RO: Clase şi metode virtuale. Clase abstracte.
EN: Virtual classes and methods. Abstract classes.

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 (si hibrida) 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 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(2114);//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 – exemplu didactic

//Poz_punct.h
// clasa de baza
class Pozitie{
  protected :
int x, y;
  public :
Pozitie (int=0, int=0);
Pozitie(const Pozitie &);//exista unul  implicit standard, aici unul diferit
~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;//ca un flag
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 pur virtuala definita
        {
            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 pur virtuala definita
        {
            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 pur  virtuala definita
        {
            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];//tablou de pointeri CB- voi folosi upcasting
//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;
}
```

Tema: *Introduceti in clasa Text un copy constructor si supraincarcati operatorul de asignare. Testati functionalitatea folosind obiecte din clasa Text.*

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

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;
}
```

Tema: 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 nonvirtuale.

Teme:

1. 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.
2. 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*).
3. 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( ).*
4. 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.
5. Definiti o clasa abstracta care contine 3 declaratii de metode pur virtuale 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.
6. Definiti o clasa numita *Record* care stocheaza informatiile aferente unei melodii (artist, titlu, durata). O clasa abstracta (*Playlist*) contine ca atribut privat 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.

7. Scrieți o aplicație 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.
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 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:

1. 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.
2. 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*).
3. 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.
4. 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.

5. Define an abstract class that contains 3 pure virtual methods 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.
6. Define a class called *Record* that stores the data related to a melody (artist, title, duration). An abstract class (*Playlist*) contains as private attribute 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.
7. 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.
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. 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.