RO: Moştenirea simplă şi multiplă

EN: Simple and multiple inheritance

Obiective:

- Înţelegerea teoretică a noţiunii de moştenire simplă în limbajul C++; implementarea practică a diferitelor tipuri de moştenire simplă;
- Utilizarea facilităţilor moştenirii multiple;

Objectives:

- Theoretical understanding of the simple C++ inheritance; practical implementation of different simple inheritance types;
- Using the multiple inheritance facilities;

Rezumat:

Moştenirea este un principiu al programării obiectuale care recomandă crearea de modele abstracte (clase de bază), care ulterior sunt concretizate (clase derivate) în funcţie de problema specifică pe care o avem de rezolvat.

Aceasta duce la crearea unei ierarhii de clase şi implicit la reutilizarea codului, la multe dintre probleme soluţia fiind doar o particularizare a unor soluţii deja existente.

Ideea principala în cadrul moştenirii este aceea că orice *clasă derivată* dintr-o *clasă de bază* "moşteneşte" toate atributele permise ale acesteia din urmă.

În procesul de moştenire, putem restricţiona accesul la componentele clasei de bază sau putem modifica specificatorii de vizibilitate ai membrilor clasei, din punctul de vedere al clasei derivate:

Clasa de bază	Moştenirea		
	private	public	protected
Membru „private"	inaccesibil	inaccesibil	inaccesibil
Membru „public"	private	public	protected
Membru „protected"	private	protected	protected

Moştenirea simplă se face după modelul:

class Nume_clasa_derivata : [specificatori_de_acces] Nume_clasa_baza
{
//corp clasa
}

În cazul *moştenirii multiple* o clasă poate să mostenească două sau mai multe clase de bază. Sintaxa generală de specificare a acestui tip de moştenire este următoarea:

class Nume_clasa_derivata : modificator_de_acces1 Nume_clasa1_baza, modificator_de_acces2 Nume_clasa2_baza [, modificator_de_acces Nume_clasaN_baza]{...};

Dacă o clasă este derivată din mai multe clase de bază, constructorii acestora sunt apelaţi în ordinea derivării, iar destructorii sunt apelaţi în ordinea inversă derivării.

Exemple:

```
//1. Propagarea membrilor protected in cazul mostenirii de tip public
//Baza_deriv.h

class Baza {
protected:
        int i, j;
public:
        void setI(int a) {
                i = a;
        }
        void setJ(int b) {
                j = b;
        }
```

```
	int getI( ) {
		return i;
	}
	int getJ( ) {
		return j;
	}
};//Baza class

class Derivata : public Baza {
public:
	int inmulteste( ) {
		return (i * j);	// corect, i si j raman protected
	}
};//Derivata class

#include <iostream>
using namespace std;
#include "Baza_deriv.h"

int main( ) {
	Derivata obiect_derivat;
cout << "\n Valori atribute (nedefinite): i, j: " << obiect_derivat.getI( ) << ", " <<
obiect_derivat.getJ( ) << endl;
	//obiect_derivat.i = 5;	// gresit, i  este protected nu public
	obiect_derivat.setI(5); //setI( ) e public din Baza
	obiect_derivat.setJ(17); // setJ( ) e public din Baza
cout << "\n Valori atribute (din Baza): i, j: " <<	obiect_derivat.getI( ) << ", " <<
obiect_derivat.getJ( ) << endl;
cout << "\n Produsul este: " << obiect_derivat.inmulteste( );//din Derivat
	} //main

//************************************************************************
//2. Exemplu de mostenire de tip "protected"
//Baza_deriv.h
class Baza {
	int x;
protected:
	int y;
public:
	int z;
	Baza(int x = 0, int y = 0) {
		this->x = x;
		this->y = y;
	}//Baza
	int getX( ) {
		return x;
	}
	void setX (int a) {
		 x=a;
	}
	int getY( ) {
		return y;
	}
	};//Baza class

class Derivata : protected Baza {
public:
	void do_this( ) {
		cout << "\n --------Clasa derivate, do_this( )------";
		//cout << "\n Valoarea variabilei private x: " << x << endl;//ramasa private
```

```
        cout << "\n Valoarea variabilei private x: "<< getX( )<< endl;//cu getX( ) public,0
        cout << "\n Valoarea variabilei protected y: " << y << endl;// protected, 0
        cout << "\n Valoarea variabilei  z: " << z << endl;//devenita protected, nedefinita
        setX(5); cout << "x= " << getX( ) << endl;//corect, getX/setX( ) devin protected
        y = 7; cout << "y= " << y << endl;//corect, y ramane protected
        z = 9; cout << "z= " << z << endl;// corect, z devine protected
        }//do_this
};//Derivata class

#include <iostream>
using namespace std;
#include "Baza_deriv.h"

int main( ) {
        int x, y;
        cout << "\n x= ";cin >> x;
        cout << "\n y= ";cin >> y;
        Baza ob1(x, y);
        Derivata ob2;//se apeleaza intai constructorul din clasa de baza cu x si y implicit =0
cout << "\n -Din clasa de baza-\nValoarea variabilei private x: " << ob1.getX( ) << "\nValoarea
variabilei protected y: " << ob1.getY( ) << endl;
//cout << "\n -Din clasa derivata-\n Valoarea variabilei private x: " << ob2.getX( )<< "\nValoarea
variabilei protected y: " << ob2.getY( ) << endl;//getX( ) si getY( ) au devenit protected la ob2
        ob2.do_this( );//e public in derivata si accesibil
}//main

//*************************************************************************
// 3. Exemplu de mostenire de tip „private”
//Baza_deriv.h
class Baza {
protected:  int a, b;
public:
        Baza( ) { a = 1, b = 1; }
        void setA(int a) {
                this->a = a;
        }
        void setB(int b) {
                this->b = b;
        }
        int getA( ) {
                return a;
        }
        int getB( ) {
                return b;
        }
        int aduna( ) {
                return a + b;
        }
        int scade( ) {
                return a - b;
        }
};

class Derivata : private Baza
{
public:
        int inmulteste() {
                return a * b;
        }
};
```

```
#include <iostream>
using namespace std;
#include "Baza_deriv.h"

int main( )
{
	Baza obiect_baza;
	cout << "\nAfis din baza (val. initiale): " << obiect_baza.getA( ) << " " <<
obiect_baza.getB( ) << '\n';
	cout << "\nSuma este (cu val. initiale, baza) = " << obiect_baza.aduna( ); // corect aduna( )
e public
	cout << "\nDiferenta este (cu val. initiale, baza) = " << obiect_baza.scade( );
 //corect scade( ) e public
	obiect_baza.setA(2);
	obiect_baza.setB(3);
cout << "\nAfis din baza (modificat): " << obiect_baza.getA( ) << " " << obiect_baza.getB( ) << '\n';
cout << "\nSuma/Diferenta dupa setare= " <<obiect_baza.aduna( ) << "/"<<
 obiect_baza.scade( )<<'\n';
	Derivata obiect_derivat;
cout << "\nProdusul este (din derivat cu val. initiale) = " << obiect_derivat.inmulteste( )<< '\n';
 // corect val. implicite
//cout << "\nSuma este (din derivat cu val. initiale, baza) = " << obiect_derivat.aduna( ); // incorect
aduna( ) devine private
	//obiect_derivat.scade( );  // eroare, scade( ) devine private
}

//*************************************************************************
//4. Mostenirea multipla
//Baza12_deriv.h
class Baza1 {
protected:
	int x;
public:
	int getX( ) {
		return x;}
	void arata( ) { cout << " Din Baza1: x = " << x << endl; }
};//Baza1 class
class Baza2 {
protected:
	int y;
public:
	int getY( ) {
		return y;}
	void arata( ) { cout << " Din Baza2: y = " << y << endl; }
};//Baza2 class

class Derivata : public Baza1, public Baza2 {
public:
	void setX(int i) { x = i; }
	void setY(int j) { y = j; }
	void arata( ) {
		cout << "\nDin Derivata: \n";
		Baza1::arata( );
		Baza2::arata( );
	}
}; //Derivata class

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

```
#include "Baza12_deriv.h"

int main( ) {
        Derivata obiect_derivat;
        obiect_derivat.setX(100); obiect_derivat.setY(200);
cout << "B1: valoarea lui x este: " << obiect_derivat.getX( ) << endl;//Baza1
cout << "B2: valoarea lui y este: " << obiect_derivat.getY( ) << endl;//Baza2
        obiect_derivat.arata( );//from Derivata class
        }//main
//*************************************************************************
//5. Constructori in mostenire
// Baza_deriv.h

class Base
{
protected:
        int m_no;
public:
        Base(int no = 0): m_no(no)
        { }
        int getM_no( ) const { return m_no; }
};
class Derived : public Base
{
protected:
        double m_cost;
public:
Derived(double cost = 0, int no = 0): Base(no), m_cost(cost) { }
//Call Base(int) constructor with value no
        double getM_Cost( ) const { return m_cost; }
        double multiplyNoCost( ) { return m_no * m_cost;}
};

#include <iostream>
#include "Baza_deriv.h"

int main( )
{
Derived derived(1.3, 15); // use Derived(double, int) constructor
std::cout << "No: " << derived.getM_no( ) << '\n';
std::cout << "Cost: " << derived.getM_Cost( ) << '\n';
std::cout << "Total_Cost: " << derived.multiplyNoCost( ) << '\n';
        return 0;
}

//6. Forme geometrice – referitor problema 6

    a) Varianta fara mostenire

//Shape_Circle.h
class Shape
{
        char name[dim];
        int r;
public:
        Shape(char* s, int l){
                strcpy(name, s);
                r = l;
        }
        char* getName( ){
```

```
                return name;
        }
        double areaCircle( )const {
                return (double)pi * r * r;
        }
        double perCircle( )const {
                return (double)2 * pi * r;
        }
};//Shape_Circle

//Shape_Circle_Square.h
class Shape
{
        char name[dim];
        int r;
        int s;
public:
        Shape(char* str, int l, int n){
                strcpy_s(name, str);
                r = l;
                s = n;
        }
        char* getName( ){
                return name;
        }
        double areaCircle( )const  {
                return (double)pi * r * r;
        }
        double perCircle( )const    {
                return (double)2 * pi * r;
        }
        double areaSquare( )const{
                return (double)s * s;
        }
        double perSquare( )const  {
                return 4 * s;
        }
};//Shape_Circle_Square

//...
//main()
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;

const int dim = 30;
const double pi = 3.14;

#include "Shape_Circle.h"
//#include "Shape_Circle_Square.h"

//#include "Shape_Circle_Square_Rectangle.h"
//...
int main( ) {
        int r;
        //int l;
//...
        char s[dim];
                cout << "\nRead the name of the shape: ";
                cin >> s;
```

```
                cout << "\nSpecify the Circle radius: ";
                cin >> r;
                //cout << "\nSpecify the Square side: ";
                //cin >> l;
                //Shape sh(s, r, l);
                Shape sh(s, r);

                cout << "\nShape Name: " << sh.getName( );
                cout << "\nCircle Perimeter: " << sh.perCircle( );
                cout << "\nCircle Area: " << sh.areaCircle( );

                //cout << "\nSquare Perimeter: " << sh.perSquare( );
                //cout << "\nSquare Area: " << sh.areaSquare( );
        }//main
```

b) *Varianta cu mostenire*

```
//Shape.h
class Shape
{
        char name[dim];
public:
        Shape(char* s)    {
                strcpy(name, s);
        }
        char* getName( ){
                return name;
        }
};
//Circle.h
class Circle :public Shape
{
        int r;
public:
        Circle(char* s, int l) :Shape(s), r(l)
        { }
        double area( )const {
                return pi * r * r;
        }
        double per( )const {
                return 2 * pi * r;
        }
};
//Square.h
class Square :public Shape
{
        int s;
public:
        Square(char* n, int l) :Shape(n), s(l)
        { }
        double area( )const {
                return (double)s * s;
        }
        double per( )const {
                return 4. * s;
        }
};
//Rectangle.h
class Rectangle :public Shape
{
```

```
        int w, h;
public:
        Rectangle(char* s, int m, int l) :Shape(s), w(m), h(l){ }
        double area( )const {
                return (double) w * h;
        }
        double per( )const {
                return 2. * (w + h);
        }
};

//main
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;

const int dim = 30;
const double pi = 3.14;

#include "Shape.h"
#include "Rectangle.h"
#include "Circle.h"
#include "Square.h"

int main( ) {
        int a, b, r, n;
        char s[dim];
        cout << "How many shapes do you want to process? ";
        cin >> n;
        for (int i = 0; i < n; i++) {
                cout << "\n\nRead the name of the shape number" << i + 1 << ": ";
                cin >> s;
                if (_stricmp(s, "circle") == 0) {
                        cout << "\nNow specify the radius: ";
                        cin >> r;
                        Circle c(s, r);
                        cout << "\nName: " << c.getName();
                        cout << "\nPerimeter: " << c.per();
                        cout << "\nArea: " << c.area();
                }
                else if (_stricmp(s, "square") == 0)
                {
                        cout << "\nNow specify the side: ";
                        cin >> r;
                        Square x(s, r);
                        cout << "\nName: " << x.getName();
                        cout << "\nPerimeter: " << x.per();
                        cout << "\nArea: " << x.area();
                }
                else if (_stricmp(s, "rectangle") == 0)
                {
                        cout << "\nSpecify the length: ";
                        cin >> a;
                        cout << "\nSpecify the width:";
                        cin >> b;
                        Rectangle d(s, a, b);
                        cout << "\nName: " << d.getName();
                        cout << "\nPerimeter: " << d.per();
                        cout << "\nArea: " << d.area();
                }
```

```
        else
                cout << "\nInvalid shape.";
    }
}//main
```

Teme:

1. Implementaţi programul prezentat în exemplul 3 şi examinaţi eventualele erori date la compilare daca exista prin eliminarea comentariilor. Modificaţi programul astfel încât să se poată accesa din funcţia *main( )*, prin intermediul obiectului *obiect_derivat*, şi metodele *aduna( )* şi *scade( )* din clasa de baza pastrand mostenirea de tip *private*.
2. Folosind modelul claselor de la mostenirea publica, implementaţi două clase, astfel:
   - clasa de bază contine metode pentru:
     -codarea unui şir de caractere (printr-un algoritm oarecare)
     => public;
     -afişarea şirului original şi a celui rezultat din transformare
     = > public;
   - clasa derivata contine o metoda pentru:
     -scrirea rezultatului codării într-un fişier, la sfârşitul acestuia.

   Fiecare inregistrare are forma: *nr_inregistrare: şir_codat*;
   Accesul la metodele ambelor clase se face prin intermediul unui obiect rezultat prin instanţierea clasei derivate. Programul care foloseşte clasele citeşte un şir de caractere de la tastatură şi apoi, în funcţie de opţiunea utilizatorului, afişează rezultatul codării sau îl scrie în fişier.
3. Să se implementeze o clasă de bază cu două atribute *protected* de tip întreg care conţine o metoda mutator pentru fiecare atribut al clasei, parametri metodelor fiind preluati in *main()* de la tastatura şi metode accesor pentru fiecare atribut care returneaza atributul specific. Să se scrie o a doua clasă, derivată din aceasta, care implementează operaţiile matematice elementare: +, -, *, / asupra atributelor din clasa de bază, rezultatele fiind returnate de metode. Să se scrie o a III-a clasă, derivată din cea de-a doua, care implementează în plus o metoda pentru extragerea rădăcinii pătrate dintr-un număr (*mul*, rezultat al operatiei * din prima clasa derivata) şi de ridicare la putere (atât *baza* (*plus*, rezultat al operatiei + din prima clasa derivata) cât şi *puterea (minus*, rezultat al operatiei - din prima clasa derivata*)* sunt trimişi ca parametri). Verificati apelul metodelor considerand obiecte la diferite ierarhii.
4. Definiţi o clasă numita *Triangle* care are 3 atribute protected pentru laturi si o metoda care calculează perimetrul unui triunghi ale cărui laturi sunt citite de la tastatura (folosite de un constructor adecvat) şi apoi o clasă derivata in mod public din *Triangle, Triangle_extended*, care în plus, calculează şi aria triunghiului. Folosind obiecte din cele doua clase apelati metodele specifice. Verificati inainte de instantiere posibilitatea definirii unui triunghi.
5. Adaugaţi în clasa derivată din exemplul anterior o metodă care calculează înalţimea triunghiului. Apelati metoda folosind un obiect adecvat.
6. Definiţi o clasă numită *Forme* care defineste o figura geometrica cu un *nume* ca si atribut de tip pointer la un sir de caractere. Clasa va contine un constructor fara parametrii, unul cu parametrii, copy constructor si se va supraincarca operatorul de asignare. Clasa va avea si o metoda getter si un destructor. Derivati in mod public o clasa *Cerc* care adauga un atribut de tip *int* pentru raza si constructori adecvati considerand si atributele (*nume*, *raza*) si o metoda getter pentru raza si alte metode care calculeaza aria si perimetrul cercului de raza *r*, valoare introdusa in *main( )* de la tastatura. Similar definiti o clasa *Patrat* si *Dreptunghi* care permit determinarea ariei si perimetrului obiectelor specifice. Instantiati obiecte din clasele derivate si afisati aria si perimetrul obiectelor. Datele specifice vor fi introduse de la tastatura. Definiti un obiect de tip *Cerc* cu parametrii care sa il copiati intr-un nou obiect la care sa ii afisati atributele. Definiti un obiect de tip *Patrat* cu parametrii si altul fara parametrii. Asignati celui fara parametrii obiectul instantiat cu parametrii si afisati atributele.
7. Consideraţi o clasa de baza *Cerc* definita printr-un atribut protected *raza*, care are un constructor cu parametrii si o metoda care determina aria cercului. Consideraţi o alta clasa de baza *Patrat* cu un atribut protected *latura* similar clasei *Cerc*. Derivaţi un mod public clasa *CercPatrat* care are un constructor ce apelează constructorii claselor de baza si o metoda care verifica daca pătratul de latura *l* poate fi inclus in cercul de raza *r*. De asemenea clasa derivata determina si perimetrul celor doua figuri geometrice. Instanţiaţi un obiect din clasa derivata (datele introduse de la tastatura), determinaţi aria si perimetrul cercului si al pătratului. Afişaţi daca pătratul cu latura introdusa poate fi inclus in cercul de raza specificat.

8. Considerați clasa *Fractie* care are doua atribute întregi protected *a* si *b* pentru numărător si numitor, doua metode de tip *set( )* respectiv *get( )* pentru fiecare din atributele clasei. Declarați o metoda publica *simplifica( )* care simplifica un obiect *Fractie*. Definiti un constructor explicit fara parametri care initializeaza *a* cu 0 si *b* cu 1, si un constructor explicit cu doi parametri care va putea fi apelat daca se verifica posibilitatea definirii unei fractii (*b*!=0). Supraîncărcați operatorii de adunare, scadere, inmultire si impartire (+,-,*,/) a fracțiilor folosind metode membre care si simplifica daca e cazul rezultatele obtinute, apeland metoda *simplifica( )* din clasa. Definiți o clasa *Fractie_ext* derivata *public* din *Fractie,* care va avea un constructor cu parametrii (ce apelează constructorul din clasa de baza). Supraîncărcați operatorii de incrementare si decrementare prefixați care aduna/scade valoarea *1* la un obiect de tip *Fractie_ext* cu metode membre.
   Instanțiați doua obiecte de tip *Fractie* fără parametrii. Setați atributele obiectelor cu date citite de la tastatura. Afișați atributele inițiale ale obiectelor si noile atribute definite. Efectuați operațiile implementate prin metodele membre, inițializând alte 4 obiecte cu rezultatele obținute. Simplificați si afișați rezultatele. Instanțiați doua obiecte de tip *Fractie_ext* cu date citite de la tastatura. Efectuați operațiile disponibile clasei, asignând rezultatele obținute la alte obiecte *Fracti_ext*. Simplificați si afișați rezultatele.

Homework:

1. Implement the program presented in the third example and examine the compilation errors if are by eliminating the existing comments? Modify the program so the object *obiect_derivat* will be able to access the *aduna( )* and *scade( )* methods, from the *main( )* function keeping the *private* inheritance.
2. Using the classes from public inheritance example, implement 2 classes with the following requests:
   - the base class has the methods for:
     - coding an array of characters (using a user-defined algorithm)
     => public;
     - displaying the original and the coded array
     = > public;
   - the derived class has a method for:
     - appending the coded array at the end of a previously created text file.

   Each record respects the format: *record_number: coded_array*;
   The methods located in both classes are accessed using an instance of the derived class. The program that uses the classes reads from the keyboard an array of characters and allows the user to choose whether the input will be coded or will be appended at the end of the text file.
3. Implement a class that has 2 protected integer variables, that contains a setter and getter methods for each attribute. Write a second class that inherits the first defined class and implements the elementary arithmetic operations (+, -, *, /) applied on the variables mentioned above the results being returned by methods. Write a third class derived from the second one that implements the methods for calculating the *square root* of a number (*mul* result obtained by the previous derived class) received as parameter, and for raising a numeric value to a certain power (the *base* (*plus*, result obtained by the previous derived class) and the *power* (*minus*, result obtained by the previous derived class) are sent to the method as parameters). Verify the methods's calling using objects at different hierchies levels.
4. Define a class called *Triangle* with 3 attributes for the triangle sides that has a method that calculates the perimeter of the triangle with the sides introduced from the KB. Another class, *Triangle_extended*, is derived in public mode from *Triangle* and defines a method for calculating the triangle's area. Using objects from both classes call the allowed methods. Verify before to instantiate the objects the possibility to define a *Triangle* object.
5. Extend the second class with a method that can compute the triangle's height. Call the method using an adequate object.
6. Define a class *Shape* that defines a shape with a *name* attribute as a pointer to character string. The class will contain a constructor without parameters, one with parameters, copy constructor and the assign operator will be overloaded. The class will also have a getter method and a destructor. Derive in public mode a *Circle* class that adds an *int* type attribute to the *radius* and appropriate constructors considering also the attributes (*name, radius*) and a getter method for the radius and other methods that calculate the area and perimeter of the circle of radius *r*, value entered in main () on the keyboard.

In the same mode define other classes (*Square, Rectangle*, etc.) Instantiate objects from the derived classes and display the area and the perimeter. The data will be introduced from the KB. Define a *Circle* object with parameters introduced from the KB, to copy to a new object and display its attributes. Define a Square object with parameters and another without parameters. Assign the instantiated object with the parameters to the one without parameters and display the attributes.

7. Consider a base class *Circle* defined by a protected attribute *radius*, that contains a constructor with parameters and a method that will determine the area of the circle. Consider other base class, *Square* with a protected attribute, *length*, similar to *Circle* class. Derive in public mode the class *RoundSquare* from both classes that will contain a constructor that will call the constructors from base classes and a method that will verify if the square of length *l* may be included in the circle of radius *r*. The derived class will also determine the perimeter of both shapes. Instantiate an object from the derived class (data from the KB) and determine the area and perimeter of the composed shapes. Display a message if the square may be included in the circle.
8. Consider the *Fraction* class that has two protected attributes *a* and *b* for the nominator and denominator and two corresponding setter and getter methods for all attributes. Declare a public method named *simplify( )* that simplifies a fraction. Define an explicit constructor without parameters that initializes *a* with 0 and *b* with 1 and another explicit constructor with two integer parameters. For this constructor is verified if b!=0 before to be called. Overload the addition, subtraction, multiplication and division operators (+, -, *, /) using member methods that simplify (if necessary) the obtained results. Define a class named *Fraction_ext* that inherits in a public mode the *Fraction* class and has a parameterized constructor that calls the constructor from the base class. Use member methods for overloading the pre-incrementation and pre-decrementation operators that will add/subtract 1 to the value of a *Fraction_ext* instance. Instantiate two *Fraction* objects without parameters. Set the attributes using values read from the keyboard. Perform the implemented operations and initialize other four objects with the obtained results. Simplify the results. Instantiate two objects of *Fraction_ext* type with data from the KB. Perform the available operations. Assign the operation results to other existing *Fraction_ext* objects. Simplify and display the obtained results.