# Lab. 2 - Elemente privind Programarea Orientata Obiect in limbajul C/C++ - Elements of OOP in C/C++ language

## Clase, obiecte, membri, accesul la membri.

**Obiective:**

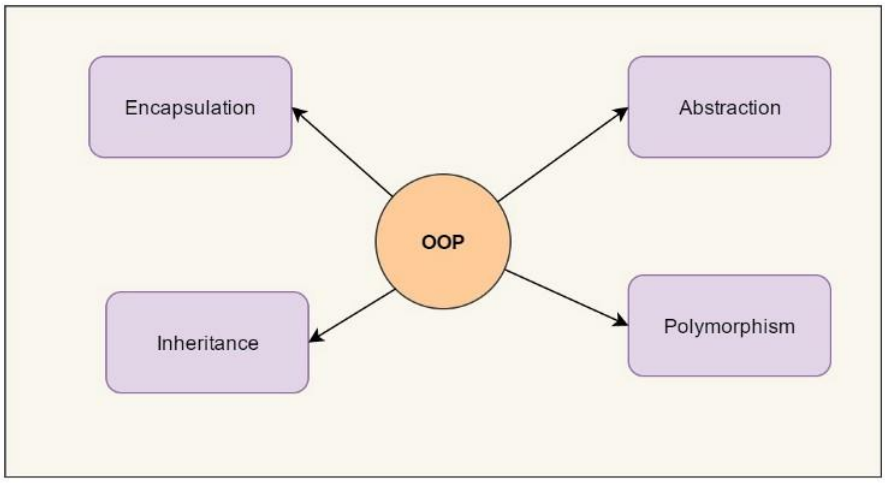
- Înţelegerea teoretică şi practică a noţiunilor de clasă, obiect, membru al unei clase, accesul la membrii unei clase, constructori, destructori.
- Scrierea de programe simple, după modelul programării obiectuale, care exemplifică noţiunile menţionate mai sus.

**Rezumat:**

Programarea orientată pe obiecte implică definirea *tipurilor abstracte de date* în vederea definirii obiectelor, a mesajelor precum şi a mecanismului de moştenire. Noţiunile de bază ale POO (Programarii Orientate Obiect) se referă în principal la *definirea claselor de obiecte*, a *moştenirii* şi a *polimorfismului*.

Cele patru principii (Piloni) de bază ale programării orientate pe obiecte sunt:
-Incapsulare,
-Abstractizare a datelor,
-Polimorfism,
-Moştenire, notiuni ce vor fi prezentate in continuare.

Clasele C++ reprezintă tipuri noi de date, care conţin metode şi variabile, un fel de „matriţe" ce sunt folosite pentru a defini metodele şi variabilele unui obiect anume, „turnat" după modelul clasei. Utilizarea claselor in POO reprezinta un mecanism de **abstractizare a datelor** cunoscut ca ADT (Abstract Data Type). ADT nu reprezinta inca POO, fiind necesar a considera polimorfismul si mostenirea.




Four Pillars of Object Oriented Programming

Odata cu evolutia limbajelor orientate obiect, tratarea exceptiilor devine un principiu nou adaugat ca al 5-lea. **Excepțiile** sunt acceptate în toate limbajele moderne orientate obiect și sunt mecanismul principal de gestionare a erorilor și a situațiilor neobișnuite în programarea orientată obiect.

Procesul de creare al unui obiect se numeşte *instanţiere* caz în care pentru fiecare variabilă se vor preciza nişte valori concrete (explicit sau implicit).

Metodele unei clase pot să fie definite în interiorul ei (asa numite metode inline) sau în afară, caz în care este necesară indicarea clasei prin operatorul rezoluţie sau scop ( :: ).

Accesul la variabilele și metodele membre ale unei clase se poate controla prin utilizarea *specificatorilor de vizibilitate*: ***public, private, protected***. Acest mecanism defineste notiunea de **incapsulare**.

Funcţiile *constructor* sunt funcţii care sunt apelate în momentul instanţierii unor obiecte, ele nu au specificat nici un fel de tip returnat, au numele identic cu cel al clasei din care fac parte. Rolul contructorilor este de a iniţializa variabilele din acea clasă şi de a aloca spaţiu de memorie corespunzător obiectului instanţiat şi a variabilelor folosite în cadrul clasei.

*Destructorul* clasei este o metodă care, analog cu constructorul, poate fi recunoscută prin refaptul că are acelaşi nume cu clasa din care face parte, este precedat de caracterul ~ şi are rolul de a elibera spaţiul de memorie alocat pentru obiectul instanţiat şi a variabilelor folosite în cadrul clasei.

## Constructori. Destructori. Tablouri de obiecte.

**Obiective:**

- Înţelegerea teoretică şi practică a noţiunilor de constructori, destructor, tablouri de obiecte.
- Scrierea de programe simple, după modelul programării prin abstractizare a datelor, care exemplifică noţiunile menţionate mai sus.

**Objectives:**

- Understanding the constructors, destructor and object arrays.
- Writing some simple OOP programs that use the notions mentioned above.

**Rezumat:**

Constructorii pot fi:

- impliciţi
- expliciţi
    - vizi: fară nici un argument
    - cu parametri: iniţializează anumite variabile din clasă şi/sau efectuează anumite operaţii conform variabilelor primite ca argumente.

Un tip special de constructor este *constructorul de copiere*, a cărui menire este de a crea copia unui obiect. Constructorii de copiere sunt obligatorii doar în cazul în care clasa respectivă conține atribute de tip pointeri ce necesită alocare dinamică pentru a se rezerva spaţiu. Mai sunt si constructori speciali de *conversie*.

Destructorul :

- poate fi definit: dacă se dorește efectuarea anumitor operaţii în momentul distrugerii obiectului. In cazul atributelor de tip pointeri alocați dinamic, se definește un destructor care realizează eliberarea acestor pointeri.
- poate să lipsească: se apelează un destructor implicit de către compilator.

Pointerul *this* pointează spre membrii instanţei curente a clasei (este apelabil doar din interiorul clasei). Este util mai ales când în interiorul unei metode dintr-o clasă este necesar să facem distincţie între variabile ce aparţin unor obiecte diferite, pointerul *this* indicând întotdeauna obiectul curent.

Pentru a crea tablouri de obiecte avem nevoie de un constructor fără parametrii (implicit vid sau unul explicit echivalent). Dacă se foloseşte un constructor cu parametri, fiecare obiect din tablou poate fi iniţializat indicând o listă de iniţializare.

**Exemple:**

```
//1. Implementarea clasei Rectangle pentru efectuarea
//unor operatii elementare cu forme geometrice dreptunghiulare
// Rectangle.h - clasa Rectangle
class Rectangle {
        // membri private
        int height;
        int width;
public:
        // membri publici
        Rectangle(int h = 10, int w = 10); // constructor explicit cu val. implicite
        int det_area( );
        void setHeight(int);
        int getHeight( ) { return height; }
        void setWidth(int);
```

```
        int getWidth( ) { return width; }
        ~Rectangle( ); // destructor explicit
};

Rectangle::Rectangle(int h, int w)     // definire constructor explicit
{
        height = h;
        width = w;
}
Rectangle::~Rectangle( )    // destructor
{
        cout << "\nApel destructor...";
        height = 0;
        width = 0;
}

int Rectangle::det_area( ) {
        return height * width;
}

void Rectangle::setHeight(int init_height)
{
        height = init_height;
}
void Rectangle::setWidth(int init_width)
{
        width = init_width;
}

//main()
#include <iostream>
using namespace std;
#include "Rectangle.h"

int main( )
{
        int i;
        cout << "\n\nTablou de obiecte initializat la declarare\n";
        Rectangle group1[4] = {
                Rectangle(),                    //echivalent Rectangle(10,10),
                Rectangle(20),//echivalent Rectangle(20,10)
                Rectangle(30),//echivalent Rectangle(30,10)
                Rectangle(40)//echivalent Rectangle(40,10)
        };

        for (i = 0; i < 4; i++)
                cout << "\nAria dreptunghiului: " << group1[i].det_area() <<
                " cu laturile, width= "<< group1[i].getWidth( ) <<" si height= "<<
group1[i].getHeight( ) <<endl;
        cout << "\n.....\n\n";

        // tablou de obiecte
        Rectangle group2[4];
        cout << "\nTablou de obiecte asignat cu metode set\n";
        for (i = 1; i < 4; i++) {
                group2[i].setHeight(i + 10);
                //group2[i].setWidth(10);//nu e necesar
        }
        for (i = 0; i < 4; i++)
                cout << "\nAria dreptunghiului: " << group2[i].det_area() <<
```

```
                    " cu laturile, width= " << group2[i].getWidth( ) << " si height= " <<
group2[i].getHeight( )<<endl;
        cout << "\n.....\n\n";

        // tablou dinamic
        Rectangle *group3 = new Rectangle[4];
        cout << "\nTablou dinamic de obiecte asignat cu metode set\n";
        for (i = 1; i < 4; i++) {
                (group3 + i)->setHeight(i + 10);
                //(group3 + i)->setWidth(10);//nu e necesar
        }
        for (i = 0; i < 4; i++)
                cout << "\nAria dreptunghiului: " << group3[i].det_area() <<
                " cu laturile, width= " << group3[i].getWidth( ) << " si height= " <<
group3[i].getHeight( )<<endl;

        delete[ ]group3;
        cout << "\n.....\n\n";
        // tablou dinamic
        Rectangle *group4 = new Rectangle[4];
        cout << "\nTablou de obiecte asignat cu constructor cu parametri\n";
        group4[0] = Rectangle(5, 10);
        group4[1] = Rectangle(15, 20);
        group4[2] = Rectangle(25, 30);
        group4[3] = Rectangle(35, 40);

        for (i = 0; i < 4; i++)
        cout << "\nAria dreptunghiului: " << (group4 + i)->det_area() <<
        " cu laturile, width= " << (group4+i)->getWidth( ) << " si height= " << (group4 + i) ->
getHeight( )<<endl;

        delete [ ]group4;
        cout << "\n.....\n\n";
}//main

//2.  Implementarea unei clase numita Stiva pentru simularea lucrului cu stiva
```

**Stiva.h**

```
class Stiva{
//membri privati
private:
        int Dim;
        char *Stack;
        int Next;
//membri publici
public:
        Stiva( );
        Stiva(int);
        Stiva(const Stiva &);
        ~Stiva( );

        int Push(char c);
        int Pop(char &c);
        int IsEmpty(void);
        int IsFull(void);
};

// constructori
Stiva :: Stiva( ){
        Next = -1;
```

```
        Dim = 256;
        Stack = new char [Dim];
}

Stiva :: Stiva(int dim_i){
        Next = -1;
        Dim = dim_i;
        Stack = new char [Dim];
}
//constructor de copiere
Stiva :: Stiva(const Stiva &inStack){
        Next = inStack.Next;
        Dim = inStack.Dim;
        Stack = new char[Dim];
        for(int i=0; i<Next;i++)
                Stack[i] = inStack.Stack[i];
}

// destructor
Stiva :: ~Stiva( ){
        delete [ ] Stack;
}

// test stiva goala
int Stiva :: IsEmpty( ){
        if (Next < 0)
                return 1;
        else
                return 0;
}
// test stiva plina

int Stiva::IsFull( ){
        if(Next >= Dim)
                return 1;
        else
                return 0;
}

// introducere in stiva
int Stiva::Push(char c){
        if(IsFull( ))
                return 0;
        *Stack++ = c;
        Next++;
        return 1;
}

// extragere din stiva
int Stiva::Pop(char &c){
        if(IsEmpty( ))
                return 0;
        c = *(--Stack);
        Next--;
        return 1;
}

// program de test
#include <iostream>
using namespace std;
```

```
#include "Stiva.h"

int main( ) {
	unsigned int i;
	char buf[64];
	strcpy_s(buf, "Bafta in sesiune !");
	cout << endl << "Sir initial: " << buf << endl;

	Stiva Mesaj;
	for (i = 0; i < (strlen(buf)); i++)
		Mesaj.Push(buf[i]);
	i = 0;
	while (!Mesaj.IsEmpty( ))
		Mesaj.Pop(buf[i++]);
	cout << endl <<" Push-> Pop -> Inversare: "<< buf << endl;
	// constructor de copiere -init
	Stiva Mesaj1(Mesaj);
	i = 0;
	while (!Mesaj1.IsEmpty( ))
		Mesaj1.Pop(buf[i++]);
	cout << endl << "Copie obiect precedent: "<<buf << endl;

	char sTest[15] = "Sir_de_test";
	cout << endl << "Sir de test: "<< sTest << endl;
	Stiva Mesaj2(strlen(sTest)+1);//al 2-lea constructor

	// constructor de copiere -init cu assign
	Stiva Mesaj3 = Mesaj2;
	i = 0;
	while (!Mesaj3.IsEmpty( ))
		Mesaj3.Pop(sTest[i++]);
	cout << endl << "Copie sir initial de test extras cu Pop: " <<sTest << endl;
}//main

//*************************************************************************************
// 3. Exemplu de utilizare a constructorului de copiere pentru un punct caruia i se asociaza un text
```

**CPunctText.h**

```
const int dim_sir = 20;

class CPunctText {
	int x;
	int y;
	int lungime_sir;
	char *sNume;
public:
	//constructor explicit vid
	CPunctText( );
	//constructor cu parametri
	CPunctText(int ix, int iy, const char *sText = "Punct");
	//constructor de copiere
	CPunctText(const CPunctText &pct);
	//destructor:
	~CPunctText( );
	void afis() {
		cout << "\nObiectul are x= " << x;
		cout << "\nObiectul are y= " << y;
		cout << "\nObiectul are sirul =  " << sNume;
	}//afis
};
```

```
CPunctText::CPunctText( ) {
        cout << "\n constructor explicit vid";
        lungime_sir = dim_sir;
        sNume = new char[lungime_sir];
}

CPunctText::CPunctText(int ix, int iy, const  char *sText) {
        cout << "\n constructor cu parametri";
        lungime_sir = strlen(sText) + 1;
        sNume = new char[lungime_sir];
        x = ix;
        y = iy;
        strcpy(sNume, sText);
}

CPunctText::CPunctText(const CPunctText &pct) {
        cout << "\n constructor de copiere";
        sNume = new char[pct.lungime_sir];
        x = pct.x;
        y = pct.y;
        lungime_sir = pct.lungime_sir;
        strcpy(sNume, pct.sNume);
}

CPunctText::~CPunctText() {
        cout << "\n destructor";
        delete[ ] sNume;
}

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

#include "CPunctText.h"

int main( ) {
        CPunctText cpt1(1, 2, "Punct1");//apel constructor cu parametri
        CPunctText cpt2(cpt1);       //apel constructor de copiere
        CPunctText cpt3 = cpt2;      //apel constructor de copiere
        CPunctText cpt4(4, 5);       //apel constructor cu parametri

        cpt3.afis();
        cpt4.afis();
        }//main
```

**Teme:**

1. Modificaţi exemplul 3 astfel încât să permită obtinerea unui nou punct, avand coordonatele obtinute prin adunarea coordonatelor a două astfel de puncte. Numele noului punct va fi rezultat prin concatenarea numelor celor două puncte. Adaugati si testati o metoda care calculeaza distanta de la un punct la origine. Modificați clasa astfel încât sa eliminați metoda *Afis()* folosind in schimb metode getter adecvate. Eliminați de asemenea atributul *lungime_sir* modificând adecvat metodele clasei. Testati utilizand si functii specifice sirurilor de caractere din VC++1y/2z (*strcy_s( ) si strcat_s( )*).
2. Să se scrie o aplicaţie C/C++ care să modeleze obiectual un tablou unidimensional de numere reale. Creaţi două instanţe ale clasei şi afișați valorile unui al 3-lea tablou, obținute prin scăderea elementelelor corespunzătoare din primele 2 tablouri. Dacă tablourile au lungimi diferite, tabloul rezultat va avea lungimea tabloului cel mai scurt.

3. Să se scrie o aplicație în care se modelează clasa *Student* cu *nume, prenume* și *notele* din sesiunea din iarnă. Să se afișeze numele studenților din grupă care au restanțe și apoi numele primilor 3 studenți din grupă în ordinea mediilor.
4. Să se scrie o aplicație C/C++ în care se citește de la tastatură un punct prin coordonatele *x, y, z*. Să se scrie o metodă prin care să se facă translația punctului cu o anumită distanță pe fiecare dintre cele trei axe. Să se verifice dacă dreapta care unește primul punct și cel rezultat în urma translației trec printr-un al treilea punct dat de la consolă.
5. Modelați clasa *Student* care să conțină atributele private *nume, prenume, note (tablou 7 valori int), grupa*. Alocați dinamic memorie pentru *n* studenți. Calculați media cu o metoda din clasa și sortați studenții după medie, afisând datele fiecărui student (*nume, prenume, grupa, medie*). Implementati si destructorul clasei care să afișeze un mesaj.
6. Definiți o clasă *Complex* modelată prin atributele de tip double *real, imag* și un pointer de tip *char* către numele fiecărui număr complex. În cadrul clasei definiți un constructor explicit cu doi parametri care au implicit valoarea 1.0 și care alocă spațiu pentru *nume* un șir de maxim 7 caractere, de exemplu *"c1"*. De asemenea, definiți un constructor de copiere pentru clasa *Complex.* Clasa va mai conține metode mutator/setter și accesor/getter pentru fiecare membru al clasei, metode care permit operațiile de bază cu numere complexe și un destructor explicit. Definiți 10 numere complexe într-un tablou. Calculați suma numerelor complexe din tablou, valoare ce va fi folosită pentru a inițializa un nou număr complex, cu numele *"Suma_c"*. Realizați aceleași acțiuni făcând diferența și produsul numerelor complexe.
7. Consideram clasa *Fractie* care are doua atribute intregi *private a* si *b* pentru numarator si numitor, doua metode de tip *set( )* respectiv *get( )* pentru atributele clasei publice si o metoda *simplifica( )* publica care simplifica obiectul curent *Fractie* de apel, returnând un alt obiect simplificat. Metoda *simplifica( )* va apela o metoda *private cmmdc( )* pentru simplificarea fracției. Definiți un constructor explicit fara parametri care initializeaza *a* cu 0 si *b* cu 1, si un constructor explicit cu doi parametri care va fi apelat dupa ce s-a verificat posibilitatea definirii unei fractii (*b*!=0). Definiți o metoda *aduna_fractie( )* care are ca si parametru un obiect de tip *Fractie* si returneaza *suma* obiectului curent de apel cu cel dat ca si parametru, ca si un alt obiect de tip *Fractie*. Analog definiti metode pentru *scadere, inmultire* si *impartire*. Instantiati doua obiecte de tip *Fractie* cu date citite de la tastatura. Afisati atributele initiale si cele obtinute dupa apelul metodei *simplifica( )*. Efectuati operatiile implementate prin metodele clasei si afisati rezultatele.
8. Considerand problema precedenta adaugati in clasa *Fractie* un atribut pointer la un sir de caractere, *nume* care va identifica numele unei fractii. Constructorul fara parametri va aloca dinamic un sir de maxim 20 de caractere, cel cu parametrii va contine un parametru suplimentar cu numele implicit, *"Necunoscut"* care va fi copiat in zona rezervata ce va fi de doua ori dimensiunea sirului implicit. Pentru acest atribut se vor crea metode accesor si mutator, care sa afiseze numele unui obiect de tip *Fractie* respectiv care sa poata modifica numele cu un nume specificat (*Sectiune_de_aur, Numar_de_aur*, etc.). De asemenea se va implementa si un copy constructor si un destructor. In programul principal instantiati doua obiecte de tip *Fractie*, unul folosind constructorul fara parametri, celalalt folosind constructorul cu parametri, valorile parametrilor fiind introduse de la tastatura. Modificati atributele primului obiect, folosind metode de tip *setter*. Initializati un al treilea obiect de tip *Fractie* folosind copy constructorul. Afisati atributele acestui obiect obtinut folosind metodele de tip *getter*.

**Homework:**

1. Modify example 3 in order to allow the addition of two *CPunctText* points. The name of the new point will be created from the names of the compounding points. Add a method that returns the distance from a point to origin. Modify the class so that you remove the *Afis ()* method by using appropriate getter methods instead. Also remove the *lungime_sir* attribute by appropriately modifying the class methods. Test using the string specific functions of VC ++ 1y/2z (*strcy_s ( )* and *strcat_s ( )*).
2. Write a C/C++ application that models in OOP a real numbers one dimensional array. Instantiate two objects of this class with the same length *n* and store in a third one the results of subtracting each of the two real number arrays' elements. If the source arrays have different lengths, the result has the length of the shortest array.
3. Model in OOP a class named *Student* containing *name, surname* and the *marks* from the winter session exams. Display the name of the students who have arears exams and the first three students in the group.

4. Write a C/C++ application that reads a point from the keyboard by giving the *x, y* and *z* coordinates. Write a method that moves the point with a given distance on each of the three axes. Verify if the line between the first and the second position of the point crosses a third given point.
5. Create a class named *Student* that has as private attributes the *name*, *surname*, some *marks (array 7 int values)*, the *group*. Allocate the necessary amount of memory for storing *n* students.. Determine the average mark with a method from the class for each student and use it for sorting the students. Display the ordered array (*name, surname, group, average_mark*). The destructor will display a message.
6. Define a class called *Complex* that stores the double variables *real*, *imag* and a pointer of character type that holds the name of the complex number. Define an explicit constructor with 2 parameters that have 1.0 as implicit value. The constructor also initializes the pointer with a 7 characters wide memory zone. Define a copy constructor for this class. Implement the mutator/setter and accessor/getter methods for each variable stored inside the class. All the operations related to the complex numbers are also emulated using some specific methods. An explicit destructor method is also part of the class. Define an array of 10 complex numbers. Determine the sum of all the numbers in this array and use this value for initializing a new instance of the class named *complex_sum*. Repeat this action for all the rest of the operations implemented inside the class.
7. Consider a class named *Fraction* that has two private integer attributes *a* and *b* for the denominator and nominator, two *set( )* and *get( )* methods and a method *simplify( )* that will simplify the current calling *Fraction* object and will return as result a *Fraction* object. *simplify( )* method will call a private *cmmdc( )* method to simplify the fraction. Define an explicit constructor without parameters that initializes *a* with 0 and *b* with 1. Define another explicit constructor that receives 2 integer parameters. For this constructor is verified if *b*!=0 before to be called. Define a method named *add_fraction( )* that returns the object obtained by *adding* the current object with the one received as parameter, as a *Fraction* object. Define in the same manner the methods that *subtract*, *multiply* and *divide* two fractions. Instantiate two *Fraction* objects having the corresponding data read from the keyboard. Display the initial attributes and the ones obtained after simplifying the fractions. Call the methods that apply the implemented arithmetical operations and display the results.
8. Considering the previous task add in the *Fraction* class another attribute consisting in a character array pointer (*name*) that identifies a fraction. The constructor without parameters will allocate a max 20 characters memory zone, the parameterized constructor will have another implicit parameter initialized with "*Unknown*" that will represent the fraction's name and the reserved space will be twice the string dimension. Implement setter and getter methods for the *name* attribute. Implement a copy constructor and a destructor. In the *main( )* function create two *Fraction* objects, one using the constructor without parameters and the other using the parameterized constructor. Modify the attributes of the first object using *setter* methods. Create a third object using the copy constructor. Display the attributes of this last object using the getter methods.

## Moştenirea simplă şi multiplă

**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* "moştenește" 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( ) {

	}
};

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 14
```

a) *Varianta fara mostenire*

```
//Shape_Circle.h
class Shape
{
        char name[DIM];
        int r;
public:
        Shape(char* s, int l){
                strcpy_s(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()
#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_s(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 (double)pi * r * r;
	}
	double per( )const {
		return (double)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 L, int l) :Shape(s), w(L), h(l)
	{ }
	double area( )const {
		return w * h;
	}
	double per( ) {
		return 2 * (w + h);
	}
};
//main
#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:**

9. 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*.
10. 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.

11. 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 memorate in doua atribute protected *int* (*plus, minus*) si in alte doua atribute protected *double* (*mul, div*) declarate in clasa. 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.
12. 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.
13. Adaugaţi în clasa derivată din exemplul anterior o metodă care calculează înalţimea triunghiului. Apelati metoda folosind un obiect adecvat.
14. Definiţi o clasă numită *Forme* care defineste o figura geometrica cu un *nume* ca si atribut de tip pointer la un sir de caractere. Derivati in mod public o clasa *Cerc* cu un constructor adecvat (*nume, raza*) si 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.
15. 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.
16. Consideraţi 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. Declaraţi 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). Supraîncărcaţi operatorii de adunare, scadere, inmultire si impartire (+,-,*,/) a fracţiilor folosind metode membre care si simplifica daca e cazul rezultatele obtinute, apeland metoda *simplifica( )* din clasa. Definiţi o clasa *Fractie_ext* derivata *public* din *Fractie*, care va avea un constructor cu parametrii (ce apelează constructorul din clasa de baza). Supraîncărcaţi operatorii de incrementare si decrementare prefixaţi care aduna/scade valoarea *1* la un obiect de tip *Fractie_ext* cu metode membre.

    Instanţiaţi doua obiecte de tip *Fractie* fără parametrii. Setaţi atributele obiectelor cu date citite de la tastatura. Afişaţi atributele iniţiale ale obiectelor si noile atribute definite. Efectuaţi operaţiile implementate prin metodele membre, iniţializând alte 4 obiecte cu rezultatele obţinute. Simplificaţi si afişaţi rezultatele. Instanţiaţi doua obiecte de tip *Fractie_ext* cu date citite de la tastatura. Efectuaţi operaţiile disponibile clasei, asignând rezultatele obţinute la alte obiecte *Fracti_ext*. Simplificaţi si afişaţi rezultatele.

**Homework:**

9. 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.
10. 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.

11. 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 stored in protect class attributes (*plus*, *minus* of *int* type, *mul*, *div* of *double* type). 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.
12. 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.
13. Extend the second class with a method that can compute the triangle's height. Call the method using an adequate object.
14. Define a class *Shape* that defines a shape with a *name* attribute as a pointer to character string. Derive in public mode a *Circle* class with an adequate constructor (*name*, *radius*) and methods that calculates the area and the perimeter of the *Circle* with the radius *r* introduced from the KB in the *main( )* function. 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.
15. 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.
16. 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.

# Clase şi metode virtuale. Clase abstracte.

**Obiective:**

- Înţelegerea moştenirii virtuale, a modalităţii de declarare, definire şi utilizare a metodelor virtuale şi a claselor abstracte în limbajul C++
- Upcasting/downcasting in C++

**Objectives:**

- The understanding of C++ virtual inheritance, of virtual classes and methods' declaration, definition and usage, of abstract classes;
- Upcasting/downcasting in C++

**Rezumat:**

*Moştenirea virtuală* are loc atunci când mai multe clase moştenesc *virtual* o clasă de bază comună. Ea constă în a crea o nouă instanţă (virtuală) a clasei părinte în fiecare dintre clasele copil (derivate), independentă de celelalte instanţe generate în clasele "paralele". Obiectele din clasa de baza vor fi accesate de o alta clasa derivata prin mostenire multipla din clasele derivate virtual din clasa de baza folosind un singur obiect de acest tip din clasa de baza.

O *metodă virtuală* este acea metodă care este definită cu specificatorul *virtual* în clasa de bază şi apoi este redefinită în clasele derivate. **Comportamentul specific apare atunci când sunt apelate printr-un pointer.** Un pointer al clasei de bază poate fi folosit pentru a indica spre orice clasă derivată din aceasta. Când un astfel de pointer indică spre un obiect derivat ce conţine o metodă virtuală redefinită, compilatorul C++ determină care versiune a metodei va fi apelată, în funcţie de tipul obiectului spre care indică acel pointer, catre un obiect din clasa derivata, sau catre un obiect din clasa de baza. Varianta cu care se va lucra se stabileşte la apel, motiv pentru care se foloseşte denumirea de "legatură dinamică" (dynamic binding) spre deosebire de "legătura statică" în care varianta cu care se lucrează este stabilită în etapa de compilare. Astfel se pot executa versiuni diferite ale metodei virtuale în funcţie de tipul obiectului (o formă de polimorfism dinamic) referit de pointer.

*C++1y/2z* recomanda utilizarea specificatorului *override* dupa numele metodei care este redefinita in clasa derivate pentru o verificare a semnaturii metodei de catre compilator.

O *clasă abstractă* este o clasă care are cel puţin o *metodă virtuală pură* (adică, metoda are o implementare vidă). Aceste clase nu sunt utilizate direct, ci furnizează un schelet pentru alte clase ce vor fi derivate din acestea. De obicei, toate metodele membre ale unei clase abstracte sunt virtuale şi au implementări vide, urmând să fie redefinite în clasele derivate. O clasă abstractă nu poate fi instanţiată (nu se pot declara obiecte având acest tip), în schimb se pot declara pointeri la o clasă abstractă.

Metodele statice, constructorii nu pot fi virtuali, dar destructorii pot fi virtuali, caz în care se garantează distrugerea şi a obiectului din clasa derivată.

**Upcasting/downcasting**

Un pointer către o clasă de bază este compatibil ca și tip cu un pointer la clasa ori care clasa derivată din ea; pointerii către clasa de bază pot fi folosiți pentru a accesa membri și din clasele derivate (moșteniți din clasa de bază – proces numit *upcasting*).

Pointerul clasei de bază poate fi utilizat cu obiecte din clasa de bază, în acest caz metodele clasei de bază sunt apelate.

La upcasting, obiectul derivat nu se schimbă. Prin *upcast* pointerul de tipul clasei de baza este asociat catre un obiect din clasa derivată, dar se pot accesa numai metodele și datele membre care sunt definite în clasa de bază. Doar **metodele virtuale** sunt supuse **legării dinamice**.

Un obiect derivat poate fi asignat unui obiect din clasa de bază, fără a specifica ceva suplimentar, proces numit de asemenea *upcasting.*

Un pointer către o clasă derivată poate fi asociat către un obiect din clasa de bază folosind un cast explicit cu un pointer către clasa derivată, proces numit *downcasting*:

*Pozitie pp0(7,7);//base class object*
*Pozitie *p=new Punct(100,100,'Z');//base class pointer obtained from a derived class with upcasting*

```
...
cout<<"\nDowncasting:\n ";
Punct *pdown;//derived pointer
pdown=(Punct*)&pp0;//base class object
pdown->afisare( );//base class method if pp0 refers to base class
pdown = (Punct*)p;//downcasting by derived class object
pdown->afisare( );//derived class method if p obtained by derived class Punct
```

**Exemplul 1:**

```
//Exemplificare upcasting si metode virtuale
/*Header.h*/
const int dim = 20;
class Student {
protected:
   char nume[dim];
   int anul;
public:
   Student(const char* n, int a) {
      strcpy(nume, n);
      anul = a;
   }
   void  setNume(const char* n) {
      strcpy(nume, n);
   }
   char* getNume() {
      return  nume;
   }
   void  setAnul(int  a) {
      anul = a;
   }
   int  getAnul() {
      return  anul;
   }
   virtual  void  mesaj() {
      cout << "Student  sanatos";
   }
};

class  StudentCovid : public  Student
{
   int  grupa;
public:
   StudentCovid(char* n, int  a, int  g);
   void  setGrupa(int  g) {
      grupa = g;
   }
   int  getGrupa() {
      return  grupa;
   }
   void  mesaj() override {//redefinire metoda virtuala
      cout << "Student  carantinat";
   }
};

StudentCovid::StudentCovid(char* n, int  a, int  g) :Student(n, a) {
   grupa = g;
}

/*Source.cpp*/
#define CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace  std;
#include "Header.h"

int  main( ){
   char  maria[ ] = "Maria", ana[ ] = "Ana", ioana[ ] = "Ioana";
   Student  ob(maria, 1), ob1(ana, 2);
   StudentCovid  ob2(ioana, 1, 2113);
   cout << "\n Obiecte clasa de baza - CB\n";
   cout << ob.getNume() << "  Anul:  " << ob.getAnul() << "  ";
```

```
   ob.mesaj();  cout << endl;
   cout << ob1.getNume() << "  Anul:  " << ob1.getAnul() << "  ";  ob1.mesaj();
   cout << "\nObiect clasa derivata - CD\n";
   cout << ob2.getNume() << "  Anul:  " << ob2.getAnul() << "  " << ob2.getGrupa() << "  ";
ob2.mesaj();
   Student* p;//pointer CB
   p = &ob;
   cout << "\nPointer  la  obiect CB" << endl;
   cout << p->getNume() << "  Anul:  " << p->getAnul() << "  "; p->mesaj();
   p = &ob2;//upcasting
   cout << "\nPointer  la  obiect CD cu acces doar la metode CB si metoda virtuala din CD" << endl;
   cout << p->getNume() << "  Anul:  " << p->getAnul() << "  ";
   // cout << "\nGrupa:  "<< p->getGrupa( ) << endl;//nu e accesibil - apartine doar CD
   p->mesaj();//e virtuala , dynamic binding
   p->setNume("Ioana-Delia");//doar la atribute comune claselor
   p->setAnul(2);
   ob2.setGrupa(2213);//specific CD
   cout << "\nUpdate obiect CD\n"<<ob2.getNume() << "  Anul:  " << ob2.getAnul() << "  " <<
ob2.getGrupa() << "  "; ob2.mesaj();
   return  0;
}
```

**Exemplul 2:**

```
//Mostenirea simpla, up/down-casting, metode virtuale
//Poz_punct.h
// clasa de baza

class Pozitie{
  protected :
int x, y;
  public :
Pozitie (int=0, int=0);
Pozitie(const Pozitie &);
~Pozitie();
//void afisare();
//void deplasare(int, int);
virtual void afisare();
virtual void deplasare(int, int);
};//CB
// constructor
Pozitie::Pozitie(int abs, int ord){
x = abs; y=ord;
cout << "Constructor CB \"Pozitie\", ";
afisare();
}

//constructor de copiere
Pozitie::Pozitie(const Pozitie &p){
x = p.x;
y = p.y;
cout << "Constructor de copiere CB \"Pozitie\", ";
afisare();
}

// destructor
Pozitie::~Pozitie(){
cout << "Destructor CB \"Pozitie \", ";
afisare();
}

void Pozitie::afisare(){
cout << " CB afisare: coordonate: x = " << x << ", y = " << y << "\n";
}
```

```
void Pozitie::deplasare(int dx, int dy){
cout<<"CB: deplasare"<<endl;
x += dx; y+=dy;
}

// clasa derivata
class Punct: public Pozitie {
int vizibil;
char culoare;
  public:
Punct(int=0, int=0, char='A');
Punct(const Punct &);
~Punct();
void arata() {vizibil = 1;
}
void ascunde() {vizibil = 0;
}
void coloreaza(char c) {culoare = c;
}
void deplasare(int, int);
void afisare();
};//CD

// constructor
Punct::Punct(int abs, int ord, char c):Pozitie(abs, ord){
vizibil = 0;
culoare = c;
cout << "Constructor CD \"Punct\", ";
afisare();//CD
}

// constructor de copiere
Punct::Punct(const Punct &p):Pozitie(p.x, p.y){
vizibil = p.vizibil;
culoare = p.culoare;
cout << "Constructor de copiere CD \"Punct\", ";
afisare();//CD
}

// destructor
Punct::~Punct(){
cout << "Destructor CD \"Punct\", ";
afisare();//CD
}

// redefinire functie de deplasare in clasa derivata
void Punct::deplasare(int dx, int dy)  override{
if (vizibil) {
cout << " CD: Deplasare afisare CD\n";
x += dx;
y += dy;
afisare();//CD
}else {
x += dx;
y += dy;
cout << "Deplasare prin CD afisare din CB\n";
Pozitie::afisare();}
}

// redefinire metoda de afisare in clasa derivata
```

```
void Punct::afisare() override{
cout << "Pozitie: x = " << x << ", y = " << y;
cout << ", culoare: " << culoare;
if (vizibil) cout << ", vizibil \n";
else cout << ", invizibil \n";
}

// program de test
#include <iostream>
using namespace std;
#include "Poz_punct.h"

int main( ){
Pozitie pp0(7,7);//base class object
cout<< "\n Metode CB \n";
pp0.afisare();
pp0.deplasare(6,9);
pp0.afisare();
cout<< "\n Metode CD \n";
Punct p0(1, 1, 'V');//derived class object
p0.afisare();
Punct p1(p0);
p1.arata();
p1.deplasare(10,10);
cout<<"\nUpcasting - obiecte:\n";
pp0=p0;//upcasting by objects
pp0.afisare();
cout<<"\nUpcasting - pointeri:\n ";
Pozitie *p;//base class pointer
p=new Punct(100,100,'Z');//derived object to the base class pointer
//cout<<"\nAfisare CB: \n"; non virtual
cout << "\nAfisare CD: derived class  object if virtual, else base class CB \n";
p->afisare();//afis invizibil
p = &pp0;
cout << "\nAfisare CB: base class  object always\n";
p->afisare();
p = &p1;
cout<<"\nAfisare CD: derived class object if virtual, else base class CB \n";
p->afisare();
Punct *pp;
pp= &p1;
cout << "\nAfisare CD: derived class object always\n";
pp->afisare();
cout << "\n Deplasare CD with 10, 10 \n";
pp->deplasare(10,10);
cout << "\nAfisare CD: derived class object with ascunde()\n";
pp->ascunde();
pp->afisare();
cout << "\n Deplasare CD with 10, 10 and ascunde()\n";
pp->deplasare(10, 10);
cout << "\nAfisare direct  from CB: derived object displayed with base class method always\n";
pp->Pozitie::afisare();
cout << "\nDowncasting:\n ";
Punct* pdown;//derived pointer
pdown = (Punct*)&pp0;//downcasting by base class object
cout << "\nAfisare CB: base class  object using a derived pointer, else derived class CD \n";
pdown->afisare();
pdown = (Punct*)p;//downcasting by derived class object
cout << "\n Afisare din Derivat, Punct" << endl;
pdown->afisare();
```

```
return 0;
}
//*****************************************************************
```

**Exemplul 3:**

```
//Modalitatea de definire si utilizare a metodelor virtuale
//Header.h

class Vehicul {
                    int roti;
                    float greutate;

public:
                    virtual void mesaj( )
                    {
                              cout << "Mesaj din clasa Vehicul\n";
                    }
};

class Automobil : public Vehicul {
                    int incarcatura_pasageri;

public:
                    void mesaj( )  override
                    {
                              cout << "Mesaj din clasa Automobil\n";
                    }
};

class Camion : public Vehicul {
                    int incarcatura_pasageri;
                    float incarcatura_utila;

public:
                    int pasageri( )
                    {
                              return incarcatura_pasageri;
                    }
};

class Barca : public Vehicul {
                    int incarcatura_pasageri;

public:
                    int pasegeri( )
                    {
                              return incarcatura_pasageri;
                    }
                    void mesaj( )  override
                    {
                              cout << "Mesaj din clasa Barca\n";
                    }
};

//main
#include<iostream>
using namespace std;
#include "Header.h"

int main( )
```

```
{
// apel direct, prin intermediul unor obiecte specifice
Vehicul monocicleta;
Automobil ford;
Camion semi;
Barca barca_de_pescuit;

                    monocicleta.mesaj( );
                    ford.mesaj( );
                    semi.mesaj( );//din Vehicul ca si CB
                    barca_de_pescuit.mesaj( );

// apel prin intermediul unui pointer specific
Vehicul *pmonocicleta;
Automobil *pford;
Camion *psemi;
Barca *pbarca_de_pescuit;

                    cout << "\n";
                    pmonocicleta = &monocicleta;
                    pmonocicleta->mesaj( );

                    pford = &ford;
                    pford->mesaj( );

                    psemi = &semi;
                    psemi->mesaj( );//din CB

                    pbarca_de_pescuit = &barca_de_pescuit;
                    pbarca_de_pescuit->mesaj( );

// apel prin intermediul unui pointer catre un obiect al clasei de baza
                    cout << "\n";
                    pmonocicleta = &monocicleta;
                    pmonocicleta->mesaj( );//Vehicul

                    pmonocicleta = &ford;//upcasting
                    pmonocicleta->mesaj( );//Automobil

                    pmonocicleta = &semi;//upcasting
                    pmonocicleta->mesaj( );//Camion- Vehicul

                    pmonocicleta = &barca_de_pescuit;//upcasting
                    pmonocicleta->mesaj( );//Barca
          return 0;
}
```

*Tema: Asociati pointerul clasei de baza catre obiecte derivate si verificati functionalitatea. Realizati aceleasi operatii considerand metodele viruale din clasa de baza, ca fiind non virtuale. Analizati rezultatele.*

```
//*************************************************************************
```

**Exemplul 4:**

```
//Exemplu cu clase abstracte si metode virtuale pure
//Header.h

enum Color {Co_red, Co_green, Co_blue};

// clasa de baza abstracta
```

```
class Shape {
    protected:
        int xorig;
        int yorig;
        Color co;

    public:
        Shape(int x, int y, Color c) : xorig(x), yorig(y), co(c) { }

        virtual ~Shape( ) { }          // destructor virtual
        virtual void draw( ) = 0;      // metoda virtuala pura

};
// O metoda virtuala pura face clasa in care apare ca fiind clasa abstracta.
// Metoda virtuala pura trebuie definita in clasele derivate sau redeclarata ca
//metoda virtuala pura in clasele derivate.

// clasa Line (intre origine si un punct destinatie)
class Line : public Shape {
        int xdest;
        int ydest;
    public:
        Line(int x, int y, Color c, int xd, int yd) :
        xdest(xd), ydest(yd), Shape(x, y, c) { }

        ~Line( ) {cout << "~Linie\n";} // destructor virtual

        void draw( ) // metoda virtuala
        {
            cout << "Linie" << "(";
            cout << xorig << ", " << yorig << ", " << int(co);
            cout << ", " << xdest << ", " << ydest;
            cout << ")\n";
        }
};

// Clasa Circle : cerc cu raza
class Circle : public Shape {
        int raza;

    public:
        Circle(int x, int y, Color c, int r) : raza(r), Shape(x, y, c) { }

        ~Circle( ) {cout << "~Cerc\n";} // destructor virtual
        void draw( ) // metoda virtuala
        {
            cout << "Cerc" << "(";
            cout << xorig << ", " << yorig << ", " << int(co);
            cout << ", " << raza;
            cout << ")\n";
        }
};

// Clasa Text : text
class Text : public Shape {
        char* str;
    public:
        Text(int x, int y, Color c, const char* s) : Shape(x, y, c)
        {
            str = new char[strlen(s) + 1];
```

```
            strcpy(str, s);
        }

        ~Text( ) {delete [ ] str; cout << "~Text\n";} // destructor virtual

        void draw( ) // metoda  virtuala
        {
            cout << "Text" << "(";
            cout << xorig << ", " << yorig << ", " << int(co);
            cout << ", " << str;
            cout << ")\n";
        }
 };

//main( )
#include<iostream>
using namespace std;
#include "Header.h"
 int  main( )
  {
      const int N = 5;
      int i;
      Shape* sptrs[N];
//upcasting
 sptrs[0] = new Line(1, 1, Co_blue, 4, 5);
 sptrs[1] = new Line(3, 2, Co_red, 9, 75);
 sptrs[2] = new Circle(5, 5, Co_green, 3);
 sptrs[3] = new Text(7, 4, Co_blue, "Salut echipa de lucru ...&...!");
 sptrs[4] = new Circle(3, 3, Co_red, 10);
      for (i = 0; i < N; i++)
          sptrs[i]->draw( );

      for (i = 0; i < N; i++)
          delete sptrs[i];
return 0;
}

//*************************************************************************
```

**Exemplul 5:**

```
//Clasa de baza Baza este mostenita virtual atat de Derivata1 cat si de Derivata 2,
//dar doar un singur obiect de timp Baza va fi creat, la instantierea unui obiect
//din clasa Derivat1si2
//Header.h

class Baza {
protected:
        int x;
public:
        Baza( ){
                cout<<"Apel la constructorul clasei de baza\n";
                x=10;
        }
}; // clasa de baza

class Derivata1 : virtual public Baza {
public:
        Derivata1( ){
                cout<<"Apel la constructorul clasei Derivata1\n";
        }
};
```

```
class Derivata2 : virtual public Baza {
public:
	Derivata2( ){
		cout<<"Apel la constructorul clasei Derivata2\n";
	}
};

class Derivata1si2 : public Derivata1, public Derivata2 {
public:
	Derivata1si2( ){
		cout<<"Apel la constructorul clasei Derivata1si2\n";
	}
};

//main( )
#include<iostream>
using namespace std;
#include "Header.h"

int main( ){
	Derivata1si2 ob;
	return 0;
}
```

//In functia *main()* afisati atributul *x* si adaugati o metoda accesor adecvata astfel incat sa puteti accesa atributul *x* din clasa de *Baza* folosind obiectul derivat instantiat. Analizati si cazul mostenirii nevirtuale.

**Teme:**

17. În cazul exemplului 2 (care exemplifică moştenirea simplă, cu clasa de bază *Pozitie* şi derivată *Punct*) se cer următoarele:
    a. urmariţi şi verificaţi ordinea de apel pentru constructori/destructori
    b. extindeţi funcţia *main( )* pentru a utiliza toate metodele din clasa de baza şi din clasa derivată
    c. introduceţi o nouă clasă *Cerc* (date şi metode), derivată din clasa *Pozitie*
    d. scrieţi un program ce utilizează aceste clase.
18. La exemplul al treilea extindeţi clasa de bază cu alte metode virtuale, redefinite în clasele derivate, cum ar fi metode *get( )* şi *set( )* pentru greutatea vehiculului (variabila *greutate*).
19. Să se scrie un program C++ în care se defineşte o clasă *Militar* cu o metodă publică virtuală *sunt_militar( )* care indică apartenenţa la armată. Derivaţi clasa *Militar* pentru a crea clasa *Soldat* şi clasa *Ofiter*. Derivaţi mai departe clasa *Ofiter* pentru a obtine clasele *Locotenent*, *Colonel, Capitan, General*. Redefiniti metoda *sunt_militar( )* pentru a indica gradul militar pentru fiecare clasa specifica. Instantiati fiecare clasa *Soldat, Locotenent,...,General*, si apelati metoda *sunt_militar( )*.
20. Declarati o clasa *Animal*, care va contine o metoda pur virtuala, *respira( )* si doua metode virtuale *manaca( )* si *doarme( )*. Derivati in mod public o clasa *Caine* si alta *Peste*, care vor defini metoda pur virtuala, iar clasa *Caine* va redefini metoda *mananca( )*, iar *Peste* metoda *doarme( )*. Instantiati obiecte din cele doua clase si apelati metodele specifice. Definiti apoi un tablou de tip *Animal*, care va contine obiecte din clasele derivate, daca e posibil. Daca nu, gasiti o solutie adecvata.
21. Definiti o clasa abstracta care contine 3 declaratii de metode pentru concatenarea, intreteserea a doua siruri de caractere si inversarea unui sir de caractere primit ca parametru. O subclasa implementeaza corpurile metodelor declarate in clasa de baza. Instantiati clasa derivata si afisati rezultatele aplicarii operatiilor implementate in clasa asupra unor siruri de caractere citite de la tastatura. Examinati eroarea data de incercarea de a instantia clasa de baza.
22. Definiti o clasa numita *Record* care stocheaza informatiile aferente unei melodii (artist, titlu, durata). O clasa abstracta (*Playlist*) contine ca variabila privata un pointer spre un sir de obiecte de tip inregistrare. In constructor se aloca memorie pentru un numar de inregistrari definit de utilizator. Clasa contine metode accesor si mutator pentru datele componente ale unei inregistrari si o metoda pur virtuala cu un parametru (abstracta), care poate ordona sirul de

inregistrari dupa un anumit criteriu codat in valoarea intreaga primita ca parametru (1=ordonare dupa titlu, 2=ordonare dupa artist, 3=ordonare dupa durata). Intr-o alta clasa (*PlaylistImplementation*) derivata din *Playlist* se implementeaza corpul metodei abstracte de sortare.

In functia *main( )*, sa se instantieze un obiect din clasa *PlaylistImplementation* si apoi sa se foloseasca datele si metodele aferente.

23. Scrieţi o aplicaţie C/C++ în care să implementați clasa de bază abstracta *PatrulaterAbstract* avand ca atribute protected patru instante ale clasei de baza *Punct* (o pereche de coordonate *x* si *y*, accesori si mutatori) reprezentand coordonatele colturilor patrulaterului. Declarați două metode membre pur virtuale pentru calculul ariei și perimetrului figurii definite. Derivați clasa *PatrulaterConcret* care implementeaza metodele abstracte mostenite si care contine o metoda proprie care determina daca patrulaterul este patrat, dreptunghi, patrulater oarecare (convex/concav). În programul principal instanțiați clasa derivata si apelați metodele implementate. Ariile se vor calcula functie de tipul patrulaterului. La patrulaterul convex oarecare aria va fi data de urmatoarea formula care exprima aria functie de laturile a, b, c, d, semiperimetrul s, si de diagonalele p, q:

    A=sqrt {(s-a)(s-b)(s-c)(s-d)-1/4(ac+bd+pq)(ac+bd-pq)}. La patrulaterul concav se va determina doar perimetrul.

24. Considerați clasa *Fractie* care are doua atribute întregi protected *a* si *b* pentru numărător si numitor, doua metode de tip *set( )* respectiv *get( )* pentru atributele clasei. Declarați o metoda virtuala *simplifica( )* care simplifica un obiect *Fractie* folosind *cmmdc-ul* determinat prin operatorul *%*. Definiți un constructor explicit fără parametri care inițializează *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 fracții (*b*!=0). Supraîncărcați operatorii de adunare, scadere, inmultire si impartire (+,-,*,/) a fracțiilor folosind functii friend 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) si redefinește metoda *simplifica( )* folosind pentru *cmmmdc* algoritmul prin diferența. Afișați un mesaj adecvat in metoda. Definiți de asemenea supraîncărcarea operatorilor compuși de asignare si adunare, scadere, inmultire si impartire(+=,-=,*=,/=) cu metode membre. Supraîncărcați operatorii de incrementare si decrementare postfixați 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 functiile *friend* din clasa de baza, 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 implementate prin metodele clasei, asignând rezultatele obținute la alte 4 obiecte *Fracti_ext*. Folosiți pentru operații copii ale obiectelor inițiale. Simplificați si afișați rezultatele. Verificați posibilitatea utilizării celor doua metode de tip *simplifica( )* (din clasa de baza si derivata) folosind instanțe din clasa de baza si derivata folosind un pointer catre clasa de baza *Fractie*.

**Homework:**

17. Considering the second example (simple inheritance, the base class *Pozitie* and the derived class *Punct)*, resolve the following tasks:
    a. verify the order in which the constructors and destructors are called
    b. extend the main function in order to use all the methods from the base and derived class
    c. write a new class called *Cerc* (attributes and methods) derived from *Pozitie*
    d. write a program that uses the classes mentioned before
18. Extend the base class from the third example by adding some other virtual methods, which will be implemented in the derived classes (like the *setter* and *getter* for the value of *greutate*).
19. Write a C++ program that defines a class called *Militar* that has a public virtual method *sunt_militar( )*. Define the classes *Soldat* and *Ofiter*, both being derived from the first class. Extend further the *Ofiter* class by implementing the classes *Locotenent, Colonel, Capitan, General*. Override the method *sunt_militar( )* for indicating the military degree represented by each class. Instantiate each of the classes *Soldat, Locotenent,...,General* and call the *sunt_militar()* method.
20. Declare a class called *Animal* that contains a pure virtual method (*respira( )*) and 2 virtual methods (*mananca( )* and *doarme( )*). The classes *Caine* and *Peste* inherit the first class in a

public mode and implements the pure virtual method. The class *Caine* overrides the *mananca()* method. The class *Peste* overrides the *doarme( )* method. Instantiate the derived classes and call the specific methods. After that, define an array of *Animal* objects that will contain instances of the derived classes (if that's possible). If not, find an appropriate solution.

21. Define an abstract class that contains 3 method declarations for concatenating, interlacing two arrays of characters and for reverting the character array received as parameter. A subclass implements the methods declared in the base class. Instantiate the 2-nd class and display the results produced by applying the methods mentioned above upon some data read from the keyboard. Examine the error given by the attempt of instantiating the base class.
22. Define a class called *Record* that stores the data related to a melody (artist, title, duration). An abstract class (*Playlist*) contains as private variable a pointer to an array of records. The pointer is initialized in the constructor by a memory allocation process (the number of records is defined by the user). The class contains accessor and mutator methods for each of a record's fields and an abstract method (pure virtual) that sorts the records array according to a criteria coded in the received parameter (1=sorting by title, 2=sorting by artist, 3=sorting by duration). The abstract method is implemented inside another class (*PlaylistImplementation*) that inherits the *Playlist* class.
    In the *main( )* function, instantiate the *PlaylistImplementation* class and initialize and use all the related data and methods.
23. Write a C++ application that defines the abstract base class *AbstractQuadrilateral* having as protected attributes four instances of the *Point* class (a pair of *x* and *y* coordinates, *getter* and *setter* methods) that represent the quadrilateral's corners. Declare two pure virtual methods for determining the area and the perimeter of the shape. Implement the derived class *ActualQuadrilateral* that implements the inherited abstract methods and has another method for determining whether the quadrilateral is a square, rectangle, or irregular quadrilateral. Instantiate the derived class and call the defined methods. The area will be determined depending on the quadrilateral type. The irregularr convex quadrilateral are wil be determined considering the following formula that express the area in terms of the sides a, b, c, d, the semiperimeter s, and the diagonals p, q:
    A=sqrt {(s-a)(s-b)(s-c)(s-d)-1/4(ac+bd+pq)(ac+bd-pq)}. At the irregularr concave quadrilateral will be determined only the perimeter.
24. Consider the *Fraction* class that has two protected attributes *a* and *b* for the nominator and denominator and two corresponding setter and getter methods. Declare a virtual method named *simplify( )* that simplifies a fraction using the greatest common divider determined using the % operator. 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 *friend* functions and 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. The derived class redefines the implementation of *simplify( )* by determining the greatest common divider using the differences based algorithm. Display an appropriate message in this method. Overload the composed addition, subtraction, multiplication and division operators (+=, -=, *=, /=) using member methods. Use member methods for overloading the post-increment and post-decrement operators that will add 1 to the value of a *Fraction_ext* instance. Instantiate 2 *Fraction* objects without parameters. Set the attributes using values read from the keyboard. Perform the operations implemented with *friend* functions from the base class and initialize another 4 objects with the obtained results. Simplify the results. Instantiate two objects of *Fraction_ext* type with data from the KB. Perform the implemented operations with the member functions and methods. Assign the operation results to other 4 existing *Fraction_ext* objects. Use for operations copies of the initial objects. Simplify and display the obtained results. Verify the possibility of using both *simplify( )* methods (base and derived class) using instances of the base and derived classes and a pointer of *Fraction* type.