Laboratorul 13. ■ OPERAŢII DE INTRARE/IEŞIRE ÎN C++ ■ CLASE ŞI OBIECTE

13.1. Operaţii de intrare/ieşire cu consola

C++ permite utilizarea sistemului de intrare/ieşire (I/O) din C (tipurile de date şi funcţiile definite în fişierele header stdio.h şi conio.h), dar defineşte şi un sistem propriu de I/O conceput în spiritul POO, mai flexibil şi mai comod.

Sistemul I/O din C++ are la bază conceptul de stream (flux) care reprezintă dispozitive logice folosite în transferul de informaţie între o sursă de date şi destinaţia sa. Sursa şi/sau destinaţia datelor pot fi asociate cu dispozitive fizice sau, pentru formatare în memorie, unor şiruri de caractere. Toate streamurile se comportă la fel, chiar dacă echipamentele fizice la care sunt conectate pot să difere substanţial. Aceleaşi funcţii pot fi folosite pentru scrierea în fişiere, pe ecran sau la imprimantă.

Operaţiile de I/O se realizează cu ajutorul a două ierarhii de clase declarate în fişierele iostream.h şi fstream.h descrise în fig. 13.1.

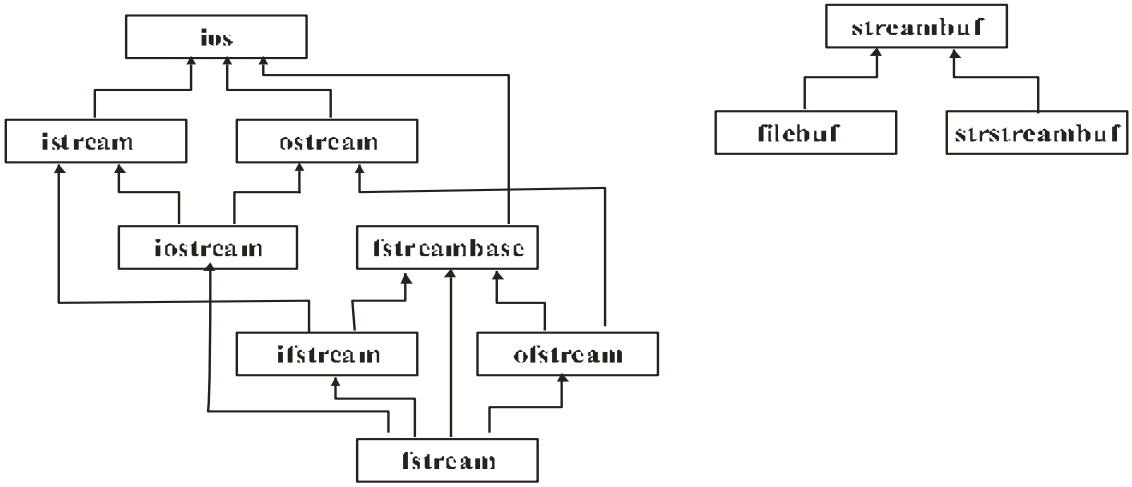

Fig. 13.1. Ierarhiile de clase folosite în operaţiile I/O

La lansarea în execuţie a unui program C++ se deschid patru streamuri predefinite, dispozitive logice de intrare/ieşire standard similare celor din limbajul C, prezentate în Tabelul nr. 13.1.

Tabelul 13.1. Dispozitive logice de intrare/ieşire standard în C++

Stream	Semnificaţie	Echipament implicit	Dispozitiv echivalent în C
cin	intrare standard	tastatură	stdin
cout	ieşire standard	ecran	stdout
cerr	ieşire standard pentru eroare	ecran	stderr
clog	ieşire standard pentru eroare cu memorie tampon	ecran	-

Streamurile standard pot fi redirecţionate spre alte dispozitive fizice sau fişiere.

Pentru operaţiile I/O se folosesc operatorii << şi >> care au fost supradefiniţi pentru operaţii de ieşire şi respectiv de intrare cu streamuri. Utilizarea dispozitivelor şi operatorilor de intrare/ieşire C++ impune includerea fişierului antet **iostream.h**.

Sintaxa folosită este următoarea:

```
cin >> var;          /* citeşte var de la cin */
cout << var;         /* scrie var la cout */
```

```
#include <iostream.h>

void main()
{
        int a;
        cout << "Introdu un intreg:";  // afisarea unui mesaj
        cin >> a;                      // citirea de la tastatura a unei valori pentru variabila a
        cout << "\na = " << a << '\n';
}
```

Operatorul << este operator de ieşire, sau mai este denumit operator de inserţie deoarece introduce caractere într-un stream. Operatorul >> este operator de intrare, sau mai este denumit operator de extragere deoarece extrage caractere dintr-un stream.

Exemplul precedent produce acelaşi efect cu secvenţa următoare care foloseşte funcţiile printf(), scanf(), specifice limbajului C:

```
#include <stdio.h>

void main()
{
        int a;
        printf("Introdu un intreg:");
        scanf("%d", &a);
        printf("\na = %d\n", a);
}
```

Aşa cum se vede, sunt posibile operaţii multiple, cu sintaxa:

```
cin >> var1 >> var2 ... >> varN;
cout << var1 << var2 ... << varN;
```

În acest caz, se efectuează succesiv, de la stânga la dreapta, scrierea la cout, respectiv citirea de la cin a valorilor varl ... varN.

Se observă că, spre deosebire de folosirea funcţiilor printf(), scanf(), la folosirea sistemului I/O oferit de C++, nu a fost necesară precizarea unor formate. Acest lucru se datorează faptului că supradefinirea operatorilor <<, respective >>, este făcută pentru toate tipurile predefinite de date.

Astfel, tipurile datelor transferate către cout pot fi:

- toate tipurile aritmetice;
- şiruri de caractere;
- pointeri de orice tip în afară de char.

Tipurile datelor citite de la cin pot fi:

- toate tipurile aritmetice;
- şiruri de caractere.

Controlul formatului pentru ambele operaţii este posibil, dar nu este obligatoriu deoarece există formate standard. Acestea sunt satisfăcătoare pentru dialogul cu consola efectuat în exemplele din capitolele următoare.

In cazul citirii unui caracter nevalid, citirea este întreruptă şi caracterul rămâne în tamponul de intrare generând probleme similare cu cele care apar la utilizarea funcţiei scanf().

```
#include <iostream.h>

void main()
{   int i;
   char nume[21];
   float r;
   cout < < "introduceti un numar intreg si apoi un numar real: “ ;
   cin>> i >> r;
   cout << "\nAti introdus: "<< i  << "si" << r <<'\n';
   cout << "Introduceti numele dvs: “;
   cin >> nume;
   cout<< "Salut, " < < nume << " !\n” ;
}
```

Pentru afişare se pot utiliza expresii:

```
#include <iostream.h>

void main()
{
  int num;
  cin >> num;
  cout << num + 1;                // se afişează rezultatul returnat de expresia “num+1”
}
```

Pentru citirea/scrierea şirurilor de caractere se poate specifica o constantă şir sau un pointer de caractere. Din acest motiv, pentru afişarea adresei unui şir este necesară o conversie explicită la pointer de alt tip, de exemplu (void *). Valorile adreselor se afişează în hexazecimal.

```
#include <iostream.h>

void main()
{
   char sir[20]=”Sir de caractere”;     // se declară un şir de caractere cu iniţializare
   cout<<sir<<’\n’;                     // se afişează conţinutul sirului de caractere “sir”
   cout<<*sir<<’\n’;                    // se afişează primul caracter din şirul de caractere ”sir”
   cout<<&sir<<’\n’;                    // se afişează adresa la care se află variabila pointer “sir”
   cout<<(void*)sir<<’\n’;              // se afişează adresa la care se află variabila pointer “sir”
   cin>>*sir;                           // se citeşte un alt caracter pentru prima poziţie a şirului
   cout<<sir<<’\n’;                     // se afişează conţinutul şirului de caractere “sir”
   cin>>sir;                            // se citeşte o nouă valoare pentru şirul de caractere
   cout<<sir<<’\n’;                     // se afişează conţinutul şirului de caractere “sir”
```

```
   char * p_sir=”abc”;              // declară un pointer la tipul char ce se inițializează cu
                                    // adresa șirului constant “abc”
   cout<<p_sir<<’\n’;               // se afișează conținutul șirului de caractere către care
                                    // pointează p_sir
   cout<<*p_sir<<’\n’;              // se afișează primul caracter din șirul constant de caractere
   cout<<&p_sir<<’\n’;              // se afișează adresa la care se află variabila pointer p_sir
   cout<<(void*)p_sir<<’\n’;        // se afișează adresa conținută de variabila pointer  p_sir,
                                    // deci adresa la care se află șirul constant “abc”
}
```

Se poate recurge la conversii explicite. Pentru a afișa caracterul corespunzător în codul ASCII a unei valori întregi, se recurge la conversie explicită la int, la fel cum, pentru a afișa caracterul corespunzător unei valori întregi, se face conversie la char, de exemplu:

```
cout << 'A';                // se afiseaza : A
cout << int('A');           // se afiseaza : 65
cout << char(65);           // se afiseaza : A
```

13.2. Clase şi obiecte

Limbajul C++ pune la dispoziţia programatorului posibilitatea programării obiectuale (OOP-Object Oriented Programming). Domeniul programării orientate spre obiecte este deosebit de larg, unificând două componente de bază: datele aplicaţiei şi codul necesar tratării acestora. Se oferă facilitatea definirii unor tipuri de date proprii şi a operatorilor destinaţi manipulării lor, acestea comportându-se asemănător datelor standard şi a operatorilor standard.

Avantajele OOP se pot descrie prin următoarele concepte:

- INCAPSULAREA – prin care se obţine contopirea datelor cu codul în cadrul aşa numitelor clase. Se asigură o bună modularizare şi localizare a eventualelor erori, precum şi o bună protecţie a datelor prin accesul controlat către acestea.
- MOŞTENIREA – care permite ca, pornind de la o clasă definită, cu anumite proprietăţi, care constituie clasa de bază, să se creeze seturi de clase asemănătoare care completează proprietăţile clasei de bază cu noi proprietăţi.
- POLIMORFISMUL – într-o ierarhie de clase obţinute prin moştenire, o metodă poate avea forme diferite de la o clasă la alta, utilizându-se supradefinirea acestora.

Clasa este un tip de date definit de utilizator care asociază unei structuri de date un set de funcţii:

Clasa = Date + Operaţii (metode)

Conceptele de domeniu şi durată de viaţă, variabile locale şi globale, variabile statice, automatice şi dinamice se aplică şi obiectelor.

C++ permite controlul accesului atât la datele membre, cât şi la funcţiile membre unei clase. În acest scop se folosesc specificatorii de control al accesului : **private, protected şi public**.

Efectul acestor specificatori asupra accesului unui membru este:

- **public** : membrul poate fi accesat de orice funcţie din domeniul de declaraţie a clasei;

- **private**: membrul este accesibil numai funcţiilor membre şi prietene clasei;
- **protected**: membrul este accesibil atât funcţiilor membre şi prietene clasei, cât şi funcţiilor membre şi prietene claselor derivate din clasa respectivă.

O funcţie membră a unei clase are acces la toate datele membre ale oricărui obiect din clasa respectivă, indiferent de specificatorul de acces.

În C++ se pot declara mai multe categorii de clase folosind cuvintele cheie: **struct, union**, respectiv **class**. Tipurile class sunt mai frecvent utilizate, ele corespunzând mai fidel conceptului OOP.

13.2.1. Tipul class

Sintaxa generală de declarare a unui tip de date class este similară cu a tipului struct:

class <nume_clasa> <: lista_clase_baza> {<lista_membri>} <lista_variabile>;

unde:

- *nume_clasa* este un identificator care desemnează numele tipului clasă declarat, care trebuie să fie unic în domeniul în care este valabilă declaraţia;
- *lista_clase_baza* este lista claselor din care este derivată clasa declarată (dacă este cazul);
- *lista_membri* reprezintă secvenţa cu declaraţii de datele membre şi declaraţii sau definiţii de funcţii membre; datele membre pot fi de orice tip, mai puţin tipul clasă declarat, dar se admit pointeri către acesta; folosirea specificatorilor auto, extern, register nu este permisă.
- *lista_variabile* este lista variabilelor de tip nume_clasa.

La declararea unei clase este obligatoriu să existe cel puţin una dintre nume_clasa şi lista_variabile. De regulă se specifică nume_clasa, fapt ce permite declararea ulterioară de obiecte din tipul clasă declarat.

Membrii unei clase au implicit atributul de acces private.

În cadrul declaraţiei clasei pot să apară specificatorii de acces de oricâte ori şi în orice ordine, toţi membrii declaraţi după un specificator având accesul controlat de acesta.

În declaraţia clasei se pot include definiţii complete ale funcţiilor membre, sau doar prototipurile funcţiilor membre, definirea acestora putând fi făcută oriunde în proiect, chiar şi în alt fişier.

Pentru definiţiile de funcţii aflate în afara declaraţiei clasei este necesar să se specifice numele clasei urmat de operatorul de rezoluţie (::) alăturat numelui funcţiei. Operatorul indică compilatorului că funcţia respectivă are acelaşi domeniu cu declaraţia clasei respective, fapt ce permite referirea directă a membrilor clasei. În caz contrar, compilatorul consideră că se defineşte o funcţie cu acelaşi nume, externă clasei respective.

Funcţiile membre ale unei clase pot fi supradefinite şi pot avea parametri impliciţi.

Pentru apelul funcţiilor membre publice şi referirea datelor membre cu acces public ale unui obiect, se folosesc operatorii de selecţie (.) şi (->), ca în cazul structurilor şi uniunilor din C.

Pentru exemplificarea declaraţiei unei clase, utilizarea obiectelor de tip clasă şi a membrilor lor, date şi funcţii, se defineşte tipul de date Punct ca şi clasă. Semnificaţia este reprezentarea

punctelor în plan prin coordonate carteziene. Clasa Punct care are ca date membre doi membri de tip int, x şi y, ce primesc ca valori, valorile coordonatelor unui punct. Clasa include ca funcţii membre o funcţie de iniţializare ce stabileşte valorile iniţiale ale coordonatelor punctului, init(), două funcţii ce permit citirea valorilor coordonatelor, getx(), respectiv gety(), funcţia de modificare a coordonatelor punctului, move() şi funcţia de afişare, afisare(), ce afişează proprietăţile punctului.

Declaraţia clasei Punct, cu membrii descrişi anterior, poate fi următoarea:

```
class Punct
{       private:                        // se specifică accesul private la membrii clasei
                int x, y;               // date membre ale clasei
        public:                         // se specifică acces public la membrii clasei
                void init(int, int);    // funcţie de iniţializare a coordonatelor
                int getx();             // funcţia returnează valoarea membrului x
                int gety();             // funcţia returnează valoarea membrului y
                void move(int, int);    // funcţie membră cu parametri
                void afisare()          // funcţie de afişare a valorilor membrilor
};
```

Aşa cum s-a precizat, funcţiile membre ale unei clase pot fi definite în declaraţia clasei sau în exteriorul său, pot avea parametri neimpliciţi sau impliciţi. Astfel, declaraţia clasei poate fi:

```
class Punct
{       private:                        // se specifică accesul private la membrii clasei
                int x, y;               // date membre ale clasei
        public:                         // se specifică acces public la membrii clasei
                void init(int initx=0, int inity=0)     // funcţie de iniţializare, funcţie membră cu
                {       x = initx;                      // parametri impliciţi
                        y = inity;
                }
                int getx()              // funcţie inline, returnează valoarea membrului x
                {       return x;
                }
                int gety()              // funcţie inline, returnează valoarea membrului y
                {       return y;
                }
                void move(int dx, int dy);      // funcţie membră cu parametri, definită în afara
                                        // declaraţiei clasei
                void afisare()          // funcţie de afişare a valorilor membrilor, funcţie inline
                {       cout<<"\nx=<<x<<"\tz="<<y";
                }
};
```

Funcţia membră move() se defineşte în afara declaraţiei clasei, folosind sintaxa următoare:

```
void Punct::move(int dx, int dy)        // definirea funcţiei move(), membră a clasei Punct
{  x+=dx;
   y+=dy;
}
```

Obiectele de tip Punct se declară, precizând tipul şi numele lor, de exemplu:

```
Punct Punct1;                           // se declară un obiect de tip Punct, Punct1
```

Obiectul Punct1 este alocat în memorie, în funcţie de locul în care este făcută declaraţia, în segmentul de date dacă declaraţia este globală, situaţie în care membrii date sunt iniţializaţi implicit cu 0, sau pe stivă, dacă declaraţia este locală unei funcţii, situaţie în care membrii date preiau valori reziduale.

În continuare este exemplificat modul de utilizare a membrilor obiectelor de tip Punct:

```
void main()
{  Punct Punct1;                          // se declară un obiect de tip Punct, Punct1
   int x1, y1;
   cout<<"\n Introduceti coordonata x= ";
   cin>>x1;
   cout<<" Introduceti coordonata y= ";
   cin>>y1;

   Punct1.init(x1, y1);                   // iniţializarea obiectului Punct1 cu valorile x1, respectiv y1
   cout<<"\n x este = "<<Punct1.getx();    // afişarea coordonatei x
   cout<<"\n y este = "<< Punct1.gety();   // afişarea coordonatei y
   Punct1.move(10, 20);                   // modificarea coordonatelor obiectului Punct1
   cout<<"\n x este = "<<Punct1.getx();    // afişarea coordonatei x
   cout<<"\n y este = "<< Punct1.gety();   // afişarea coordonatei y
   Punct Punct2;                          // se declară un obiect de tip Punct, Punct2
   Punct2.init();                         // membrii x şi y preiau valorile implicite ale
                                          // parametrilor, deci x=y=0
   Punct2.afisare();                      // afişarea caracteristicilor obiectului Punct2
   Punct *p_Punct3;                       // se declară un pointer la Punct care poate prelua
                                          // adresa unui obiect de tip Punct
   p_Punct3 = &Punct2;
   p_Punct3 -> move ( 5, 12);             // apelul funcţiilor membre se face folosind
                                          // operatorul de selecţie "->"
   p_Punct3 -> afisare();
}
```

Obiectele declarate de tip Punct (Punct1, Punct2) sunt structuri de date alcătuite din doi membri int x şi respectiv y, care pot fi controlaţi cu funcţiile asociate care asigură atribuirea de valori, afişarea, modificarea acestora (init(), afisare(), getx(), gety(), move()).

Accesul la membrii x şi y nu se poate face din afara clasei, ci doar prin funcţiile membre, datorită accesului restricţionat la aceştia prin specificatorul private.

```
   Punct1.x = 15;                 // eroare, membrul Punct1.x este privat
   Punct2.y = Punct1.x;           // eroare Punct1.x, Punct2.y sunt membri privaţi
```

Pentru fiecare obiect al clasei se alocă spaţiul de memorie necesar datelor membre. Pentru funcţiile membre, în memorie există codul într-un singur exemplar, cu excepţia funcţiilor inline pentru care se inserează codul executabil pentru fiecare apel în parte.

Operatorul de atribuire (=) poate fi folosit pentru obiecte din acelaşi tip class, determinând copierea datelor membru cu membru, ca în cazul structurilor din C.

```
Punct2=Punct1;          //membrul x al obiectului Punct2 primeşte valoarea membrului x al
                        // obiectului Punct1 membrul y al obiectului Punct2 primeşte valoarea
                        // membrului y al obiectului Punct1
Punct2.afisare();
```

Se pot folosi operatorii new şi delete pentru crearea/distrugerea de obiecte de tip class.

```
Punct * p_Punct4=new Punct;
p_Punct4 -> init(5,7);
p_Punct4-> move(2, 2);
p_Punct4-> afisare();
delete p_Punct4;
```

13.2.2. Autoreferinţa. Cuvântul cheie “this”

În definiţiile funcţiilor membre sunt necesare referiri la datele membre ale clasei respective. Acest lucru se poate face fără a specifica un obiect anume. La apelarea unei funcţii membre, identitatea obiectului asupra căruia se acţionează este cunoscută datorită transferului unui parametru implicit care reprezintă adresa obiectului.

Dacă în definiţia unei funcţii este necesară utilizarea adresei obiectului, acest lucru se realizează cu cuvântul cheie **this**, asociat unui pointer către obiectul pentru care s-a apelat funcţia.

Clasa Punct definită anterior se poate completa cu funcţia membră cu prototipul:

void adresa();

definită astfel:

```
void Punct::adresa()
{
cout << ”\n Adresa obiectului Punct este:”
cout << this;
}
```

Funcţia main() se poate completa cu următoarele linii de program:

```
Punct1.adresa();
Punct2.adresa();
p_Punct3->adresa();
```

Exemplu

/**

Exemplul 13.1. Se defineşte o clasă pentru reprezentarea tablourilor de date de tip double. Metodele clasei sunt:

init(); - initializează elementele tabloului

void citire(); - citeşte de la tastatură valorile elementelor tabloului

void afisare(); - afişează valorile elementelor tabloului

void modif_el(int, double); - modifică valoarea unui element

double media(); - calculează şi returnează media aritmetică a elementelor tabloului

**/

```
#include <iostream.h>
#include <conio.h>
#define N 8                // dimensiunea tablourilor reprezentate
```

```
class tablou
{       double tab[N];          // tabloul de date double
public:
        init();
        void citire();
        void afisare();
        void modif_el(int, double);
        double media();
};

tablou::init()
{  for (int i=0; i<N; i++)
        tab[i]=0;
}
void tablou::citire()
{  for(int i=0; i<N; i++)
   {    cout<<"tab["<<i<<"]=";
        cin>>tab[i];
   }
}
void tablou::afisare()
{ cout<<endl;
  for(int i=0; i<N; i++)
        cout<<"tab["<<i<<"]="<<tab[i]<<endl;
}
void tablou::modif_el(int i, double val)
{ if (i<N)
        tab[i]=val;
}
double tablou::media()
{  double med=0;
  for(int i=0; i<N; i++)
        med+=tab[i];
  med/=N;
  return med;
}
void main()
{ tablou t1;            // declaratia unui obiect de tip tablou
  double media;
  t1.afisare();         // se afiseaza elementele tabloului – se vor afisa valori reziduale
  getch();
  t1.init();            // se initializeaza cu 0 elementele tabloului
  t1.afisare();         // se afiseaza elementele tabloului – prin initializare au primit valoarea 0
  getch();
  t1.citire();          // se citesc valori de la tastatură pentru elementele tabloului
  t1.afisare();         // se afiseaza elementele tabloului
  getch();
  media=t1.media();     // se calculeaza media aritmetica a elementelor tabloului
  cout <<"\nMedia aritmetica a elementelor tabloului 1 este: "<<media<<endl;
  getch();
  tablou t2 = t1;       // se declară obiectul t2 de tip tablou; prin atribuire, copiază valorile
                        // elementelor obiectului t1
  t2.afisare();         // se afiseaza valorile elementelor obiectului t2
  getch();
```

```
  t1.modif_el(3,7.5);   // se modifica elementul de index 3, atribuindu-i valoarea 7.5
  t1.afisare();         // se afiseaza valorile elementelor obiectului t2
  tablou *p;            // se declara un pointer de tip tablou
  p = &t1;              // p preia adresa obiectului t1
  p->afisare();         // se afiseaza valorile elementelor obiectului t1, prin pointerul p
  getch();
}
```

13.2.3.Constructori şi destructori

Unele obiecte necesită alocarea unor variabile dinamice la creare, eventual atribuirea de valori de iniţializare adecvate datelor înainte de utilizare. Pe de altă parte, eliminarea unor obiecte impune efectuarea unor operaţii, cum ar fi eliberarea zonei de memorie alocată dinamic.

Pentru crearea, iniţializarea, copierea şi distrugerea obiectelor, în C++ se folosesc funcţii membre speciale, numite constructori şi destructori.

Constructorul se apelează automat la crearea fiecărui obiect al clasei, indiferent dacă este static, automatic sau dinamic (creat cu operatorul new), inclusiv pentru obiecte temporare.

Destructorul se apelează automat la eliminarea unui obiect, la încheierea timpului de viaţă sau, în cazul obiectelor dinamice, cu operatorul delete.

Aceste funcţii efectuează operaţiile prealabile utilizării obiectelor create, respectiv eliminării lor. Alocarea şi eliberarea memoriei necesare datelor membre rămâne în sarcina compilatorului.

Constructorul este apelat după alocarea memoriei necesare datelor membre ale obiectului (în faza finală creării obiectului).

Destructorul este apelat înaintea eliberării memoriei asociate datelor membre ale obiectului (în faza iniţială a eliminării obiectului).

Constructorii şi destructorii se deosebesc de celelalte funcţii membre prin câteva caracteristici specifice:

- numele funcţiilor constructor sau destructor este identic cu numele clasei căreia îi aparţin; numele destructorului este precedat de caracterul tilde (~);
- la declararea şi definirea funcţiilor constructor sau destructor nu se specifică nici un tip returnat (nici măcar tipul void);
- nu pot fi moşteniţi, dar pot fi apelaţi de clasele derivate;
- nu se pot utiliza pointeri către funcţiile constructor sau destructor;
- constructorii pot avea parametri, inclusiv parametri impliciţi şi se pot supradefini; destructorul nu poate avea parametri şi este unic pentru o clasă, indiferent câţi constructori au fost declaraţi.

Constructorul fără parametri se numeşte constructor implicit.

Dacă o clasă nu are constructori şi destructor definiţi, compilatorul generează automat un constructor implicit, respectiv un destructor implicit, care sunt funcţii publice.

Constructorii pot fi publici sau privaţi, dar, de regulă, se declară cu acces public. Dacă se declară constructorul cu acces private, nu se pot crea obiecte de acel tip.

Se consideră clasa Punct. Clasa conţine o funcţie membră, init(), care, apelată, iniţializează datele membre ale unui obiect de tip Punct. Problema este că, grija apelului acestei funcţii revine

utilizatorului. În cazul în care datele membre nu sunt iniţializate, se poate opera cu obiecte cu valori necontrolate.

Pentru a evita astfel de situaţii, se poate asigura iniţializarea automată a obiectelor de tip Punct de la declarare, prin definirea de constructori care să preia operaţiile efectuate de funcţia init(). Clasa Punct poate fi definită astfel:

```
#include <iostream.h>
#include <conio.h>
class Punct
        {           private:
                            int x, y;
                    public:
                            Punct();                // constructor implicit
                            Punct(int, int);        // constructor cu parametri
                            Punct(Punct&);          // constructor de copiere
                            ~Punct();               // destructor
                            int getx();
                            int gety();
                            void move(int,int);
                            void afisare();
        };
Punct::Punct()
{       x=0;
        y=0;
        cout<<"\napel constructor implicit: x="<<x<<", y="<<y<<", adresa de
memorie:"<< this<<endl;
}
Punct::Punct(int abs, int ord)
{       x=abs;
        y=ord;
        cout<<"\napel constructor cu parametri x="<<x<<", y="<<y<<", adresa de
memorie:"<<this<<endl;
}
Punct::Punct(Punct& p)
{       x=p.x;
        y=p.y;
        cout<<"\napel constructor de copiere x="<<x<<", y="<<y<<", adresa de
memorie:"<<this<<endl;
}
Punct::~Punct()
{
        cout<<"\napel destructor ptr. x="<<x<<", y="<<y<<", adresa de memorie:"<<this<<endl;
}
int Punct::getx()
{       return x;       }
int Punct::gety()
{       return y;       }
void Punct::move(int dx, int dy)
{       x+=dx;
        y+=dy;
}
void Punct::afisare()
{       cout<<"\nCoordonatele Pozitiei sunt:";
```

```
        cout<<"\tx="<< x <<"\ty="<< y << endl;
}

void main()
{
 Punct p1;      // pentru p1 se apeleaza constructorul implicit
 Punct p2(3,7);         // pentru p2 se apeleaza constructorul cu parametri
 Punct * pp;
 cout<<"\nPozitia p1:";
 p1.afisare();                  // p1 a fost initializat cu (0,0)
 cout<<"\nPozitia p2:";
 p2.afisare();                  // p2 a fost initializat cu (3,7)
 p1.move(5,10);                 // modificare p1
 p2=p1;                         // p2 copiaza membru cu membru valorile din p1
 cout<<"\nPozitia p1:";
 p1.afisare();
 cout<<"\nPozitia p2:";
 p2.afisare();
 Punct p3(p1);                  // p3 se creeaza ca o copie a lui p1
 cout<<"\nPozitia p3, copie a lui p1:";
 p3.afisare();
 pp=&p1;                        // Punct p1 va putea fi referita prin variabila pointer pp
 cout<<"\nPozitia referita prin pointerul pp este:";
 pp->afisare();
 pp=new Punct;                  //alocare dinamica de memorie-se apeleaza constructorul implicit
 cout<<"\nPozitia referita prin pointerul pp este:";
 pp->afisare();
 delete pp;
 getch();
}
```

Pentru vizualizarea apelului funcţiei constructor s-a inclus afişarea unui mesaj. Clasa nu are nevoie de destructor, el însă a fost definit formal pentru a permite vizualizarea momentului în care se elimină obiectele de tip Punct.

Utilizarea claselor care au definite funcţii constructor şi destructor garantează că obiectele create, indiferent că sunt statice, automatice sau dinamice, sunt aduse într-o stare iniţială adecvată utilizării, cu evitarea utilizării unor valori reziduale, iar la eliminarea lor se efectuează toate operaţiile prealabile necesare.

13.2.4.Manevrarea dinamică a obiectelor

În mod similar utilizării funcţiilor C malloc() şi respectiv free(), se pot crea şi distruge obiecte dinamice utilizând operatorii **new** şi **delete**.

Folosirea operatorului **new** în cazul tipurilor clasă, în afară de alocarea de memorie, are ca efect şi apelul unui constructor, ceea ce nu se întâmplă în cazul funcţiei malloc(). Operatorul **delete** este analog funcţiei free(), dar, în plus faţă de aceasta, în momentul eliminării obiectului din memorie va apela funcţia destructor.

```
 Punct *pp;                     // declararea unui pointer la tipul Punct
 pp=new Punct;                  // alocare dinamica de memorie - se apeleaza  constructorul implicit
```

```
cout <<"\nPozitia referita prin Puncterul pp este:";
pp->afisare();
delete pp;                          // eliminarea obiectului alocat la adresa pp;
                                    // se apelează în mod implicit destructorul
```

13.2.5.Tablouri de obiecte

Tablourile pot avea elemente de orice tip, inclusiv de tip clasă.

La crearea unui tablou cu elemente de tip clasă, se va apela constructorul clasei tip element pentru fiecare element în parte. În cazul tablourilor nu există posibilitatea specificării valorilor corespunzătoare parametrilor, deci, pentru tipul clasă corespunzător elementelor tabloului, este obligatoriu să existe declarat constructor implicit sau constructor cu toţi parametrii impliciţi. Pentru fiecare element de tablou de tip clasă se apelează constructorul, iar la încetarea domeniului de existenţă a tabloului, pentru fiecare element de tablou se apelează destructorul clasei.

Tablourile pot fi create prin alocare statică de memorie:

```
cout<<"\nSe aloca static un tablou cu 5 elemente de tip Punct\n";
Punct tab[5];                       // declararea unui tablou cu elemente de tip Punct; se
                                    // apelează de 5 ori constructorul implicit
cout<<"\nElementele tabloului au valorile:\n";
for( i=0; i<5; i++)                 // se apelează functia de afisare pentru fiecare element al
    tab[i].afisare();               // tabloului
```

sau prin alocare dinamică de memorie:

```
cout<<"\nSe aloca dinamic un tablou cu 6 elemente de tip Punct\n";
 pp=new Punct[6];                   // se aloca memorie pentru un tablou cu 6 elemente de tip
                                    // Punct; se apelează de 6 ori constructorul implicit

 cout<<"\nElementele tabloului au valorile:\n";
 for( i=0; i<6; i++)                // se apelează functia de afisare pentru fiecare element al
      pp[i].afisare();              // tabloului
```

În cazul alocării statice a tabloului, el va fi eliminat din memorie în momentul în care se încheie domeniul de existenţă al său, de exemplu la încheierea funcţiei în care a fost declarat. În cazul alocării dinamice, eliberarea memoriei se face prin folosirea operatorului delete cu sintaxa:

```
delete [] pp;                       // se eliberează memoria alocată tabloului si se apelează
                                    // destructorul pentru fiecare element al tabloului în parte
```

13.2.6.Transferul obiectelor ca parametri sau rezultat

În expresii, operanzii pot fi obiecte ale claselor definite, la fel rezultatul returnat de către acestea. De asemenea, parametrii funcţiilor şi rezultatul returnat de către acestea pot fi obiecte ale unor clase.

În procesul evaluării expresiilor, ca şi la preluarea valorilor pentru parametri şi returnarea

rezultatelor, apar situaţii de memorare temporară de valori, creându-se obiecte temporare. Obiectele temporare au durata de viaţă limitată la durata de execuţie a blocului de instrucţiuni căruia le aparţin, dar, spre deosebire de variabilele automatice, ele au existenţă ascunsă utilizatorului.

Un caz particular îl constituie parametrii transferaţi prin valoare. Ei pot fi asimilaţi cu variabilele automatice, dar spre deosebire de acestea timpul lor de viaţă este mai lung decât timpul de execuţie al domeniului lor. Aceste obiectele temporare sunt create înainte de începerea execuţiei funcţiei şi sunt eliminate după încheierea execuţiei acesteia.

La transferul prin valoare a unui parametru sau al unui rezultat se creează obiecte temporare prin apelul constructorului de copiere al clasei respective, sau a constructorului de copiere generat de compilator, în lipsa definirii acestuia.

Pentru obiecte de dimensiuni mari, se recomandă transferul parametrilor şi rezultatului prin referinţă deoarece, la transfer doar sunt redenumite obiecte deja existente, deci nu se mai creează obiecte temporare pentru parametri, astfel încât se reduce încărcarea stivei şi de asemenea creşte viteza de execuţie. În această situaţie însă trebuie avut în vedere faptul că se pot produce modificări asupra unor obiecte externe funcţiei. Uneori chiar se urmăreşte acest lucru, alteori însă nu. Dacă se doreşte protecţia acestor obiecte, se poate folosi cuvântul cheie **const** care va împiedica orice modificare a unui obiect.

Se pot defini funcţii care au parametrii de tip Punct sau care returnează obiecte de acest tip, transferul putând fi făcut în orice modalitate, prin valoare, prin adresă, sau prin referinţă.

Spre exemplu, se poate defini o funcţie care face transfer prin valoare a unei date de tip Punct:

```
void functia_1(Punct p)
{       p.afisare();
        p.move(1,2);
        p.afisare();
}
```

Parametrul p se creează prin transfer de valoare, adică copiind parametrul efectiv folosit la apel, ca urmare se va apela constructorul de copiere. Modificarea obiectului p nu afectează parametrul efectiv folosit.

În exemplul următor se face transfer prin referinţă.

```
void functia_2(Punct&p)
{       p.afisare();
        p.move(1,2);
        p.afisare();
}
```

Variabila p este aliasul unei variabile exernă funcţiei, astfel că modificarea se va produce asupra acesteia. Dacă se doreşte protejarea ei, se poate defini funcţia astfel:

```
void functia_2(const Punct&p)
{       p.afisare();
        p.move(1,2);            // Eroare, p nu poate fi modificat
        p.afisare();
}
```

Funcţia cu prototipul:

```
Punct functia_3();
```

returnează prin valoare un obiect de tip Punct. La returnare, se generează un obiect de tip Punct prin copierea unei variabile din domeniul funcţiei. Se va folosi constructorul de copiere.

Exemplu

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

Exemplul 13.2. prezinta o aplicaţie care include aspectele referitoare la crearea şi manevrarea obiectelor de tip clasă discutate în acest laborator.
Observatii: Funcţiile constructor şi destructor conţin mesaje prin care, la execuţie, se poate observa momentul în care se apelează aceste funcţii. Prin afişarea adreselor obiectelor pentru care au fost apelate funcţiile (prin pointerul this), se poate observa că, exceptând alocarea dinamică de memorie, ordinea de creare a obiectelor este inversă eliminării lor.

```
*****************************************************************************/
#include <iostream.h>
#include <conio.h>

class Punct     {
                  private:
                        int x, y;
                  public:
                        Punct();                    // constructor implicit
                        Punct(int, int);            // constructor cu parametri
                        Punct(Punct&);              // constructor de copiere
                        ~Punct();                   // destructor
                        int getx();
                        int gety();
                        void move(int,int);
                        void afisare();
                 };
void functia_1(Punct);              // Transfer prin valoare
void functia_2(Punct&);             // Transfer prin referinta
Punct functia_3();                  // Transfer prin valoare
Punct& functia_4(Punct&);           // Transfer prin referinta

void main()
{ Punct p1;                         // pentru p1 se apeleaza constructorul implicit
  Punct p2(3,7);                    // pentru p2 se apeleaza constructorul cu parametri
  Punct * pp;
  int i;

  cout<<"\nPozitia p1:";
  p1.afisare();                     // p1 a fost initializat cu (0,0)
  cout<<"\nPozitia p2:";
  p2.afisare();                     // p2 a fost initializat cu (3,7)
  p1.move(5,10);                // modificare p1
  p2=p1;                        // p2 copiaza membru cu membru valorile din p1
```

```
cout<<"\nPozitia p1:";
p1.afisare();
cout<<"\nPozitia p2:";
p2.afisare();
Punct p3(p1);                   // p3 se creeaza ca o copie a lui p1
cout<<"\nPozitia p3, copie a lui p1:";
p3.afisare();
pp=&p1;                         // Punct p1 va putea fi referita prin variabila Puncter pp
cout<<"\nPozitia referita prin Puncterul pp este:";
pp->afisare();
pp=new Punct;                   //alocare dinamica de memorie-se apeleaza constructorul implicit
cout<<"\nPozitia referita prin Puncterul pp este:";
pp->afisare();
delete pp;
getch();
cout<<"\nSe aloca static un tablou cu 5 elemente de tip Punct\n";
Punct tab[5];                   // declararea unui tablou cu elemente de tip Punct-se apeleaza
                                // constructorul implicit
cout<<"\nElementele tabloului au valorile:\n";
for( i=0; i<5; i++)
        tab[i].afisare();
for( i=0; i<5 ; i++)
        tab[i].move(i,i);
cout<<"\nElementele tabloului au valorile:\n";
for( i=0; i<6; i++)
        tab[i].afisare();
getch();
cout<<"\nSe aloca dinamic un tablou cu 6 elemente de tip Punct\n";
pp=new Punct[6];                //alocare dinamica de memorie-se apeleaza constructorul implicit
cout<<"\nElementele tabloului au valorile:\n";
for( i=0; i<6; i++)
        pp[i].afisare();
for( i=0; i<6 ; i++)
        pp[i].move(i,i);
cout<<"\nElementele tabloului au valorile:\n";
for( i=0; i<6; i++)
        pp[i].afisare();
getch();
cout<<"\nSe dealoca tabloul cu elemente de tip Punct\n";
delete [] pp;                   // se apeleaza destructorul pentru fiecare element al tabloului
getch();
cout<<"\np1:";
p1.afisare();
cout<<"\n------- Apel functia 1 -------";
functia_1(p1);
cout<<"\n------- Final functia 1 -------";
cout<<"\np1:";
p1.afisare();
getch();
cout<<"\n------- Apel functia 2 -------";
functia_2(p1);
cout<<"\n------- Final functia 2 -------";
cout<<"\np1:";
p1.afisare();
```

```
 getch();
 cout<<"\n------- Apel functia 3 -------";
 p1=functia_3();
 cout<<"\n------- Final functia 3 -------";
 cout<<"\np1:";
 p1.afisare();
 getch();
 cout<<"\n------- Apel functia 4 -------";
 p1=functia_4(p1);
 cout<<"\n------- Final functia 4 -------";
 getch();
}
Punct::Punct()
{       x=0;
        y=0;
        cout<<"\napel constructor implicit: x="<<x<<", y="<<y;
        cout<< ", adresa de memorie:" << this << endl;
        getch();
}
Punct::Punct(int abs, int ord)
{       x=abs;
        y=ord;
        cout<<"\napel constructor cu parametri x="<<x<<", y="<<y;
        cout<<", adresa de memorie:"<<this<<endl;
        getch();
}
Punct::Punct(Punct& p)
{       x=p.x;
        y=p.y;
        cout<<"\napel constructor de copiere x="<<x<<", y="<<y;
        cout<<", adresa de memorie:"<<this<<endl;
        getch();
}
Punct::~Punct()
{
        cout<<"\napel destructor ptr. x="<<x<<", y="<<y;
        cout<<", adresa de memorie:"<<this<<endl;
        getch();
}
int Punct::getx()
{       return x;       }

int Punct::gety()
{       return y;       }

void Punct::move(int dx, int dy)
{       x+=dx;
        y+=dy;
}
void Punct::afisare()
{       cout<<"\nCoordonatele pozitiei sunt:";
        cout<<"\tx="<< x <<"\ty="<< y << endl;
}
void functia_1(Punct p)
```

```
{       p.afisare();
        p.move(1,2);
        p.afisare();
}
void functia_2(Punct&p)
{       p.afisare();
        p.move(1,2);
        p.afisare();
}
Punct functia_3()
{       Punct p(11, 22);
        p.afisare();
        p.move(1,2);
        p.afisare();
        return p;
}
Punct& functia_4(Punct&p)
{       p.afisare();
        p.move(1,2);
        p.afisare();
        return p;
}
```

13.2.7. Funcţii şi clase prietene unei clase

Accesul la membrii private ai unei clase se poate acorda, în afara funcţiilor membre şi unor funcţii nemembre, dacă sunt declarate cu specificatorul friend.

Funcţiile declarate prietene unei clase pot fi funcţii independente sau funcţii membre ale altor clase. Funcţiile prietene sunt externe clasei, deci apelul lor nu se face asociat unui obiect al clasei.

Funcţiile prietene sun funcţii ordinare, care se declară şi se definesc folosindu-se sintaxa obişnuită. Relaţia de prietenie cu o clasă este declarată în interiorul clasei căreia îi este prietenă acea funcţie, ataşând cuvântul cheie friend la prototipul funcţiei:

friend tip_functie nume_functie(lista_parametri);

Funcţiile prietene unei clase au acces direct la membrii privaţi ai clasei, deci se încalcă principiul încapsulării, dar în anumite situaţii sunt utile.

In continuare este exemplificat modul de utilizare a unei funcţii prietene unei clase.

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

class pozitie
{  int x, y;
public:
     pozitie(int abs, int ord)
       {  x=abs;  y=ord;  }
     void deplasare(int dx, int dy)
       {  x+=dx;  y+=dy;  }
     friend void compar(pozitie &, pozitie &);          // funcţia compar() se declară prietenă
                                                        // clasei pozitie
```

```
    friend double distanta(pozitie &, pozitie &);      // funcţia distanta() se declară prietenă
                                                       // clasei pozitie
};
void compar(pozitie &p1, pozitie &p2)        // funcţia compar() este externă clasei poziţie, deci nu
  {                                          // poate fi definită decât în afara clasei pozitie
    if ((p1.x= =p2.x)&&(p1.y= =p2.y))        // funcţia prietenă compar() are acces la
                                             // membrii private ai clasei pozitie
        cout<<"\n Pozitii identice";
    else
        cout<<"\n Pozitii diferite";
}
double distanta(pozitie &p1, pozitie &p2)
{     return sqrt((p1.x-p2.x)* (p1.x-p2.x)+ (p1.y-p2.y)* (p1.y-p2.y));
}

void main()
{   pozitie p1(1, 1), p2(3, 3);
    compar(p1, p2);                                          // apel al funcţiei compar()
    p1.deplasare(2, 2);
    compar(p1, p2);                                          // apel al funcţiei compar()
    cout<<"\nDistanta dintre p1si p2 este:" << distanta(p1, p2);      // apel al funcţiei distanta()
}
```

13.2.8. Supradefinirea operatorilor

Limbajul C++ permite programatorului definirea diverselor operaţii cu obiecte ale claselor, folosind simbolurile operatorilor standard.

Operatorii standard sunt deja supradefiniţi, ei putând intra în expresii ai căror operanzi sunt de diferite tipuri fundamentale, operaţia adecvată fiind selectată în mod similar oricăror funcţii supradefinite.

Un tip clasă poate conţine definirea unui set de operatori specifici asociaţi, prin supradefinirea operatorilor existenţi, utilizând funcţii cu numele:

operator simbol_operator

unde:

- *operator* este cuvânt cheie dedicat supradefinirii operatorilor;
- s*imbol_operator* poate fi simbolul oricărui operator, mai puţin operatorii: (.), (.*), (::) şi (? :).

Pentru definirea funcţiilor operator se pot folosi două variante de realizare:

- definirea funcţiilor operator cu funcţii membre clasei ;
- definirea funcţiilor operator cu funcţii prietene clasei.

Prin supradefinirea operatorilor **nu se pot modifica**:

- **pluralitatea operatorilor** (operatorii unari nu pot fi supradefiniţi ca operatori binari sau invers);
- **precedenţa şi asociativitatea operatorilor**.

Funcţia operator trebuie să aibă cel puţin un parametru, implicit sau explicit de tipul clasă căruia îi este asociat operatorul. Pentru tipurile standard operatorii îşi păstrează definirea.

Operatorii =, [], (), -> pot fi supradefiniţi doar cu funcţii membre nestatice ale clasei.

Programatorul are deplină libertate în modul în care defineşte noua operaţie, dar în general pentru a da o bună lizibilitate programului, se recomandă ca noua operaţie să fie asemănătoare

semnificaţiei originale, dacă ea există pentru respectivul operator asociat cu clasa definită.

De exemplu, dacă se reprezintă numere complexe printr-o clasă, se poate defini operaţia de adunare, folosind operatorul +, cu semnificaţia matematică a operaţiei. Pentru operatori, cum ar fi &, care în definiţia originară reprezintă operaţia "şi la nivel de bit", se pot face asocieri cu alte operaţii efectuate cu numere complexe.

Funcţiile operator trebuie să aibă cel puţin un parametru de tip clasă, acesta reprezentând unul dintre operanzi.

Prin supradefinirea operatorilor, scrierea programelor devine mai uşoară şi se obţine claritate în transpunerea algoritmilor.

- **Supradefinirea operatorilor folosind funcţii prietene clasei**

Folosind funcţii prietene unei clase, se pot defini operatori unari sau binari.

Funcţia prietenă operator care supradefineşte un operator binar, va avea doi parametri care reprezintă de fapt operanzii. Cel puţin un operand trebuie să fie de tip clasă, deci cel puţin un parametru trebuie să fie de tip clasă.

Spre exemplu, se consideră clasa complex care reprezintă numere complexe prin componentele lor, parte reală, respectiv parte imaginară (re + i * im), .

În cazul definirii operaţiei de adunare a două numere complexe, funcţia operator + definită ca funcţie prietenă clasei, este necesar să preia doi parametri de tip complex, prototipul funcţiei fiind:

complex operator+(complex, complex);

sau

complex operator+(complex&, complex&);

Ca urmare a definirii funcţiei operator+, expresia x+y , unde x şi y sunt două obiecte de tip complex, este echivalentă cu un apel al funcţiei operator+ de forma:

operator+(x, y);

Operatorii unari pot fi supradefiniţi cu funcţii membre clasei, având un parametru de tipul clasei, cu sintaxa:

operator op(x) // x e de tip clasă

- **Supradefinirea operatorilor folosind funcţii membre ale clasei**

Folosind funcţii membre ale unei clase, se pot defini aproape toţi operatorii. Operatorii =, [], (), -> pot fi supradefiniţi doar prin funcţii membre ale claselor.

O funcţie membră a unei clase primeşte ca parametru implicit adresa obiectului pentru care este apelată (referită prin cuvântul cheie this), acesta constituind primul operand, deci funcţia va trebui să aibă un singur parametru explicit, care să precizeze cel de al doilea operand. Prototipul funcţiei va fi:

complex operator+(complex);

Expresia x+y , unde x şi y sunt obiecte de tip complex, este echivalentă cu un apel de forma:

x.operator+(y)

Se observă că, în cazul definirii funcţiei operator ca funcţie membră a clasei, primul operand

este obligatoriu de tipul clasei respective.

Operatorii unari pot fi supradefiniţi atât cu funcţii membre clasei, cât şi cu funcţii prietene clasei. Expresia “op x”, unde op este simbolul operatorului supradefinit, este echivalentă cu:

x.operator op(); // pentru definire cu funcţie membră

sau

operator op(x) // pentru definire cu funcţie prietenă

Exemplu

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

Exemplul 13.3. Se defineste clasa complex pentru reprezentarea numerelor complexe. Se definesc functii care opereaza cu numere complexe (suma, diferenta, inmultire, ...). Sunt exemplificate definitii de operatii unare si binare, definite prin functii membre ale clasei si functii prietene.

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

class complex
{       float re, im;     //re=partea reala,
                          //im=partea imaginara
public:
        complex(float r=0, float i=0)    //constructor
         {
                re=r; im=i;
         }

        void afisare()
         {  cout<<"\nre= "<<re;
            cout<<", im= "<<im;
         }
        void citire();
        complex operator-();
        complex operator+(complex&);
        friend int operator==(complex&,complex&);
        friend complex operator~(complex&);                    // returneaza conjugatul
        friend complex operator*(complex&, complex&);
};
void complex::citire()
{       cout<<"\nre=";
        cin>>re;
        cout<<"im=";
        cin>>im;
}
complex complex::operator-()
{       complex c;
        c.re=-re;
        c.im=-im;
        return c;
}
```

```
complex complex::operator+(complex&c)
{       complex s;
        s.re=re+c.re;
        s.im=im+c.im;
        return s;
}
complex operator~(complex&c)
{       complex cc;
        cc.re=c.re;
        cc.im=-c.im;
        return cc;
}
complex operator*(complex&c1, complex&c2)
{       complex c;
        c.re = c1.re*c2.re - c1.im*c2.im;
        c.im = c1.re*c2.im + c1.im*c2.re;
        return c;
}
int operator==(complex&c1, complex&c2)
{       if(c1.re==c2.re && c1.im==c2.im)
                return 1;
        else
                return 0;
}
void main(void)
{       complex c1(1,1), c2(2,2), c3;
        cout<<"c1: ";
        c1.afisare();
        cout<<"\nc2: ";
        c2.afisare();
        cout<<"\nc3: ";
        c3.afisare();
        c3 = c1 + c2;
        cout<<"\nc3 = c1 + c2 : ";
        c3.afisare();
        c3 = c1 * c2;
        cout<<"\nc3 = c1 * c2 :";
        c3.afisare();
        c3 = -c1;
        cout<<"\nc3 = -c1 :";
        c3.afisare();
        c3 = ~c1;
        cout<<"\nc3 = ~c1 : ";
        c3.afisare();
        cout<<"\nc1=";
        c1.citire();
        cout<<"\nc2=";
        c2.citire();
        if (c1 == c2)
                cout<<"\n c1, c2 sunt identice";
        else
                cout<<"\n c1, c2 sunt diferite\n";
        cin.get();
}
```