:"<<this->seriePermis<<endl;
    }
};

class Angajat: public Persoana, public Sofer { // mostenire multipla
protected:
    string numeAngajator;
public:
    // constructor care utilizeaza clasele mostenite si apeleaza constructorii lor
    Angajat(string name, string seriePermis, string numeAngajator): Persoana(name), Sofer(seriePermis){
        this->numeAngajator = numeAngajator;
    }
    void afisareInformatiiAngajat(){
        cout<<"din afisareInformatiiAngajat:"<<endl;
        Persoana::afisareInformatiiPersoana();      // apelare metoda din clasa de baza
        Sofer::afisareInformatiiSofer();            // apelare metoda din clasa de baza
        cout<<"nume angajator:"<<this->numeAngajator<<endl;
    }
};

int main()
{
    Persoana *persoana = new Persoana("Ionel");
    persoana->afisareInformatiiPersoana();
    Sofer *sofer = new Sofer("abc234 67");
    sofer->afisareInformatiiSofer();

    Angajat *angajat = new Angajat("Marian", "re 564hy8f", "Microsoft");
    cout<<"apeluri din obiectul angajat"<<endl;
    angajat->afisareInformatiiPersoana();       // apelare metoda din clasa de baza
    angajat->afisareInformatiiSofer();          // apelare metoda din clasa de baza
    angajat->afisareInformatiiAngajat();
    return 0;
}
```

Tema de casa

T.5.2 Scrieti o clasa Desert care gestioneaza un produs alimentar de acest tip. Extindeti clasa la Prajitura si TortAniversare. Definiti 1-3 membrii variabile specifici fiecarui tip de clasa. Definiti o metoda membru in Desert. Suprascrieti metoda respectiva in cele doua sub-clase, Prajitura si TortAniversare si folositi in ea membrii variabile din Prajitura si sub-clasa respectiva (Prajitura si TortAniversare).