```
                    fl>>str.s;
                    return fl;
                    }
        };

void main(void) // testarea clasei string
        {
        string s1("string-ul 1"),s2,s;
        char st[100];

        s2.set("string-ul 2");
        s=s1+s2;
        cout<<"Concatenarea celor doua string-uri: "<<s<<endl;
        s+=s1;
        cout<<"Concatenarea celor doua string-uri: "<<s<<endl;
        cout<<"Lungimea string-ului:               "<<!s<<endl;
        cout<<"Pe pozitia 5 se afla caracterul:    "<<s[4]<<endl;
        if (s1==s2) cout<<"String-urile sunt identice"<<endl;
        else cout<<"String-urile difera"<<endl;
        if (s1<s2) cout<<"s1 < s2"<<endl;
        s.get(st);
        cout<<"String-ul extras:                   "<<st<<endl;
        }
```

Constructorul de copiere se apelează când se transmite un obiect de tip *string* prin valoare (vezi operatorii + şi <). De asemenea, de fiecare dată când se returnează un obiect de tip *string* (vezi operatorii +, = şi +=), este apelat constructorul de copiere definit în interiorul clasei, care spune modul în care se face efectiv copierea (cu alocările şi eliberările de memorie aferente).

Pentru a vedea efectiv traseul de execuţie pentru programul de mai sus, propunem cititorului rularea acestuia pas cu pas. Rularea pas cu pas în mediul de programare Borland se face cu ajutorul butonului F7 sau F8. Lăsarea liberă a execuţiei programului până se ajunge la linia curentă (pe care se află cursorul) se face apăsând butonul F4. Pentru ca să fie posibilă urmărirea execuţiei programului, în meniul *Options*, la *Debugger*, trebuie bifat *On* în *Source Debugging*. În Visual C++ urmărirea execuţiei pas cu pas a programului se face apasand butonul F11, iar lăsarea liberă a execuţiei programului până la linia curentă se face apăsând Ctrl+F10.

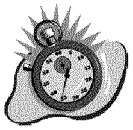
Lăsăm plăcerea cititorului de a completa alte şi funcţii în clasa *string* cum ar fi pentru căutarea unui string în alt string, înlocuirea unui şir de caractere într-un string cu un alt şir de caractere, extragerea unui subşir de caractere dintr-un string etc.

Unitatea de învăţare M2.U5. Moştenirea în C++, funcţii virtuale, destructori virtuali, funcţii pur virtuale

Cuprins

Obiectivele unității de învățare

Obiectivul principal al acestui capitol este modul în care se se implementează moştenirea în C++.



Durata medie de parcurgere a unității de învățare este de 2 ore.

U5.1. Moştenirea în C++

În C++ o clasă poate sa nu fie derivată din nici o altă clasă, sau poate deriva una sau mai multe clase de bază:

```
class nume_clasa [ : [virtual] [public/protected/private] clasa_baza1,
                   [virtual] [public/protected/private] clasa_baza2, ... ]
{
        // ...
};
```

În funcție de specificarea modului de derivare (*public*, *protected* sau *private*), accesul la datele şi metodele clasei de bază este restricționat mai mult sau mai puțin în clasa derivată (cel mai puțin prin *public* şi cel mai mult prin *private*).

Specificarea modului de derivare este opțională. Implicit, se ia, ca şi la definirea datelor şi metodelor interne clasei, modul *private* (în lipsa specificării unuia de către programator).

În continuare prezentăm într-un tabel efectul pe care îl au specificatorii modului de derivare asupra datelor şi metodelor clasei de baza în clasa derivată:

Tip date şi metode din clasa de bază	Specificator mod de derivare	Modul în care sunt văzute în clasa derivată datele şi metodele clasei de bază
public	public	public
public	protected	protected
public	private	protected
protected	public	protected
protected	protected	protected
protected	private	protected
private	public	private
private	protected	private
private	private	private

După cum se poate vedea din tabelul de mai sus pentru a avea acces cât mai mare la membrii clasei de bază este indicată folosirea specificatorului *public* în momentul derivării.

Constructorul unei clase derivate poate apela constructorul clasei de bază, creându-se în memorie un obiect al clasei de bază (denumit **sub-obiect** al clasei derivate), care este văzut ca o

particularizare a obiectului clasei de bază la un obiect de tipul clasei derivate. Apelul constructorului clasei de bază se face astfel:

```
class deriv: public baza
        {
        // ....
        deriv(parametri_constructor_deriv) :baza(parametri_constructor_baza)
                {
                // ....
                }
        // ....
        };
```

Parametrii constructorului *baza* (la apelul din clasa derivată) se dau în funcție de parametrii constructorului *deriv*. De exemplu, pentru o clasă *patrat* derivată dintr-o clasă *dreptunghi* (ambele cu laturile paralele cu axele de coordonate), apelul constructorului *dreptunghi* la definirea constructorului *patrat* se poate face astfel:

```
class patrat: public dreptunghi
{
private:
        float x,y,l;
public:
        patrat(float X, float Y, float L): public dreptunghi(X, Y, X+L, Y+L)
        {
                x=X;
                y=Y;
                l=L;
        }
};
```

Dreptunghiul din exemplul de mai sus este definit prin coordonatele a două vârfuri diagonal opuse, iar pătratul prin coordonatele vârfului stânga-sus şi prin lungimea laturii sale. De aceea, pătratul este văzut ca un dreptunghi particular, având cele două vârfuri diagonal opuse de coordonate *(X,Y)*, respectiv *(X+L,Y+L)*.

Asupra moştenirii multiple o să revenim după ce introducem noțiunea de *virtual*.

U5.2. Funcții virtuale

În clase diferite în cadrul unei ierarhii pot apărea funcții cu aceeaşi semnătură (acelaşi nume şi aceiaşi parametri), în engleză *overriding*. Astfel, putem avea situația în care într-o clasă derivată există mai multe funcții cu aceeaşi semnatură (unele moştenite din clasele de pe nivele superioare ale ierarhiei şi eventual una din clasa derivată). În această situație se pune problema cărei dintre aceste funcții se va răsfrânge apelul dintr-un obiect alocat static al clasei derivate. Regula este că se apelează funcția din clasa derivată (dacă există), iar dacă nu există o functie în clasa derivată, atunci se caută funcția de jos în sus în ierarhie. Dacă dorim să apelăm o anumită funcție de pe un anumit nivel, atunci folosim operatorul de rezoluție pentru a specifica din ce clasă face parte funcția dorită:

```
clasa::functie(parametri_de_apel);
```

După cum putem vedea, problemele sunt rezolvate într-o manieră elegantă atunci când se lucrează cu obiecte alocate static.

Dacă lucrăm însă cu pointeri către clasă, problemele se complică. Putem defini un pointer *p* către clasa de baza *B* care reține adresa unui obiect dintr-o clasă derivată *D*. Când se apelează o funcție sub forma *p.functie(...)*, funcția este căutată mai întâi în clasa *B* către care este definit pointerul *p*, ci nu în clasa *D* aşa cum ne-am putea aştepta. Mai mult, dacă funcția există în clasa *D* şi nu există în *B*, vom obține eroare la compilare. De fapt, pointerul *p* reține adresa către subobiectul din clasa *B*, construit odată cu obiectul clasei derivate *D*, din cauza că *p* este un pointer către clasa *B*.

Iată în continuare situația descrisă mai sus:

```
#include<iostream.h>

class B
{
public:
     B()
     {
          cout<<"Constructor clasa de baza"<<endl;
     }
     void functie()
     {
          cout<<"Functie clasa de baza"<<endl;
     }
};

class D:public B
{
public:
     int x;
     D()
     {
          cout<<"Constructor clasa derivata"<<endl;
     }
     void functie()
     {
          cout<<"Functie clasa derivata"<<endl;
     }
};

void main()
{
     B *p=new D;
     p->functie();
}
```

După cum am văzut, este evident că pe ecran în urma execuției programului de mai sus vor apărea mesajele:

```
Constructor clasa de baza
Constructor clasa derivata
Functie clasa de baza
```

Dacă *functie* nu era implementată în clasa de baza *B*, obţineam eroare la compilare pentru că pointerul *p* reţine adresa subobiectului şi este evident că în această situaţie la adresa indicată de *p* nu există nimic referitor la clasa *D*. Din aceleaşi considerente, dacă încercăm referirea la campul `x` sub forma `p->x`, vom obţine de asemenea eroare la compilare.

Există posibilitatea ca o funcţie membră unei clase să fie declarată ca fiind virtuală.

```
virtual tip_ret functie_membra(parametri)
```

Să precizăm şi faptul că numai funcţiile membre unei clase pot fi declarate ca fiind virtuale.

Declaraţia unei funcţii din clasa de bază ca fiind virtuale se adresează situaţiilor de tipul celei de mai sus (clasa *D* este derivată din *B* şi `B *p=new D;`). Astfel, dacă în faţa declaraţiei metodei *functie* din clasa *B* punem cuvântul rezervat **virtual**, atunci în urma execuţie programului de mai sus pe ecran se vor afişa mesajele:

```
Constructor clasa de baza
Constructor clasa derivata
Functie clasa derivata
```

Deci, declaraţia virtual a funcţiei din clasa de bază a ajutat la identificarea corectă a apelului funcţiei din clasa derivată.

Să vedem ce se întâmplă de fapt atunci când declarăm o funcţie ca fiind virtuală.

Cuvântul cheie **virtual** precede o metodă a unei clase şi semnalează că, dacă o funcţie este definită într-o clasă derivată, aceasta trebuie apelată prin intermediul unui pointer. Compilatorul C++ construieşte un tabel în memorie denumit tablou virtual (în engleză *Virtual Memory Table* – **VMT**) cu pointerii la funcţiile virtuale pentru fiecare clasă. Fiecare instanţă a unei clase are un pointer către tabelul virtual propriu. Cu ajutorul acestei reguli compilatorul C++ poate realiza legarea dinamică între apelul funcţiei virtuale şi un apel indirect prin intermediul unui pointer din tabelul virtual al clasei. Putem suprima mecanismul apelării unei funcţii virtuale explicitând clasa din care face parte funcţia care este apelată folosind operatorul de rezoluţie.

În C++, în cadrul unei ierarhii nu pot apărea funcţii virtuale cu acelaşi nume şi aceeaşi parametri, dar cu tip returnat diferit. Dacă încercăm să scriem astfel de funcţii este semnalată eroare la compilare. Astfel, dacă în exemplul de mai sus metoda *functie* din clasa de bază ar fi avut tipul returnat *int* şi era virtuală, obţineam eroare la compilare. Putem avea însă într-o ierarhie funcţii care nu sunt virtuale, cu acelaşi nume şi aceaşi parametri, dar cu tip returnat diferit.

Datorită faptului că apelul unei funcţii virtuale este localizat cu ajutorul tabelei VMT, apelarea funcţiilor virtuale este lentă. De aceea preferăm să declarăm funcţiile ca fiind virtuale numai acolo unde este necesar.

Concluzionând, în final putem spune că declararea funcţiilor membre ca fiind virtuale ajută la implementarea corectă a polimorfismului în C++ şi în situaţiile în care lucrăm cu pointeri către tipul clasă.

U5.3. Destructori virtuali

Constructorii nu pot fi declaraţi virtuali, în schimb destructorii pot fi virtuali.

Să vedem pe un exemplu simplu ce se întâmplă când destructorul clasei de bază este şi respectiv nu este declarat virtual:

```
#include<iostream.h>
```

```
class B
{
public:
     B()
     {
          cout<<"Constructor clasa de baza"<<endl;
     }
     ~B()
     {
          cout<<"Destructor clasa de baza"<<endl;
     }
};

class D:public B
{
public:
     D()
     {
          cout<<"Constructor clasa derivata"<<endl;
     }
     ~D()
     {
          cout<<"Destructor clasa derivata"<<endl;
     }
};

void main()
{
     B *p=new D;
     delete p;
}
```

Pe ecran vor apărea mesajele:

```
Constructor clasa de baza
Constructor clasa derivata
Destructor clasa de baza
```

După cum am văzut, pointerul *p* reține adresa obiectului clasei *B* construit odată cu obiectul clasei *D*. Neexistând nici o legatură cu obiectul clasei derivate, distrugerea se va face numai la adresa *p*, adică se va apela numai destructorul clasei *B*. Pentru a se distruge şi obiectul clasei derivate *D*, trebuie să declarăm destructorul clasei de bază ca fiind virtual. În această situație la execuția programului, pe ecran vor apărea urmatoarele patru mesaje:

```
constructor clasa baza
constructor clasa derivata
destructor clasa derivata
destructor clasa baza
```

Aşadar, este indicat ca într-o ierarhie de clase în care se alocă dinamic memorie destructorii să fie declarați virtuali !

U5.4. Funcţii pur virtuale

Într-o ierarhie întâlnim situaţii în care la un anumit nivel, o funcţie nu este implementată. Cu această situaţie ne întâlnim mai ales în clasele abstracte care pregătesc ierarhia, trasează specificul ei. Funcţiile care nu se implementează pot fi declarate ca fiind *pur virtuale*. Astfel, tentativa de apelare a unei pur virtuale se soldează cu eroare la compilare. În lipsa posibilităţii declarării funcţiilor pur virtuale, în alte limbaje de programare, pentru metodele neimplementate se dau mesaje.

O funcţie pur virtuală se declară ca una virtuală numai că la sfârşit punem *=0*. Compilatorul C++ nu permite instanţierea unei clase care conţine metode pur virtuale. Dacă o clasă *D* derivată dintr-o clasă *B* ce conţine funcţii pur virtuale nu are implementată o funcţie care este pur virtuală în clasa de bază, atunci problema este transferată clasei imediat derivate din *D*, iar clasa *D* la rândul ei devine abstractă, ea nu va putea fi instanţiată.

Dăm în continuare un exemplu în care clasa *patrat* este derivată din clasa *dreptunghi*, care la randul ei este derivată din clasa abstractă *figura*. Pentru fiecare figură dorim să reţinem denumirea ei şi vrem să putem calcula aria şi perimetrul. Dreptunghiul şi pătratul se consideră a fi cu laturile paralele cu axele de coordonate. Dreptunghiul este definit prin coordonatele a două vârfuri diagonal opuse, iar pătratul prin coordonatele unui varf şi prin lungimea laturii sale.

```
# include <math.h>      // pentru functia "fabs"
# include <string.h>    // pentru functia "strcpy"
# include <iostream.h>

class figur // clasa abstracta, cu functii pur virtuale
        {
        protected:
                char nume[20]; // denumirea figurii
        public: // functii pur virtuale
                virtual float arie() = 0;
                virtual float perimetru() = 0;
                char* getnume()
                {
                        return nume;
                }
        };

class dreptunghi: public figura
        {
        protected:
                float x1, y1, x2, y2; // coordonate varfuri diagonal opuse
        public:
                dreptunghi(float X1, float Y1, float X2, float Y2)
                {
                        strcpy(nume,"Dreptunghi");
                        x1=X1;
                        y1=Y1;
                        x2=X2;
                        y2=Y2;
                }
                virtual float arie()
                {
                        return fabs((x2-x1)*(y2-y1));
                }
                virtual float perimetru()
                {
                        return 2*(fabs(x2-x1)+fabs(y2-y1));
```

```
        }
};

class patrat: public dreptunghi
        {
        protected:
                float x, y, l; // coordonate vf. stanga-sus si lungime latura
        public:
                patrat(float X, float Y, float L): dreptunghi(X, Y, X+L, Y+L)
                        { // constructorul patrat apeleaza dreptunghi
                                strcpy(nume,"Patrat");
                                x=X;
                                y=Y;
                                l=L;
                        }
                virtual float perimetru()
                        {
                                return 4*l;
                        }
        };

void main()
{
        dreptunghi d(100,50,200,200);
        patrat p(200,100,80);

        cout<<"Arie      "<<d.getnume()<<": "<<d.arie()<<endl;
        cout<<"Perimetru "<<d.getnume()<<": "<<d.perimetru()<<endl;
        cout<<"Arie      "<<p.getnume()<<": "<<p.arie()<<endl;
        cout<<"Perimetru "<<p.getnume()<<": "<<p.perimetru()<<endl;
}
```

În exemplul de mai sus, în interiorul clasei *patrat*, dacă dorim să apelăm funcția *perimetru* din clasa *dreptunghi* folosim operatorul de rezoluție:

```
dreptunghi :: perimetru();
```

Funcția *perimetru* din clasa *patrat* se poate rescrie folosind funcția *perimetru* din clasa *dreptunghi* astfel:

```
class patrat: public dreptunghi
        {
        // ....
                virtual float perimetru()
                        {
                        return dreptunghi::perimetru();
                        }
        };
```

Să facem câteva observații legate de programul de mai sus:

1. Câmpul *nume* reține denumirea figurii şi este moştenit în clasele *dreptughi* şi *patrat*. Funcția *getnume* returnează numele figurii şi este de asemenea moştenită în clasele derivate.

2. Funcţia *arie* nu este definită şi în clasa *patrat*. În consecinţă, apelul *d.arie()* se va răsfrânge asupra metodei din clasa de baza *dreptunghi*.
3. Clasa *figura* conţine metode pur virtuale, deci este abstractă. Ea trasează specificul ierarhiei (clasele pentru figuri derivate din ea vor trebui să implementeze metodele *arie* şi *perim*). Clasa *figura* nu poate fi instanţiată.

Este evident că în exemplul de mai sus puteam să nu scriem clasa *figura*, câmpul *nume* şi funcţia *getnume* putând fi redactate în interiorul clasei *dreptunghi*.

Definirea clasei abstracte *figura* permite tratarea unitară a conceptului de figură în cadrul ierarhiei. Astfel, tot ceea ce este descendent direct sau indirect al clasei *figura* este caracterizat printr-un nume (care poate fi completat direct şi interogat indirect prin intermediul funcţiei *getnume*) şi două metode (pentru arie şi perimetru).

O să dăm în continuare o posibilă aplicaţie la tratarea unitară a conceptului de figură. Mai întâi însă o să scriem încă o clasă derivată din *figura* – clasa *cerc* caracterizată prin coordonatele centrului şi raza sa:

```
#define PI 3.14159

class cerc: public figura
        {
        protected:
                float x, y, r;
        public:
                cerc(float x, float y, float r)
                {
                        strcpy(nume,"Cerc");
                        this->x=x;
                        this->y=y;
                        this->r=r;
                }
                virtual float arie()
                {
                        return PI*r*r;
                }
                virtual float perimetru()
                {
                        return 2*PI*r;
                }
        };
```

Scriem un program în care construim aleator obiecte de tip figură (dreptunghi, pătrat sau cerc). Ne propunem să sortăm crescator acest vector de figuri după arie şi să afişăm denumirile figurilor în această ordine:

```
void qsort(int s,int d,figura **fig)
{
        int i=s,j=d,m=(s+d)/2;
        figura *figaux;

        do
        {
                while (fig[i]->arie()<fig[m]->arie()) i++;
                while (fig[j]->arie()>fig[m]->arie()) j--;
                if (i<=j)
                {
```

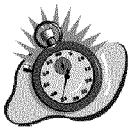
```
                        figaux=fig[i]; fig[i]=fig[j]; fig[j]=figaux;
                        if (i<m) i++;
                        if (j>m) j--;
                    }
                }
                while (i<j);
                if (s<m) qsort(s,m-1,fig);
                if (m<d) qsort(m+1,d,fig);
}

void main(void)
{
        figura **fig;
        int i,n;

        randomize();
        cout<<"Dati numarul de figuri:";
        cin>>n;
        fig = new figura*[n];
        for (i=0;i<n;i++)
                switch (random(3))
                {
                        case 0: fig[i]=new patrat(random(100),random(100)
                                ,random(100)+100);
                                break;
                        case 1: fig[i]=new dreptunghi(random(100),random(100)
                                ,random(100)+100,random(100)+100);
                                break;
                        case 2: fig[i]=new cerc(random(100),random(100)
                                ,random(100)+100);
                                break;
                }
        qsort(0,n-1,fig);
        cout<<"Figurile sortate dupa arie:"<<endl;
        for (i=0;i<n;i++)
                cout<<fig[i]->getnume()<<" cu aria: "<<fig[i]->arie()<<endl;
        for (i=0;i<n;i++) delete [] fig[i];
        delete [] fig;
}
```

Unitatea de învățare M2.U6. Moştenire multiplă, clase virtuale, constructori pentru clase virtuale

Cuprins

Obiectivele unității de învățare

Familiarizarea cu moştenirea multiplă şi cu clasele virtuale din C++ şi cu modul lor de implementare.



Durata medie de parcurgere a unității de învățare este de 2 ore.

U6.1. Moştenire multiplă

Limbajul C++ suportă moştenirea multiplă, adică o clasă D poate fi derivată din mai multe clase de bază B1, B2, … , Bn (n > 0):

```
class D: [mod_derivare] B1, [mod_derivare] B2, ... , [mod_derivare] Bn
        {
        // descriere clasa D
        };
```

Un constructor al unei clase derivate apelează câte un constructor al fiecărei clasei de bază. Apelurile constructorilor claselor de bază sunt despărțite prin virgulă. Iată un exemplu:

```
class Derivata: public Baza1, private Baza2
        {
        Derivata(...):Baza1(...),Baza2(...)
                {
                // descriere constructor Derivata
                }
        };
```

O clasă nu poate avea o clasă de bază de mai multe ori. Deci, nu putem avea spre exemplu o derivare de forma:

```
class Derivata: public Baza1, public Baza2, public Baza1
        {
        // descriere clasa Derivata
        };
```

Ce se întâmplă dacă două clase *B1* şi *B2* sunt baze pentru o clasă *D*, iar *B1* şi *B2* sunt derivate din aceeaşi clasă *C* ? În aceasta situație, aşa cum vom vedea în subcapitolul urmator, clasa *C* se declară în general ca fiind virtuală.

U6.2. Clase virtuale

O clasă *C* se moşteneste virtual dacă poate exista situația ca la un moment dat două clase *B1* şi *B2* să fie baze pentru o aceeaşi clasă derivată *D*, iar *B1* şi *B2* să fie descendente (nu neaparat direct) din aceeaşi clasă *C*.

Dăm un exemplu:

```
class C
{
protected:
        void fct() {}
};

class B1: virtual public C
{
};

class B2: virtual public C
{
};

class D: public B1, public B2
{
public:
        void test()
        {
                fct();
        }
};
```

În exemplul de mai sus, clasa *D* este derivată din *B1* şi *B2*, iar *B1* şi *B2* sunt ambele derivate din clasa *C*. Astfel, facilitatile oferite de clasa *C* (functia *fct*) ajung în *D* de două ori: prin intermediul clasei *B1* şi prin intermediul clasei *B2*. De fapt, orice obiect al clasei *D* va avea în această situaţie două sub-obiecte ale clasei *C*. Din cauză că funcţia *fct* este apelată în interiorul clasei *D* nu se poate decide asupra cărei dintre cele două funcţii din cele două subobiecte se va răsfrânge apelul. Astfel, obţinem eroare la compilare. Pentru a elimina această problema, clasa *C* trebuie să fie declarată virtuală în momentul în care ambele clase, *B1* şi *B2*, derivează pe *C*:

```
class C
{
protected:
        void fct() {}
};

class B1: virtual public C
{
};

class B2: virtual public C
{
};

class D: public B1, public B2
{
public:
        void test()
        {
                fct();
        }
};
```

U6.3. Constructori pentru clase virtuale

Să vedem în ce ordine se apelează constructorii claselor de bază la construcţia unui obiect al clasei derivate. Regula este că întâi se apelează constructorii claselor virtuale (de sus în jos în ierarhie şi de la stânga spre dreapta în ordinea enumerării lor) şi apoi cei din clasele care nu sunt virtuale, evident tot de sus în jos şi de la stânga spre dreapta.

Dacă o clasă este derivată din mai multe clase de bază, atunci clasele de bază nevirtuale sunt declarate primele, aşa încât clasele virtuale să poată fi construite corect. Dăm un exemplu în acest sens:

```
class D: public B1, virtual public B2
      {
      // descriere clasa D
      };
```

În exemplul de mai sus, întâi se va apela constructorul clasei virtuale B2, apoi al clasei B1 şi în final se apelează constructorul clasei derivate D.

Pentru moştenirea multiplă considerăm situaţia în care clasa *patrat* este derivată din clasele *dreptunghi* şi *romb*, iar clasele *dreptunghi* şi *romb* sunt derivate ambele din clasa *patrulater*, care este derivată la randul ei din clasa abstractă *figura*. Dreptunghiul şi pătratul au laturile paralele cu axele de coordonate, iar rombul are două laturi paralele cu axa *Ox*.

```
# include <math.h>
# include <string.h>
# include <stdlib.h>
# include <iostream.h>

# define PI 3.14159

class figura
{
protected:
            char nume[20]; // denumirea figurii
public: // functii pur virtuale
      virtual float arie() = 0;
      virtual float perimetru() = 0;
      char* getnume()
      {
            return nume;
       }
};

class patrulater:public figura
{
protected:
      float x1,y1,x2,y2,x3,y3,x4,y4;
public:
      patrulater(float X1,float Y1,float X2,float Y2,
            float X3,float Y3,float X4,float Y4)
      {
            strcpy(nume,"Patrulater");
            x1=X1;
            y1=Y1;
            x2=X2;
            y2=Y2;
            x3=X3;
            y3=Y3;
            x4=X4;
```

```
            y4=Y4;
        }
        virtual float arie()
        {
            return      fabs(x1*y2-x2*y1+x2*y3-x3*y2+x3*y4-x4*y3+x4*y1-
x1*y4)/2;
        }
        virtual float perimetru()
        {
            return sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1))+
                   sqrt((x3-x2)*(x3-x2)+(y3-y2)*(y3-y2))+
                   sqrt((x4-x3)*(x4-x3)+(y4-y3)*(y4-y3))+
                   sqrt((x1-x4)*(x1-x4)+(y1-y4)*(y1-y4));
        }
};

class dreptunghi: virtual public patrulater // clasa patrulater virtuala
{
protected:
        float x1, y1, x2, y2;
public:
        dreptunghi(float X1,float Y1,float X2,float Y2):
            patrulater(X1,Y1,X2,Y1,X2,Y2,X1,Y2)
        {
            strcpy(nume,"Dreptunghi");
            x1=X1;
            y1=Y1;
            x2=X2;
            y2=Y2;
        }
        virtual float arie()
        {
            return fabs((x2-x1)*(y2-y1));
        }
        virtual float perimetru()
        {
            return 2*(fabs(x2-x1)+fabs(y2-y1));
        }
};

class romb: virtual public patrulater // clasa patrulater virtuala
{
protected:
        float x, y, l, u;
public:
        romb(float X,float Y,float L,float U):

        patrulater(X,Y,X+L,Y,X+L*cos(U)+L,Y+L*sin(U),X+L*cos(U),Y+L*sin(U))
        {
            strcpy(nume,"Romb");
            x=X;
            y=Y;
            l=L;
            u=U;
        }
        virtual float arie()
        {
            return l*l*sin(u);
        }
```

```
        virtual float perimetru()
        {
                return 4*l;
        }
};

class patrat: public dreptunghi, public romb
{
protected:
        float x, y, l;
public:
        patrat(float X,float Y,float L):dreptunghi(X,Y,X+L,Y+L),
                romb(X,Y,L,PI/2),patrulater(X,Y,X+L,Y,X+L,Y+L,X,Y+L)
        {
                strcpy(nume,"Patrat");
                x=X;
                y=Y;
                l=L;
        }
        virtual float arie() // calcul arie patrat ca fiind dreptunghi
        {
                return dreptunghi::arie();
        }
        virtual float perimetru() // calcul perimetru patrat ca fiind romb
        {
                return romb::perimetru();
        }
};

void main(void)
        {
        patrulater P(10,10,100,40,110,100,20,30);
        dreptunghi d(10,20,200,80);
        romb r(20,50,100,PI/3);
        patrat p(20,10,100);

        cout<<"Aria patrulaterului:       "<<P.arie()<<endl;
        cout<<"Perimetrul patrulaterului: "<<P.perimetru()<<endl<<endl;
        cout<<"Aria dreptunghiului:       "<<d.arie()<<endl;
        cout<<"Perimetrul dreptunghiului: "<<d.perimetru()<<endl<<endl;
        cout<<"Aria rombului:             "<<r.arie()<<endl;
        cout<<"Perimetrul rombului:       "<<r.perimetru()<<endl<<endl;
        cout<<"Aria patratului:           "<<p.arie()<<endl;
        cout<<"Perimetrul patratului:     "<<p.perimetru()<<endl<<endl;
        }
```

Observații:

1) Patrulaterul este dat prin coordonatele celor 4 vârfuri ale sale (în sens trigonometric sau invers trigonometric): *(x1,y1)*, *(x2,y2)*, *(x3,y3)* şi *(x4,y4)*.
2) Dreptunghiul (paralel cu axele de coordonate) este considerat ca fiind definit prin două vârfuri diagonal opuse având coordonatele: *(x1,y1)* şi *(x2,y2)*.
3) Rombul (cu două laturi paralele cu abscisa) este caracterizat prin coordonatele vârfului stânga-sus *(x,y)*, lungimea laturii sale *l* şi unghiul din vârful de coordonate *(x,y)* având măsura *u* în radiani.
4) Pătratul (paralel cu axele de coordonate) are vârful stânga-sus de coordonate *(x,y)* şi latura de lungime *l*.

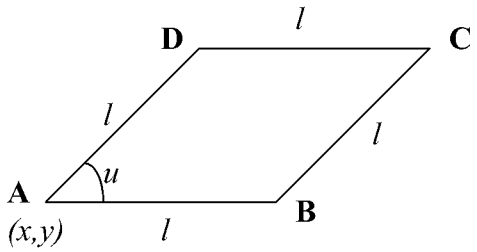
5) Perimetrul patrulaterului e calculat ca suma lungimilor celor 4 laturi:

$$\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}+\sqrt{(x_3-x_2)^2+(y_3-y_2)^2}+$$
$$\sqrt{(x_4-x_3)^2+(y_4-y_3)^2}+\sqrt{(x_1-x_4)^2+(y_1-y_4)^2}.$$

6) Pentru patrulater am folosit o formulă din geometria computațională pentru calculul ariei. Aria unui poligon oarecare $A_1A_2 \ldots A_n$ (în cazul nostru n = 4) este:

$$S_{A_1A_2\ldots A_n}=\frac{\left|\sum_{i=1}^{n} x_i \cdot y_{i+1}-x_{i+1}\cdot y_i\right|}{2},\ unde\ A_i(x_i,y_i)\ (i=1\ldots n)\ si\ A_{n+1}=A_1.$$

7) Pentru a considera un romb ca fiind un patrulater oarecare (vezi constructorul clasei *patrulater* apelat de constructorul clasei *romb*), trebuie să determinăm coordonatele celor 4 vârfuri ale sale.



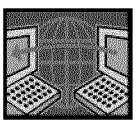
Vârful A are coordonatele *(x,y)*, iar B are *(x+l,y)* (translația lui A pe axa Ox cu *l*). Pentru a găsi coordonatele punctului D am aplicat o rotație a punctului B în jurul lui A cu un unghi *u* în sens trigonometric:

$$\begin{cases} x_D = l\cdot\cos(u)-0\cdot\sin(u)+x = x+l\cdot\cos(u) \\ y_D = l\cdot\sin(u)+0\cdot\cos(u)+y = y+l\cdot\sin(u) \end{cases}.$$

Pentru varful C al rombului am realizat o translație a punctului D pe axa Ox cu *l* şi am obținut coordonatele *(x+l·cos(u)+l, y+l·sin(u))*.

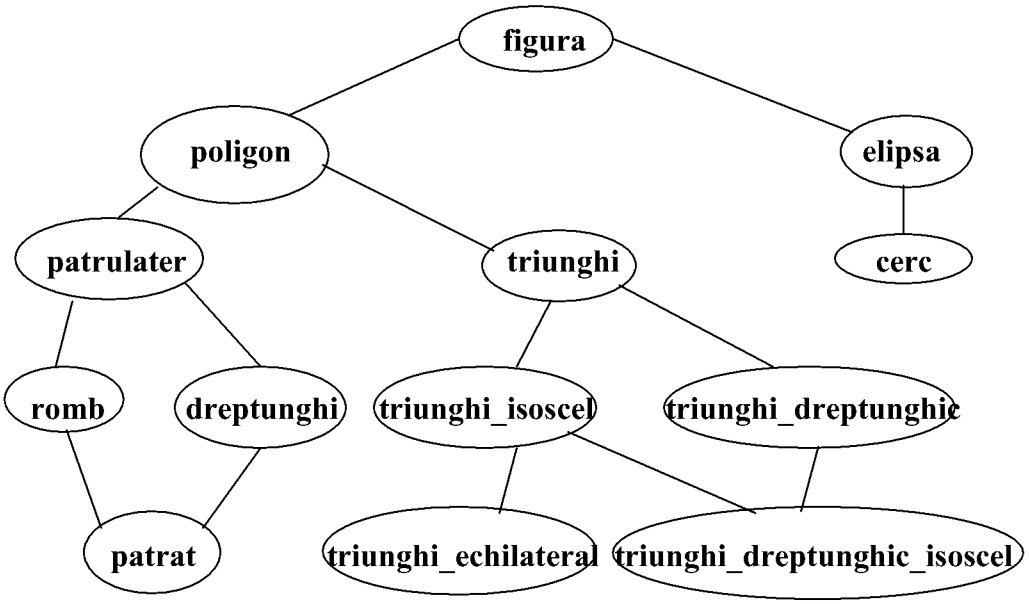
Rezumat

Am văzut până acum cum se scrie cod orientat pe obiecte: cum se scrie o clasă, o metodă, un constructor, destructorul clasei, o funcție prietenă. De asemenea, am făcut cunoştință cu moştenirea din C++ şi problemele pe care le ridică moştenirea multiplă. Am văzut cum se rezolvă elegant aceste probleme cu ajutorul claselor virtuale.



Temă

Lăsăm ca exercițiu cititorului implementarea următoarei ierarhii de figuri:



Unitatea de învățare M2.U7. Clase imbricate, clase şablon

Cuprins

Obiectivele unității de învățare

Familiarizarea cu noțiunea de clasa imbricată, precum şi cu modul de scriere a codului generic orientat pe obiecte în C++.

Durata medie de parcurgere a unității de învățare este de 2 ore.

U7.1. Clase imbricate

C++ permite declararea unei clase în interiorul (*imbricată*) altei clase. Pot exista clase total diferite cu acelaşi nume imbricate în interiorul unor clase diferite.

Dăm un exemplu: