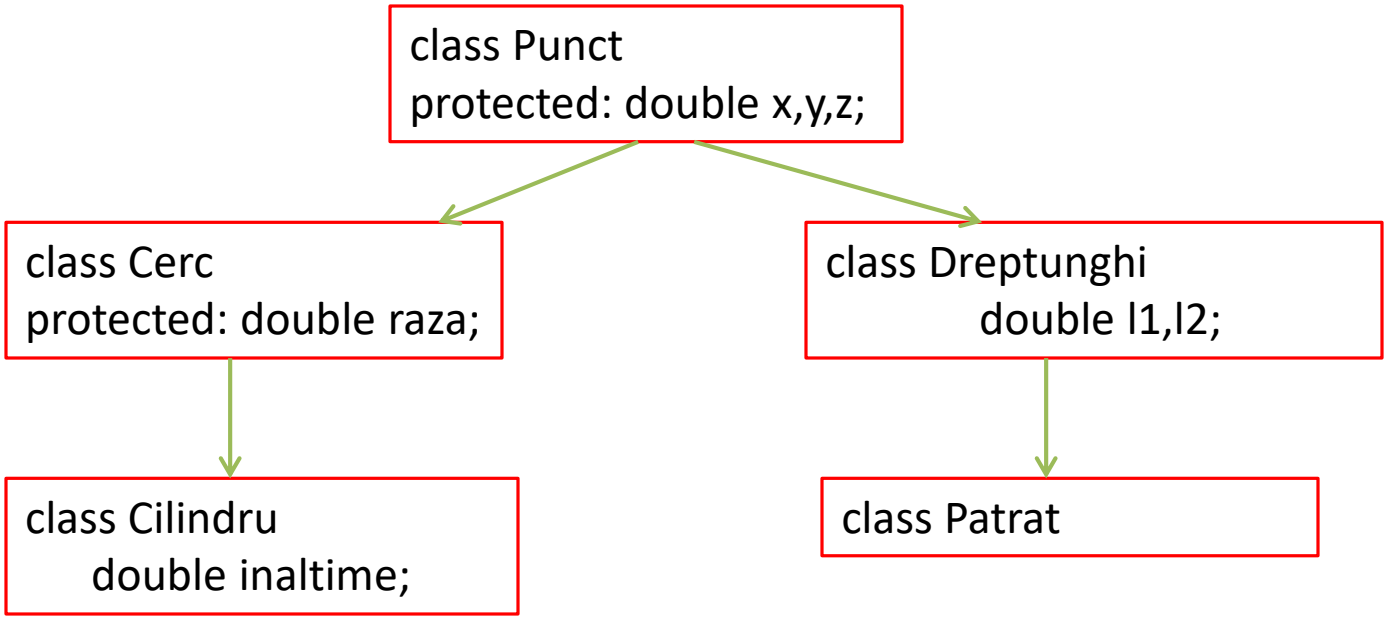
C8: Mostenire multipla. Clase si functii abstracte. Interfete

Cuprins:

- Principiul de substitutie Liskov (LSP)
- Functii inline
- Mostenire multipla – “death diamond” – clase de baza virtuale
- Ambiguitati la derivarea din mai multe clase de baza; tratarea ambiguitatilor
- Identificatorii final si override (C++ 11)
- Functii si clase abstracte
- Interfete
- Functii virtuale cu tipuri de date returnate diferite

Considerati urmatoarea ierarhie de clase:



Cerinta:

Realizati un vector care sa contina obiecte de tip: Cerc, Punct, Cilindru si Dreptunghi (lista neomogena).

Parcurgeti-l si afisati **toate** atributele figurilor, precum si **aria, perimetrul si volumul**.

Unde ati adauga in ierarhie o clasa Patrat?

Observatie: toate metodele virtuale necesare trebuie declarate in clasa de baza pentru a putea sa fie accesate prin intermediul pointerilor de tip Punct stocati in vectorul care va fi creat (altfel vom avea o trunchere a VTABLE-urilor claselor derivate la tipul de baza - Punct).

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

class Punct
{
  protected: double x,y,z;

  public:
     Punct(double i, double j, double k):x(i),y(j),z(k){ }

     virtual double perimetrul(){  return 0;  }

     virtual double aria(){  return 0;  }

     virtual double volumul(){  return 0;  }

     virtual  void afisare(){
            cout<<"Coordonate centru: "<<x<<", "<<y <<", "<<z <<endl;
            cout<<"Perimetrul: "<<perimetrul()<<endl;
            cout<<"Aria: "<<aria()<<endl;
            cout<<"Volumul: "<<volumul()<<endl;}

     virtual ~Punct(){} //pentru ca am metode virtuale

    };
```

```
class Cerc:public Punct
{ protected: double R;
  static const double pi=3.14; //o singura instanta folosita in comun de catre toate obiectele Cerc
 public:
     Cerc(double i,double j, double k,double raza):Punct(i,j,k),R(raza){}
     void afisare(){     cout<<"Raza:"<<R<<endl; Punct::afisare();   }
     double perimetrul(){  return 2*pi*R; }
     double aria(){  return pi*R*R; }
     void setR (double raza){ R=raza; }
    //metoda volum va fi mostenita din clasa Punct, nu trebuie sa ii schimb implementarea
  };
```

```
class Dreptunghi:public Punct
{ protected: double l1,l2;
 public:
     Dreptunghi(double i,double j, double k,double ll1,double ll2):Punct(i,j,k),l1(ll1),l2(ll2){}
     void afisare(){cout<<"Laturile:"<<l1<<", "<<l2<<endl; Punct::afisare();   }
     double perimetrul(){ return 2*(l1+l2); }
     double aria(){ return l1*l2; }
     void setL1(double l) { l1=l; }
     void setL2(double l) { l2=l; }
 //metoda volum va fi mostenita din clasa Punct, nu trebuie sa ii schimb implementarea
 };                                                    //bold –  reutilizarea codului;
```

```
class Patrat:public Dreptunghi
{
  public:
      Patrat(double i,double j, double k,double l):Dreptunghi(i,j,k,l,l){} //patrat o sa aiba 5 atribute;
      // 4 erau suficiente; cum pot rectifica aceasta problema?
      void afisare(){   cout<<"Latura:"<<l1<<endl;   Punct::afisare();       }
     //celelate metode se mostenesc din clasa Dreptunghi;
}; //setL2 si setL1 se mostenesc din Dreptunghi, ce se poate intampla?


  class Cilindru:public Cerc
 {      double H;
  public:
      Cilindru(double i, double j, double k, double raza, double inal):Cerc(i,j,k,raza),H(inal){}

      void afisare(){cout<<"Inaltimea:"<<H<<endl;   Cerc::afisare(); }

      double perimetrul(){return 0;}//daca nu faceam aceasta implementare se
                                          //mostenea metoda perimetrul() din clasa Cerc

      double aria() { return 2*Cerc::aria()+ H*Cerc::perimetrul();}

      double volumul() { return Cerc::aria()*H;}

      void setH(double h) { H=h; }
     };
```

```
int main()
    {
       Punct ** v=new Punct*[5];
        v[0]=new Cerc(1,1,0,8);
        v[1]=new Cerc(2,2,0,2);
        v[2]=new Cilindru(3,3,3,3,3);
        v[3]=new Dreptunghi(4,4,0,4,1);
        v[4]=new Patrat(5,5,0,5);

//1. Ce se intampla daca v continea obiecte de tip Punct, nu Punct*?
//    Cum arata v in memorie?
        for (int i=0;i<5;i++)
        {
          v[i]->afisare(); //afisez toate atributele, aria, perimetrul si volumul
         cout<<endl<<"_____________________________________"<<endl;
         }

 //2. Ce se intampla daca  metoda afisare()/aria() /perimetrul() /volumul() nu era virtuala?

        delete [] v;      //apel destructori -virtuali

//3. Ce se intampla daca  destructorii nu erau virtuali?

        return 0;
     }
```

OBSERVATII

1. Cum procedez daca nu vreau sa mostenesc atribute dintr-o clasa de baza?

- daca le declar private in clasa de baza sau folosesc derivare privata – nu fac decat sa le ascund in clasa derivata;
- **atributele clasei de baza se mostenesc mereu in derivata**

Principiul de substitutie Liskov (Liskov substitution principle - LSP) spune ca:
daca B este un subtip al lui A, atunci obiectele de tipul A din program pot fi inlocuite cu obiecte de tip B fara a altera proprietatile dorite ale aplicatiei (corectitudinea).

Incalc acest principiu undeva in aplicatia precedenta?

```
            Dreptunghi d(0,0,0,3,1);
            p.setL1(2);
            cout<< p.aria();  //aria 2
//LSP
              Patrat d(0,0,0,3);
              d.setL1(2);
              cout<< d.aria();                    //6  in loc de 4 cat ma asteptam
```

=>

=>Ierarhia de clase propusa nu respecta LSP. Cum rezolv problema?

```
class Patrat:public Punct
{   protected: double l1;
  public:
      Patrat(){}
      Patrat(double i,double j, double k,double l):Punct(i,j,k),l1(l){}
      void afisare(){cout<<"Latura:"<<l1<<endl;     Punct::afisare();}
      double perimetrul(){return 4*l1;}
      double aria(){return l1*l1;}
      void setL1(double l) {l1=l;}
  };

class Dreptunghi:public Patrat
{  double l2;
  public:
  Dreptunghi(double i, double j, double k, double ll1, double ll2):Patrat(i,j,k,ll1), l2(ll2){}
void setL2(double l) {l2=l;}
//suprascriu metodele aria, perimetrul si afisare
};
//nu mai pot sa am un comportament nedorit
```

```
/*2. */#include <iostream>
using namespace std;

class Baza
{ protected: int atr1=0;
  public:     Baza(){}
              Baza(int i ){atr1=i;}
};

class Derivata1:public Baza
{ protected: int atr2=0;
  public:     Derivata1(){}
              Derivata1(int i, int j):Baza(i){atr2=j;}
};

class Derivata2:public Derivata1
{ protected: int atr3=0;
  public: Derivata2(){}
          Derivata2(int i, int j,int k):Baza(i){atr2=j; atr3=k;}
};
    int main()
     {
             Derivata2 d(3,3,3);
             // ERROR:  type `class Baza' is not a direct base of `Derivata2'
              return 0;
    }
```

OBSERVATIE

Am dreptul sa apelez doar constructorul din clasa de baza, in cazul nostru Derivata1

DE CE?

Compilatorul il apeleaza pe cel default din Derivata1 care il apeleaza pe cel default din Baza =>s-ar apela de doua ori constructorul din Baza

* exista o exceptie – slide 16

3. Functii virtuale declarate inline

Daca functia este inline - codul functiei este substituit in sursa compilata*, nu se face efectiv apel de functie si astfel castig timp la executie.

OBS: Functiile implementate in headear sunt automat tratate ca inline*.

Sunt vreodata functiile membre virtuale inline, chiar inline?

- uneori

- mai exact, daca obiectul este referit prin pointer sau referinta, un apel catre o functie virtuala **NU** poate sa fie tratat ca inline (legare dinamica - are loc dupa compilare)

- **totusi, functia *poate** fi tratata ca inline daca la compilare se stie exact tipul obiectului** (nu lucrez cu un pointer sau o referinta catre un obiect – legare statica).

**compilatorul decide daca sa substituie apelul functiei cu codul efectiv*

Observatie:

Diferenta din punct de vedere al **vitezei de executie** intre functiile declarate inlinie si cele ne-inline este mult mai importanta decat diferenta intre o functie normala si una virtuala.

*Diferenta intre o functie obisnuita si o functie virtuala este de doar **o referinta in plus in memorie - pentru functia virtuala**.

*Diferenta intre functii inline si ne-inline poate sa aiba influenta mai mare asupra vitezei de executie (iar daca avem multe apeluri ale acelei functiei membre prin intermediul obiectelor care nu sunt pointeri sau referinte, iar functia nu e inline, o sa avem o degradare substantiala a vitezei de executie a aplicatiei).

Implementati orice functie scurta ca inline (de exemplu : functiile pentru calculul ariei, volumului, perimetrului, metodele pentru setarea atributelor din aplicatia precedenta)!

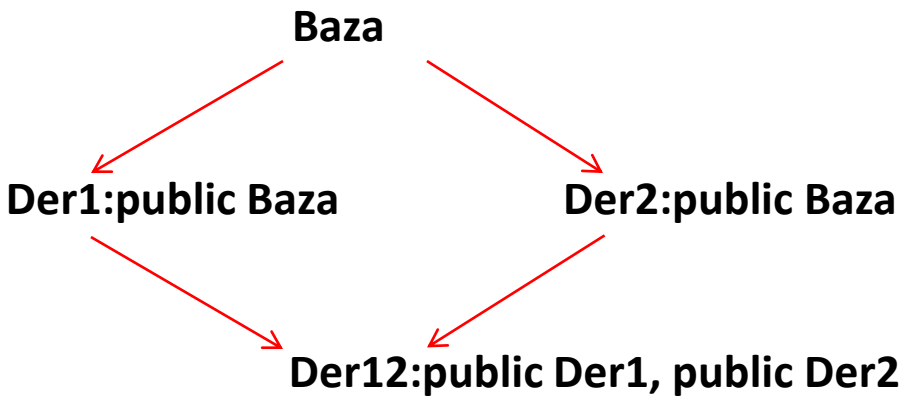
Pot sa declar constructori/ destructori ca inline?

DA

C8: Mostenire multipla. Clase si functii abstracte. Interfete

Mostenire multipla. Clase de baza virtuale.

Ce se intampla daca avem urmatoarea ierarhie? (“death diamond”)



C6: In cazul in care o clasa este derivata din doua clase de baza, constructorul sau o sa se apeleze constructorii claselor de baza.

Der12():Der1(),Der2(){}

Ce se intampla in cazul de fata?

Problema: Constructorul clasei Der12 va apela constructorii Der1 si Der2, care vor apela fiecare constructorul Baza.

Automat, se va aloca de doua ori spatiu pentru atributele mostenite din Baza.

Functiile din Baza sunt mostenite in dublu exemplar.

In niciun caz nu doream sa se intample asa ceva.

Pentru a rezolva aceasta situatie – clasele de baza Der1 si Der2 trebuiesc derivate in mod virtual din clasa de baza - > sunt **clase de baza virtuale** pentru Der12:

```
class Der1: virtual public Baza
class Der2: virtual public Baza
```

C8: Mostenire multipla. Clase si functii abstracte. Interfete

Clasele de baza virtuale:

- sunt utilizate in cazul **mostenirii multiple cand o serie de clase sunt derivate din aceeasi clasa de baza si ele urmeaza sa fie clase parinte pentru o alta clasa**

- **sintaxa:** class Der1: **virtual** public Baza
  …
  class DerN: **virtual** public Baza

- **efectul:** - nu se observa la nivelul claselor Der1,..DerN, ci in urmatorul nivel de derivare (in clasa Der1_N) – atributele din baza o sa fie mostenite intr-o singura copie

class Der1_N: public Der1,…, public DerN

OBS: In cazul utilizarii claselor virtuale, pentru a crea un obiect de tip Der1_N - se vor apela constructorii claselor imediat superioare, **dar se va apela o singura data constructorul clasei Baza, mai exact cel fara parametrii.**

Este sarcina programatorului sa initializeze explicit atributele din clasa Baza, daca acest lucru e de interes.

Avem comportament similar pentru destructorul ~Der1_N().

```
class Baza
{protected:
            int id=0;
public:
            inline Baza(){};
             inline Baza(int ii):id(ii){};
            virtual inline void afisare(){   cout << id ;  }
};

class Der1: virtual public Baza
{protected :
            int x=0;
public:
            inline Der1(int ii,int xx):Baza(ii), x(xx){   };
            inline void afisare(){   Baza::afisare();  cout << "  "<< x << endl;  }
};

class Der2: virtual public Baza
{protected :
            int y=0;
public:
             inline Der2(int ii,int yy):Baza(ii),y(yy){  };
             void afisare(){ Baza::afisare();  cout << " " << y << endl;  }
};           //afisare ramane virtuala, chiar daca nu mai e declarata inline
```

OBS: Faptul ca am declarat functiile inline ar ajuta daca implementarea lor nu ar fi facuta in header. (Teoretic trebuia sa impart clasele in interfete - headere si implementare).

```
class Der12: public Der1,public Der2
{
                int z;
public:
    //          Der12(int ii,int xx,int yy,int zz):Der1(ii,xx),Der2(ii,yy),z(zz){};
//OBS: Cand se face apelul constructorilor Der1 si Der2 ne asteptam sa se apeleze constructorul
//        Baza(int) care seteaza atributul din clasa Baza.
//        Dar nu se intampla asta. Compilatorul apeleaza constructorul default Baza trecand peste
//        implementarea data in Der1 si Der2 constructorilor cu parametri.
//        In acest caz, daca nu implementam constructorul Baza() aveam o eroare.
//         Mecansimul e specific pentru derivarea din clase de baza virtuale-se asigura protectie in cazul
//         in care am transmite valori diferite prin Der1(int,int) si Der2(int,int)  in Baza(int)

//daca vreau sa apelez explicit constructorul Baza(int) o sa scriu:
              Der12(int ii,int xx,int yy,int zz):Der1(ii,xx),Der2(ii,yy),Baza(ii) , z(zz){};
             //testati 1.fara apel Baza(ii) si 2. apoi comentati implementarea Baza::Baza()

              void afisare(){Der1::afisare(); cout <" " << y <<" "<< z<< endl;}//afisez doar o data id
};
//OBS: Destructorul pentru obiecte de tip Der12 va apela o singura data destructorul generat
//automat  ~Baza().
```

```
int main()
{   Der12(1,2,3,4).afisare();

    int n;
    cout<<"dati dimensiunea:"<<endl;
    cin>>n;
    Baza ** colectie=new Baza*[n];

for(int i=0;i < n; i++)
{
            if (i%4==0)       colectie[i]=new Der1(i,i);
            else if (i%4==1) colectie[i]=new Der2(i,i);
            else if (i%4==2) colectie[i]=new Der12(i,i,i,i);
            else if (i%4==3) colectie[i]=new Baza(i);
}

for(int i=0;i < n; i++)
{
cout << "Obiectul de pe pozitia "<< i <<" este un "<<typeid(colectie[i]).name();
cout << " care pointeaza catre un obiect de tipul: "<<typeid(*colectie[i]).name();
cout<<" care contine valorile:"<<endl;
colectie[i]->afisare();
cout<<endl<<"_____________________"<<endl;
}
return 0;}
```

//ce listeaza aplicatia

Obiectul de pe pozitia 0 este un Baza care pointeaza catre un obiect de tipul: Der1 care contine valorile: 0 0

Obiectul de pe pozitia 1 este un Baza care pointeaza catre un obiect de tipul: Der2 care contine valorile: 1 1

Obiectul de pe pozitia 2 este un Baza care pointeaza catre un obiect de tipul: Der12 care contine valorile: 2 2 2

Obiectul de pe pozitia 3 este un Baza care pointeaza catre un obiect de tipul: Baza care contine valorile: 3

typeid – #include <typeinfo>

- este un **operator** care permite aplicatiei sa recupereze tipul obiectului adresat printr-un pointer sau printr-o referinta in momentul executiei (runtime type identification - RTTI)
  - returneaza un obiect de tip std::type_info
  - nu se poate supradefinii

name() – returneaza un sir de caractere – numele tipului de date din type_info

C8: Mostenire multipla. Clase si functii abstracte. Interfete

Probleme si ambiguitati care pot aparea intr-o ierarhie de clase

1. Considerati implementarea:

```
class Der12: public Der1,public Der2
{
        int z;
public:
        Der12(int ii,int xx,int yy,int zz):Der1(ii,xx),Der2(ii,yy),Baza(ii),z(zz){};
        //void afisare(){cout << id<<" " << x <<" "<< y<< " "<<z<<endl;}
};
```

Ce se intampla daca nu implementam functia de afisare in Der12?

S-ar mosteni doua functii: Der1::afisare() si Der2::afisare(); afisare e declarata virtual in Baza

Si care va fi folosita(la run-time)?

Din fericire compilatorul nu va permite asa ceva si vom avea eroare:

ERROR no unique final overrider for `virtual void Baza::afisare()' in `Der12'

Functia afisare era virtuala si trebuia sa aiba o implementare unica!

2. Ce se intampla daca avem clasele Baza1 si Baza2 cu functii nevirtuale de afisare si o clasa derivata Der care nu redefineste afisare ?

```
class Baza1{
  protected: int atr1;
  public:     Baza1(int j){atr1=j;}
              void afisare(){cout<<atr1<<endl;}
 };

class Baza2{
  protected: int atr2;
  public:       Baza2(int i){atr2=i;}
                void afisare(){cout<<atr2;}
};

class Der: public Baza1, public Baza2 {
    public:       Der(int i, int j):Baza1(i),Baza2(j){}    } ;
```

//mostenesc ambele implementari ale functie de afisare; nu am eroare la compilarea claselor

```
     int main() {
        Der d(1,1);   d. afisare();
             // ERROR: request for member `afisare' is ambiguous , candidates are:
//void Baza2::afisare() and void Baza1::afisare()
        d.Baza1::afisare(); d.Baza2::afisare(); //trebuie sa precizez la care functie ma refer
        return 0;}
```

C8: Mostenire multipla. Clase si functii abstracte. Interfete

Ambiguitati datorate mostenirii multiple

1. Daca in doua clase de baza exista functii cu acelasi nume si aceeasi semnatura - nevirtuale, iar in clasa derivata nu se redefineste acea functie => sunt mostenite ambele versiuni

Un alt exemplu este prezentat mai jos:

2. - ambiguitatea, in acest caz, se datoreaza unui atribut cu acelasi nume in clasele Baza1 si Baza2

3. - respectiv, a doua functii cu acelasi nume, dar semnaturi diferite mostenite din Baza1 si Baza2

```
class Baza1 {
public:
  int i;
  int j;
  void f(int) {/*impl*/ }
};

class Baza2 {
public:
  int j;
  void f() {/*impl*/}
};
```

```
class Der : public Baza1, public Baza2 {
public:
  int i; //acopera ca vizibilitate atributul i din Baza1
};        // mosteneste j din Baza1 si din Baza2 si Baza1::f() si Baza2::f(int)

int main() {
  Der d;

  d.i = 5;
//este acoperit ca vizibilitate atributul din Baza1; ne referim -  fara ambiguitati la i din Der

// d.j = 10;
     //sunt mostenite 2 atribute cu acelasi nume
    //ERROR: request for member `j' is ambiguous candidates are: int Baza2::j int Baza1::j;

  d.Baza1::j = 10;   //precizez despre care j este vorba

  //  d.f();
     //   sunt mostenite doua functii cu acelasi nume , chiar daca avem semnatura diferita
     //   ERROR: request for member `f' is ambiguous   candidates are: void Baza2::f()
     //   void Baza1::f(int)
  d.Baza2::f();    //precizez despre care functie este vorba
  return 0;
}
```

C8: Mostenire multipla. Clase si functii abstracte. Interfete

Cum procedez daca nu doresc ca o functie sa poata sa fie suprascrisa intr-o clasa derivata?

1. Identificator final (C++11)

- are sens doar in contextul derivarii
- are 2 scopuri: sa previna supraincarcarea unei functii virtuale sau derivarea dintr-o clasa

Functii virtuale declarate final

- daca la un anumit punct intr-o ierarhie de clase, o functie initial virtuala este declarata ca finala, acea functie nu va putea fi supraincarcata in clasele ulterior derivate
- **o functie nu poate fi declarata final daca nu e virtuala**

```
class A {
  public:  virtual void f(){};
};
class B: public A {
  public:  void f() final {};
};
class C: public B {
  public: void f() {}; //ERROR B::f is final
};
```

Clase declarate final (C++11)

- daca la un anumit punct intr-o ierarhie de clase, o clasa este declarata finala => nu se va mai putea deriva in continuare din acea clasa

```
class O{};
class A final:public O{
};
class B: public A {
};
//ERROR class A is final
```

Identificatorul override (C++11)

- precizeaza ca o functie dintr-o clasa derivata trebuie sa suprascrie o functie din baza

```
class A {
        public: virtual void afis(){}
};
class B: public A {
          public:  void afis() const override {}
};      //ce credeti ca se intampla?
```

[Error] 'void B::afisare()' marked override, but does not override

- deci acest specificator este util pentru a preintampina astfel de erori logice (se verifica si tipul returnat)

Cum as mai putea sa ma asigur ca nu se poate deriva dintr-o anumita clasa?

Declar constructorul private. Si atunci cum mai creez obiecte?

```
#include <iostream>
using namespace std;
class Baza{
        int i;
        Baza(int x):i(x){};  //e private
public:  static Baza makeObj(int x){
            return Baza(x);
         };//o functie statica – folosita pt a crea obiecte. De ce statica?
        void afis(){ cout<<i; }
 };

/* class Der: public Baza{      // ERROR: base `Baza' with only non-default constructor in class
}; */                           // without a constructor

int main(){
 //Der d;
 Baza c=Baza::makeObj(1);
 c.afis();
 return 0;
}
```

Named Constructor Idiom: se declara private sau protected constructorii clasei si se pun la dispozitie metode statice care returneaza obiecte. Aceste metode statice se numesc (constructori numiti) "Named Constructors." In general, exista o metoda statica pentru fiecare mod de constructie al unui obiect.

C8: Mostenire multipla. Clase si functii abstracte. Interfete

Functii si clase abstracte

O metoda virtuala se numeste abstracta (pura) daca nu are implementare:

virtual tip_returnat metoda(lista parametrii) **= 0**;

O clasa se numeste abstracta daca are cel putin o metoda abstracta.

Observatii:

1. Functiile abstracte sunt folosite in cazul in care o metoda nu are o implementare posibila intr-o clasa de baza, dar dorim sa precizam ca ea are un comportament virtual in ierarhia pe care o implementam

2. Nu se pot instantia obiecte de tipul unei clase abstracte, **DAR** pot sa declar pointeri de acel tip

3. Suntem obligati, ca eventual, sa punem la dispozitie o implementare a acestei functii intr-o clasa derivata din acea clasa abstracta, altfel si derivata ar fi abstracta

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

class figura{   //clasa abstracta
public:
        virtual double perimetru()=0;     //functii abstracte
        virtual double aria()=0;
        void afisare(){//voi afisa mereu aria si perimetrul figurii
                cout<<"Aria:      "<<aria()<<endl;
                cout<<"Perimetrul: "<<perimetru()<<endl;
        }
        virtual ~figura(){};
};

class punct:public figura
{protected:
        int x,y;
public:
        punct(int xx,int yy){x=xx;y=yy;}
        double perimetru(){return 0;} //pun la dispozitie o implementare pentru functia perimetru
        double aria(){return 0;}          //pun la dispozitie o implementare pentru functia aria
};                                        //altfel nu as putea sa creez obiecte de tip punct
```

void afisare() – se mosteneste

```
class segment:public figura
{protected:
            int l;
public:     segment(int ll){l=ll;};
            double perimetru(){return l;} //pun la dispozitie o implementare pentru functia perimetru
            double aria(){return 0;}      //pun la dispozitie o implementare pentru functia aria
};

int main(int argc, char *argv[])
{
//   figura f;   //ERROR; nu pot instantia obiecte de tipul unei clase abstracte; inca nu stiu care e
                 //implementarea pentru metodele abstracte;
 segment l(1); //pot instantia obiecte de tipul claselor derivate care au implementat metodele aria si
                 //perimetru
//Si atunci de ce este abstractizarea importanta?
//Pot sa declar un vector de pointeri de tip figura:
figura **v=new figura*[2];
//si sa il populez cu obiecte de tipul claselor derivate neabstracte  (au implementat metodele abstracte)
v[0]=&l;  v[1]=new punct(3,4);
//si sa utilizez proprietatile functiilor virtuale (declarate in clasa de baza)
for (int i=0;i<2;i++) v[i]->afisare();
// altfel nu aveam cum sa creez vectorul neomogen de obiecte si sa folosesc polimorfismul
// clasele Punct si Segment nu ar fi avut o baza comuna; s-a pus la dispozitie o clasa de baza abstracta
   return 0;  }
```

void afisare() – se mosteneste

C8: Mostenire multipla. Clase si functii abstracte. Interfete

Interfata

O clasa fara atribute si cu toate metodele abstracte se numeste **interfata**.

Interfetele si clasele abstracte sunt foarte importante pentru dezvoltarea de ierarhii de clase - ele reprezinta o **schita** a ceea ce trebuie implementat/suprascris in clasele derivate.

Interfetele sunt utile pentru a preciza, de la bun inceput intr-o ierarhie de clase, care sunt metodele care trebuiesc supraincarcate in clasele derivate din acea ierarhie => **precizeaza care sunt comportamente comune (chiar daca acestea au implementari specifice) pentru tipurile de date din ierarhie.**

Clasele “concrete” - **derivate** din una sau mai multe interfete - sunt **obligate** ca, eventual, sa **furnizeze implementari** pentru toate metodele abstracte/comportamentele declarate/precizate in interfete.

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

class figura{    //interfata
public:  //nu am atribute; toate metodele sunt abstracte
                virtual double perimetru()=0;
                virtual double aria()=0;
                 virtual void afisare()=0;
};


class punct:public figura {
protected:
                int x,y;
public:
                punct(int xx,int yy){x=xx;y=yy;}
                 void afisare(){cout<<x<<y;}
};  //clasa abstracta; nu am implementat functiile aria() si perimetrul();
    //nu pot sa instantiez obiecte de tipul punct
```

```
class cerc:public punct
{protected:    int raza;
public:
            cerc(int xx,int yy,int r):punct(xx,yy){raza=r;}
            double perimetru(){return 3.14*2*raza;};
            double aria(){return 3.14*raza*raza;};
            void afisare(){cout<<"centrul"; punct::afisare();
                        cout<<"raza"<<raza<<"aria"<<aria();};
};                          //pot reutiliza codul din punct chiar daca nu pot instantia obiecte
//…alte tipuri de figuri

int main(int argc, char *argv[])
{cerc c(0,0,1);
 c.afisare();
//punct p(1,1); //punct e o clasa abstracta

 figura **vec=new figura*[2];
 vec[0]=new cerc(1,1,4);    vec[0]->afisare();

// vec[1]=new punct(1,1); //punct e o clasa abstracta
 return 0;}
```

OBSERVATII

1. Interfetele sunt extrem de utile la dezvoltarea de aplicatii mari – bine structurate. Cand se deriveaza dintr-o interfata, programatorul va fi obligat sa implementeze metodele abstracte in clasele derivate: nu are cum sa omita implementarea - altfel compilatorul va sesiza o problema

2. Considerati ca aveti urmatoarele clase:

Student {protected: char *nume; double media;}
Angajat {protected: double salariu; int vechime;}
Student_Angajat:public Student , public Angajat{}

Se cere sa se creeze o lista neomogena care sa contina obiecte de tip Student, Angajat, Student_Angajat; sa se parcurga lista si sa se afiseze toate atributele obiectelor din lista.

DAR Student si Angajat nu au in comun o clasa de baza ca sa pot realiza vectorul si folosi functii virtuale.
Pot sa creez o interfata comuna pentru cele 3 clase:

```
class Object{
public:
    virtual void afisare()=0;
    virtual ~Object(){}
};
```

- si sa derivez din ea **Student:virtual public Object** si **Angajat:virtual public Object** in care o sa implementez cele 2 metode abstracte
- iar in clasa Student_Angajat se da o implementare specifica pentru functia de afisare

C8: Mostenire multipla. Clase si functii abstracte. Interfete

Destructori virtuali puri/abstracti

- pot sa fie declarati destructori virtuali abstracti (este chiar necesar)
- totusi, sintaxa nu este cea la care ne asteptam. In prima instanta scriu:

  virtual ~tip_date()=0;
- DAR trebuie sa li se dea o implementare

```
class BazaAbstracta { //interfata
public:
virtual ~ BazaAbstracta() = 0;
};

BazaAbstracta ::~ BazaAbstracta() {}

class Derivata : public BazaAbstracta{
};
// destructorul  va fi virtual

int main() {
Derived d;
return 0;
}
```

- In prima aplicatie am creat tipuri de date pentru figuri geometrice atat 2D cat si 3D.

- O problema era ca punctul nu avea arie, perimetru sau volum, figurile 2D nu aveau volum si cele 3D nu aveau perimetru.

- Cum as putea sa realizez altfel ierarhia de clase; ce metode contine fiecare clasa in parte?

Exemplu cu mai multe niveluri care contin interfete

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

class Fig_geom    //interfata de baza
{public:
     // virtual double perimetrul()=0; -specific figurilor 2D
      virtual inline double aria()=0;
      //virtual double volumul()=0; -specific figurilor 3D
      virtual void afisare()=0;
      virtual ~Fig_geom()=0;
};
Fig_geom::~Fig_geom(){}

class Fig_geom_2D:public Fig_geom      //interfata derivata
{public:
      virtual inline double perimetrul()=0;
};

class Fig_geom_3D:public Fig_geom      //interfata derivata
{public:
      virtual inline double volumul()=0;

}; //ce contin VTABEL-urile acestor clase?
```

```
 class Cerc::public Fig_geom_2D
{  double R;
   static const double pi=3.14;
  public:
      inline Cerc(double raza):R(raza){}
      void afisare(){  cout<<"Raza:"<<R<<endl;
                       cout<<"Perimetrul: "<<perimetrul()<<endl;
                       cout<<"Aria: "<<aria()<<endl;
                      }
      inline double perimetrul(){return 2*pi*R;}
      inline double aria(){return pi*R*R;}
       }; //nu mai am in plus functia volum, care nu isi avea sensul in acest context

class Dreptunghi:public Fig_geom_2D
{
  double lung,lat;
  public:
      inline Dreptunghi(double l1,double l2):lung(l1),lat(l2){}
      void afisare(){   cout<<"Laturile:"<<lung<<", "<<lat<<endl;
                        cout<<"Perimetrul: "<<perimetrul()<<endl;
                        cout<<"Aria: "<<aria()<<endl;
                       }
       inline double perimetrul(){return 2*(lung+lat);}
       inline double aria(){return lung*lat;}
}; //nu mai am in plus functia volum, care nu isi avea sensul in acest context
```

```
class Cilindru:public Fig_geom_3D
{  double R, H;
  static const double pi=3.14;
  public:
     inline  Cilindru(double raza, double inal):R(raza),H(inal){}

    void afisare(){     cout<<"Inaltimea:"<<H<<endl;
                        cout<<"Aria: "<<aria()<<endl;
                        cout<<volumul()<<endl;
                    }
      inline double aria(){
          return 2*Cerc(R).aria()+Dreptunghi(H,Cerc(R).perimetrul()).aria();
      }

     inline double volumul(){
          return Cerc(R).aria()*H;}
     }; //nu mai am in plus functia perimetrul, care nu isi avea sensul in acest context


//deoarece am facut implementarea functiilor aria, perimetrul si volumul in headere – ele ar fi
//oricum  tratate ca inline; precizarea era necesara daca separam headerul de fisierul sursa


//implementare alte figuri 3D
```

```
int main()
{
    Fig_geom ** v=new Fig_geom*[3];
    v[0]=new Cerc(2);
    v[1]=new Cilindru(3,3); //am acces la metoda volumul?
    v[2]=new Dreptunghi(4,4); //am acces la metoda perimetrul? De ce?


    for (int i=0;i<3;i++)
            v[i]->afisare();
    delete [] v;

   Fig_geom_2D ** x=new Fig_geom_2D*[2];
    x[0]=new Cerc(2);
    x[1]=new Dreptunghi(4,4);

    for (int i=0;i<2;i++)
            cout<<  x[i]->perimetrul();
    delete [] x;

  return 0;
}
//Concluzie: structurez ierarhia in fctie de cerinta.
```

C8: Mostenire multipla. Clase si functii abstracte. Interfete

Functii virtuale - cu tip returnat diferit

In mod normal, tipul returnat de functiile virtuale nu are voie sa se schimbe de la o implementare a functiei la alta (in ierarhia de clase) prin supraincarcare.

Acest aspect este aproape mereu valabil. Exista totusi un context special in care se poate returna un alt tip de date:

Daca o functie virtuala returneaza un **pointer sau o referenta la tipul unei clasei de baza**, atunci varianta supraincarcata in clasele derivate poate sa returneze un pointer sau o referinta de tipul **unei clasei derivate din acea clasa de baza**.

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

class Hrana {  //interfata
public:
virtual char* tipHrana() const = 0;
};

class HranaPasari : public Hrana {
public:
        char* tipHrana() const
        {
                return "Graunte";
        }
};

class HranaPisici : public Hrana {
public:
        char* tipHrana() const
        {
                return "Pasari";
        }
};
```

```
class Animal { //interfata
public:
        virtual char* tip() const = 0;

        virtual Hrana* mananca() = 0;
};

class Pasare : public Animal {
private:    HranaPasari hp;

public:
        char* tip() const { return "Pasare"; }

        HranaPasari* mananca() { return &hp; }  //in mod normal scriam: Hrana*  manaca(){...}
}; //pot sa returnez HranaPasari * deoarece acest tip e derivat din Hrana

class Pisica : public Animal {
private:    HranaPisici hpi;

public:
        char* tip() const { return "Pisica"; }

        HranaPisici* mananca() { return &hpi; } //in mod normal scriam: Hrana*  manaca() {...}
};
```

```
int main() {
Pasare a;
Pisica b;
Animal* p[] = { &a, &b };
for(int i = 0; i < 2; i++)
cout << p[i]->tip() << " mananca " << p[i]->mananca()->tipHrana() << endl;

//Observatie: Care e avantajul?


//considerati ca in clasa Pisica aveam:  Hrana* mananca() { return &hpi; }
//trebuia sa convertesc Hrana* la HranaPisici* ca sa pot sa afisez tipul de hrana

HranaPisici h= (HranaPisici&)(*(b.mananca()));
cout<<h.tipHrana();  //pasari

//sau
 cout<< ((HranaPisici*)(b.mananca()))->tipHrana(); //pasari

return 0;
}
```

Tema:

1. Implementati, ca pentru ultimul exemplu, ierarhiile de clase: format_pag, format_carte, format_jurnal si publicatie, carte, jurnal (agregare cu format_pag, format_carte, format_jurnal). Creati o lista neomogena de publicatii si afisati detaliile legate de format pentru fiecare tip de publicatie in parte.

2. Gasiti un exemplu de problema ce se rezolva folosind o ierarhie de clase cu un nivel de derivare virtuala si rezolvati-o.