Obiective:

- Ierarhii de clase;
- Polimorfism;
- Functii virtuale;
- Functii virtuale pure;
- Clase abstracte;
- Mosternire multipla;
- Clase virtuale.

Clasele de baza virtuale

Pentru a preveni ca programele noastre sa mosteneasca mai multe copii ale unei clase de baza date putem declara aceasta clasa de baza ca fiind *virtuala.*
Exemplu:

```
#include <iostream.h>
class baza
{
        public:
        int i;
};
class derivata1:virtual public baza
{
        public:
        int j;
};
class derivata2:virtual public baza
{
        public:
        int k;
};
class derivata3:public derivata1,public derivata2
{
        public:
        int suma;
};
void main()
{
        derivata3 obiect;
        obiect.i=10;                    //….it’s ok?? But without virtual base??
        obiect.j=20;
        obiect.k=30;
        obiect.suma=obiect.i+obiect.j+obiect.k;
        cout<<"suma lui "<<obiect.i<<" cu "<<obiect.j<<" si cu "<<obiect.k<<" este
"<<obiect.suma<<endl;
}
```

Daca derivarea lui “derivata1” si a lui “derivata2” nu era virtual atunci aveam erori de ambiguitate in momentul cand doream sa dam valoare lui i <la linia 3 din functia main>.

Functii virtuale

Atunci cand clasele din programul nostru mostenesc o functie virtuala, functia mostenita mentine atributul virtual al functiei de baza. Adica, o functie virtuala va ramane tot virtuala indiferent de cate ori va fii mostenita.
Atunci cand o clasa mosteneste metodele altei clase, se poate intampla ca numele membrilor claselor sa fie in conflict. Atunci cand modificam pointerul clasei de baza pentru a indica clasele derivate, accesarea metodei funct( ) determina executia functiilor locale pe care le defineste fiecare clasa.
Exemplu:

```
#include <iostream.h>
class baza
{
        public:
        virtual void funct()
        {
                cout<<"Functia clasei de baza "<<endl;
        }
};
class derivata1:public baza
{
        public:
        void funct()
        {
                cout<<" Functia clasei derivata1 "<<endl;
        }
};
class derivata2:public derivata1
{
        public:
        void funct()
        {
                cout<<"Functia clasei derivata2 "<<endl;
        }
};

void main()
{
        baza *p, b;
        derivata1 d1;
        derivata2 d2;
        p=&b;      //ce indica???
        p->funct();   //care dintre functii se va apela?
        p=&d1;
        p->funct();  //??
        p=&d2;
        p->funct();   //??
}
```

Ce se intampla daca functia funct( ) din clasa de baza nu e virtuala???

Functii virtuale pure

O *functie virtuala pura* este o functie pe care o declaram in clasa de baza si care impune clasei derivate sa detina o implementare a sa. Prototip:
virtual tip nume_functie(parametri)=0;
Exemplu:

```
#include <iostream.h>
class baza
{
	public:
	virtual void funct()
	{
		cout<<" Clasa de baza "<<endl;
	}
	virtual void functv(void)=0;
};
class derivata:public baza
{
	public:
	void funct()
	{
		cout<<" :) "<<endl;
	}
	virtual void functv(void)
	{
		cout<<"functie virtuala pura"<<endl;
	}
};
void main()
{
	baza *p=new derivata;
	p->funct();
	p->functv();
}
```

Clase abstracte

Atunci cand o clasa contine cel putin o functie virtuala pura spunem ca acea clasa e *abstracta.*
Limbajul C++ nu va va permite sa creati o variabila al carei tip sa fie o clasa abstracta.
O clasa care are ca membrii doar functii virtuale pure se numeste *interfata.*

Polimorfism

Este capacitatea unui obiect de a lua mai multe forme. Acest lucru este posibil in C++ prin intermediul functiilor virtuale. Acelasi pointer poate indica diferite clase pentru a realiza diferite operatii.
Exemplu:

```
#include <iostream.h>
#include <string.h>
class baza
{
        public:
        virtual int add(int a,int b){ cout<<"add din baza "<<endl; return (a+b); }
   virtual int scad(int a,int b){ cout<<" scad din baza"<<endl; return (a-b); }
        virtual int inmult(int a,int b){ cout<<"inmult din baza"<<endl; return (a*b); }
};
class derivata1:public baza
{
int inmult(int a,int b)
{
        cout<<" inmult din derivata1"<<endl;
        return (a*b);
}
};
class derivata2:public baza
{
        int scad(int a, int b)
        {
                cout<<"scad din derivata2"<<endl;
                return (abs(a-b));
        }
};

void main()
{
        baza *p=new derivata1;
        cout<<p->add(300,299)<<"  "<<p->scad(799,200)<<"  "<<p->inmult(599,1)<<endl;
        p=new derivata2;
        cout<<p->add(222,111)<<"  "<<p->scad(18,288)<<"  "<<p->inmult(398,2)<<endl;
}
```

Ierarhii de clase
Probleme:

1. Implementati o ierarhie de clase care sa permita lucrul cu figurile geometrice. Respectati urmatoarele:

-Definiti o clasa abstracta Figura. O Figura e caracterizata prin nume si are o metoda GetName( ) pentru a obtine numele. De asemenea clasa Figura mai are o metoda abstracta GetArea( ) care va returna aria sa;

-Definiti o clasa Cerc, derivata din clasa Figura. Clasa Cerc va avea un constructor care va primi numele unui cerc, coordonatele centrului si o raza. Clasa Cerc va avea o metoda care va supraincarca metoda GetArea( ) din clasa Figura;

-Definiti inca o clasa abstracta FiguraPoligonala, derivata din clasa Figura. Clasa FiguraPoligonala va defini o interfata pentru a lucra cu figuri construite din puncte si segmente. Clasa FiguraPoligonala va avea o metoda abstracta numita GetPointCount( )

pentru citirea numarului de puncte al figurii. Clasa FiguraPoligonala va supraincarca operatorul de indexare []. Se va declara virtuala functia operator[] in clasa FiguraPoligonala;

-Definiti o clasa Triunghi, derivata din clasa FiguraPoligonala. Clasa Triunghi va avea un constructor care va primi numele triunghiului si coordonatele a trei puncte ale triunghiului. Clasa Triunghi va supraincarca metodele GetArea( ) si GetPointsCount( ) si operatorul [ ];

-Definiti o clasa Dreptunghi, derivata din clasa FiguraPoligonala. Clasa Dreptunghi va avea un constructor care va primi numele dreptunghiului si coordonatele a doua puncte (colturi) opuse ale dreptunghiului. Clasa Dreptunghi va supraincarca metodele GetArea( ), GetPointCount( ) si operatorul [ ];

-Pentru lucrul cu puncte definiti o clasa Punct care va retine coordonatele unui punct. Clasa Punct va avea un constructor care va primi coordonatele unui punct. Construiti in clasa Punct doua metode GetX( ) si GetY( ) pentru a citi coordonatele. Adaugati de asemenea metoda DistanceTo(Punct& other) care va returna distanta de la un punct la alt punct. Utilizati clasa Punct in clasele Cerc, Triunghi si Dreptunghi (relatie de compozitie);

Functia main va arata in modul urmator:

```
int main()
{
Cerc c("C", 2.5, 2.5, 2.5);
Triunghi t("ABC", 0,0, 5,0, 0,5);
Dreptunghi r("PQRS",0,0,5,5);
int i;
cout << "--=== FIGURI ===---" << endl;
Figura* figura[3];
figura[0] = &c;
figura[1] = &t;
figura[2] = &r;
for (i=0; i < 3; i++)
cout<<figura[i]->GetName()<< ": aria="<<figura[i]<<endl;
cout<< "--=== FIGURI POLIGONALE ===---"<<endl;
FiguraPoligonala* figpoli[2];
figpoli[0] = &t;
figpoli[1] = &r;
for (i=0; i < 2; i++)
{
cout<<figpoli[i]->GetName()<<": aria="<<figpoli[i]->GetArea()<<", numar
puncte="<<figpoli[i]->GetPointCount()<<",puncte:";
for (int j=0; j<figpoli[i]->GetPointCount(); j++)
cout<<" "<<(*figpoli[i])[j];
cout<<endl;
}
return 0;
}
```

Programul trebuie sa afiseze:

```
---=== FIGURI ===---
C: aria=45.3416
ABC: aria=12.5
PQRS: aria=35.3553

---=== FIGURI POLIGONALE ===---
ABC: aria=12.5, numar puncte=3, puncte: (0, 0) (5, 0) (0, 5)
PQRS: aria=35.3553, numar puncte=4, puncte: (0, 0) (0, 5) (5, 5) (5, 0)
```

Sugestii:

-Cand implementati operatorul de indexare (*operator[]*) pentru clasa *Triunghi* si pentru clasa *Dreptunghi*, trebuie sa tratati situatia cand avem un index invalid. De exemplu in clasa Triunghi are sens doar daca avem una din valorile indexate 0, 1 sau 2, deoarece putem avea doar 3 puncte pentru triunghi. De asemenea pentru clasa Dreptunghi are sens doar daca trecem ca valori indexate 0, 1, 2 sau 3, deoarece avem doar patru puncte intr-un dreptunghi. Definiti un membru static de tipul Punct in clasa FiguraPoligonala. Apelati membrul BAD_POINT si initializati-l cu orice valoare considerati (de exemplu un punct cu coordonate negative). Apoi, cand supraincarcati operatorul de indexare [], daca valoarea indexata este invalida, returnati o referinta la BAD_POINT. Apoi faceti membrul BAD_POINT sa fie protected, ca sa poata fi accesat din clase derivate.

-Construiti mai intai diagrama de clase si apoi implementati

Ierarhia de Clase: (diagrama UML)