

Adrian DEACONU

Programarea orientată pe obiecte I

Limbajul C++

Cuvânt înainte

Cartea de față se dorește a fi, în principal, un ghid pentru studenții din domeniul Informatică, dar, evident, ea poate fi utilă tuturor celor care vor să învețe să programeze orientat pe obiecte, în general și în C++, în particular. Este bine ca cel care citește această lucrare să nu fie începător în programare și, mai mult, este nevoie să aibă cunoștințe avansate despre limbajul C. Anumite concepte generale cum ar fi constante, variabile, funcții, tipuri numerice, caractere, string-uri, pointeri, tablouri etc. se consideră cunoscute.

Lucrarea este structurată pe două părți.

În prima parte se prezintă elementele introduse odată cu apariția limbajului C++, care nu existau în C și care nu au neapărat legătură cu programarea orientată pe obiecte.

În partea a doua este făcută o prezentare teoretică a programării orientate pe obiecte, introducând și conceptele POO. După această scurtă prezentare pur teoretică se prezintă programarea orientată pe obiecte din C++. Tot aici sunt prezentate și o parte din clasele care se instalează odată cu mediul de programare (clase pentru lucrul cu fluxuri, clasa complex etc.).

La fiecare capitol, sunt date exemple sugestive, care ilustrează din punct de vedere practic elemente de noutate. Este bine ca aceste exemple să fie înțelese și, acolo unde este nevoie, să fie scrise și rulate de către cititor. Programele din această carte nu conțin erori, deoarece ele au fost întâi testate și abia apoi introduse în lucrare.

În general, tot ceea ce este prezentat în această carte (teorie și aplicații) este recunoscut atât de compilatorul C++ al firmei Borland, cât și de compilatorul Visual C++ al companiei Microsoft.

Autorul.

CUPRINS

PARTEA ÎNTÂI

1.1. Ce este limbajul C++ ?	5
1.2. Elemente introduse de C++	5
1.3. Declarația variabilelor în C++	6
1.4. Fluxuri standard în C++	7
1.5. Manipulatori	7
1.6. Indicatori de formatare	8
1.7. Alocarea dinamică a memoriei în C++	10
1.8. Funcții în C++	15
1.8.1. Funcții cu același nume și cu parametrii diferiți	15
1.8.2. Transmiterea parametrilor prin referință	16
1.8.3. Valori implicite pentru parametrii funcțiilor	18
1.8.4. Funcții <i>inline</i>	19
1.8.5. Funcții șablon (template)	20
1.9. Supraîncărcarea operatorilor	22
1.10. Tratarea excepțiilor	28

PARTEA A DOUA

2.1. Prezentare teoretică a P.O.O.	33
2.2. Programarea orientată pe obiecte în C++	35
2.2.1. Neajunsuri ale POO în C	35
2.2.2. Declarația unei clase în C++	36
2.2.3. Declarația și descrierea funcțiilor membre	36
2.2.4. Constructori	38
2.2.5. Destructori	38
2.2.6. Funcții prietene (<i>friend</i>) unei clase	40
2.2.7. Declarația, descrierea operatorilor pt. o clasă.....	42
2.2.8. Membri statici	44
2.2.9. Pointerul <i>this</i>	45
2.2.10. Constructorul de copiere	49
2.2.11. Moștenirea în C++	51
2.2.12. Funcții virtuale	53
2.2.13. Destructori virtuali	55
2.2.14. Funcții pur virtuale	56
2.2.15. Moștenire multiplă	60

2.2.16. Clase virtuale	60
2.2.17. Constructori pentru clase virtuale	62
2.2.18. Clase imbricate	66
2.2.19. Clase șablon (<i>template</i>)	67
2.3. Fluxuri în C++	71
2.3.1. Ierarhia <i>streambuf</i>	71
2.3.2. Ierarhia <i>ios</i>	72
2.4. Fișiere în C++	76
2.5. Prelucrarea string-urilor în C++	91
2.6. Clasa <i>complex</i> din C++	94
 Indicații și răspunsuri	 96
 ANEXE	 98
 BIBLIOGRAFIE	 104

PARTEA ÎNTÂI

Obiective

În aceasta primă parte ne propunem să vedem ce este limbajul C++ și să studiem elementele introduse de C++, care nu au neapărat legătură cu programarea orientată pe obiecte.

1.1. Ce este limbajul C++ ?

Limbajul C++ este o extensie a limbajului C. Aproape tot ce ține de limbajul C este recunoscut și de către compilatorul C++. Limbajul C++ a apărut ca o necesitate, în sensul că el a adus completări limbajului C care elimină câteva neajunsuri mari ale acestuia. Cel mai important neajuns al limbajului C este lipsa posibilității de a scrie cod orientat pe obiecte în adevăratul sens al cuvântului. În C se poate scrie într-o manieră rudimentară cod orientat pe obiecte folosind tipul **struct**, în interiorul căruia putem avea atât câmpuri, cât și metode. Orientarea pe obiecte cu tipul struct are câteva mari lipsuri: membrii săi se comportă toți ca niște membri publici (accesul la ei nu poate fi restrictionat), nu avem constructori, destructori, moștenire etc.

Limbajul C a fost lansat în anul 1978 și s-a bucurat încă de la început de un real succes. Acest lucru s-a datorat ușurinței cu care un programator avansat putea scrie programe în comparație cu restul limbajelor ce existau atunci pe piață, datorită în special modului abstract și laconic în care se scrie cod. De asemenea, modul de lucru cu memoria, cu fișiere este mult mai transparent. Acest lucru are ca mare avantaj viteza crescută de execuție a aplicațiilor, dar poate foarte ușor conduce (mai ales pentru începatori) la erori greu de detectat, datorate “călcării” în afara zonei de memorie alocate.

La sfârșitul anilor '80 a apărut limbajul C++ ca o extensie a limbajului C. C++ preia facilitățile oferite de limbajul C și aduce elemente noi, dintre care cel mai important este noțiunea de clasă, cu ajutorul căreia se poate scrie cod orientat pe obiecte în toată puterea cuvântului. Limbajul C++ oferă posibilitatea scrierii de funcții și clase șablon, permite redefinirea (supraîncărcarea) operatorilor și pentru alte tipuri de date decât pentru cele care există deja definiți, ceea ce oferă programatorului posibilitatea scrierii codului într-o manieră mult mai elegantă, mai rapidă și mai eficientă.

În anul 1990 este finalizat standardul ANSI-C, care a constituit baza elaborării de către firma Borland a diferitelor versiuni de medii de programare.

În prezent sunt utilizate într-o mare măsură limbajele Java (al cărui compilator este realizat firma Sun) și Visual C++ (care face parte din pachetul Visual Studio al firmei Microsoft), care au la baza tot standardul ANSI-C. Există însă și competitori pe măsură. Este vorba în prezent în special de limbajele ce au la bază platforma .NET - alternativa Microsoft pentru mașina virtuală Java. Poate cel mai puternic limbaj de programare din prezent este C# (creat special pentru platforma .NET). Limbajul C# seamănă cu C/C++, dar totuși el nu este considerat ca făcând parte din standardul ANSI-C.

În C++ se poate programa orientat pe obiecte ținându-se cont de toate conceptele: *abstractizare, moștenire, polimorfism* etc.

Odată cu mediul de programare al limbajului C++ (fie el produs de firma Borland sau de firma Microsoft) se instalează și puternice ierarhii de clase, pe care programatorul le poate folosi, particulariza sau îmbogăți.

1.2. Elemente introduse de C++

În acest capitol vom face o scurtă enumerare a elementelor introduse de C++, care nu se găseau în limbajul C.

Lista celor mai importante noutăți aduse de limbajul C++ este:

- tipul **class** cu ajutorul căruia se poate scrie cod adevărat orientat pe obiecte
 - posibilitatea declarării variabilelor aproape oriunde în program
 - o puternică ierarhie de clase pentru fluxuri
 - alocarea și eliberarea dinamică a memoriei cu ajutorul operatorilor **new** și **delete**
 - posibilitatea de a scrie mai multe funcții cu același nume dar cu parametrii diferiți
 - valori implicite pentru parametrii funcțiilor
 - funcțiile *inline*
 - funcțiile și clasele șablon (suport adevărat pentru programare generică)
 - tratarea excepțiilor stil modern (folosind instrucțiunea **try ... catch**)
 - supraîncărcarea operatorilor
 - clasa *complex*
- etc.

Aceste elemente introduse de C++ vor fi prezentate pe larg în cele ce urmează.

1.3. Declarația variabilelor în C++

În C variabilele locale trebuie să fie declarate pe primele linii ale corpului funcției (inclusiv în funcția principală). În C++ declarațiile variabilelor pot fi făcute aproape oriunde în program. Ele vor fi cunoscute în corpul funcției din locul în care au fost declarate în jos. Declarațiile de variabile pot apărea chiar și în interiorul instrucțiunii *for*. Iată un exemplu în acest sens:

```
int n=10,a[10];
for (int s=0,i=0;i<n;i++)
    s+=a[i];
float ma=s/n;
```

Ce se întâmplă însă dacă declarăm o variabilă în corpul unei instrucțiuni de decizie, compusă sau repetitivă? Este posibil ca grupul de instrucțiuni ce alcătuiește corpul să nu se execute niciodată. Pentru a se rezolva această problemă orice declarație de variabilă în interiorul unui grup de instrucțiuni delimitat de acolade sau din corpul unei instrucțiuni este cunoscută numai în interiorul grupului, respectiv corpului. Variabilele declarate în interiorul blocurilor intră în stiva de execuție a programului de unde ies la terminarea execuției blocului.

Dăm un exemplu:

```
for (int i=0;i<n;i++)
    for (int j=0;j<n;j++)
    {
        int k=0;
        ...
    }
cout<<i; // corect, variabila i exista
cout<<j; // incorect, variabila j nu mai exista
cout<<k; // incorect, variabila k nu mai exista
```

Spre deosebire de C++, în Java nici variabila *i* din exemplul de mai sus nu ar fi fost cunoscută după ce se iese din instrucțiunea *for*.

1.4. Fluxuri standard în C++

Pentru a lucra cu fluxuri în C++ există o puternică ierarhie de clase. Pentru a folosi facilitățile C++ de lucru cu fluxuri este necesar și în general suficient să se includă fișierul antet “*iostream.h*”, care reprezintă o alternativă la ceea ce oferea fișierul antet “*stdio.h*” în C.

Pentru extragerea de date dintr-un flux în C++ în modul text se folosește în general operatorul >>, iar pentru introducerea de date într-un flux în modul text folosim operatorul <<. Așadar, celor doi operatori care erau folosiți în C numai pentru shift-area biților unei valori întregi, în C++ li s-a dat o nouă semnificație.

În C++ avem obiectul **cin** care corespunde în C fluxului standard de intrare *stdin* (tastatura). De exemplu, pentru citirea de la tastatură a două variabile procedăm astfel:

```
int n;  
float x;  
cin>>n>>x;
```

Instrucțiunea de mai sus are următoarea semnificație: din fluxul standard de intrare se extrag două valori (una întreagă și apoi una reală). Cele două valori se depun în variabilele *n* și respectiv *x*.

Pentru afișarea pe ecranul monitorului se folosește obiectul **cout** care corespunde fluxului standard de ieșire **stdout**. Iată un exemplu.

```
cout<<"Am citit: "<<n<<" si "<<x<<endl;
```

În fluxul standard de ieșire se trimit: o constantă de tip *string*, o valoare întreagă (cea reținută în variabila *n*), un alt *string* constant și o valoare reală (reținută în variabila *y*). După afișare, manipulatorul *endl* introdus de asemenea în flux face salt la începutul următoarei linii de pe ecran.

În C++ avem două obiecte pentru fluxuri de erori. Este vorba de **cerr** și **clog**. Primul obiect corespunde fluxului standard de erori **stderr** din C. Introducerea de date în fluxul **cerr** are ca efect afișarea lor imediată pe ecranul monitorului, dar pe altă cale decât cea a fluxului **cout**. Al doilea flux de erori (**clog**) nu are corespondent în C. Acest flux este unul buffer-izat, în sensul că mesajele de erori se colectează într-o zonă de memorie RAM, de unde pot ajung pe ecran numai când se dorește acest lucru, sau la terminarea execuției programului. Golirea buffer-ului pe ecran se poate face apelând metoda *flush()*, sub forma:

```
clog.flush();
```

1.5. Manipulatori

Manipulatorii pot fi considerați niște funcții speciale care se introduc în lanțurile de operatori << sau >> în general pentru formatare. În exemplul din capitolul anterior am folosit manipulatorul *endl*, care face salt la linie nouă.

Manipulatorii fără parametri sunt descriși în fișierul antet “*iostream.h*”, iar cei cu parametri apar în fișierul antet “*iomanip.h*”. Dăm în continuare lista manipulatorilor:

Manipulator	Descriere
dec	Pregătește citirea/scrierea întregilor în baza 10
hex	Pregătește citirea/scrierea întregilor în baza 16

oct	Pregătește citirea/scrierea întregilor în baza 8
ws	Scoate toate spațiile libere din fluxul de intrare
endl	Trimite caracterul pentru linie nouă în fluxul de ieșire
ends	Inserează un caracter NULL în flux
flush	Golește fluxul de ieșire
resetiosflags(long)	Inițializează biții de formatare la valorile date de argumentul <i>long</i>
setiosflags(long)	Modifică numai biții de pe pozițiile 1 date de parametrul <i>long</i>
setprecision(int)	Stabilește precizia de conversie pentru numerele în virgulă mobilă (numărul de cifre exacte)
setw(int)	Stabilește lungimea scrierii formatare la numărul specificat de caractere
setbase(int)	Stabilește baza în care se face citirea/scrierea întregilor (0, 8, 10 sau 16), 0 pentru bază implicită
setfill(int)	Stabilește caracterul folosit pentru umplerea spațiilor goale în momentul scrierii pe un anumit format

Biții valorii întregi transmise ca parametru manipulatorilor *setiosflags* și *resetiosflags* indică modul în care se va face extragerea, respectiv introducerea datelor din/în flux. Pentru fiecare dintre acești biți în C++ există definită câte o constantă.

Formatările cu *setiosflags* și *resetiosflags* au efect din momentul în care au fost introduse în flux până sunt modificate de un alt manipulator.

1.6. Indicatori de formatare

După cum le spune și numele, indicatorii de formatare arată modul în care se va face formatarea la scriere, respectiv la citire în/din flux. Indicatorii de formatare sunt constante întregi definite în fișierul antet "*iostream.h*". Fiecare dintre aceste constante reprezintă o putere a lui 2, din cauză că fiecare indicator se referă numai la un bit al valorii întregi în care se memorează formatarea la scriere, respectiv la citire. Indicatorii de formatare se specifică în parametrul manipulatorului *setiosflags* sau *resetiosflags*. Dacă vrem să modificăm simultan mai mulți biți de formatare, atunci vom folosi operatorul `|` (sau pe biți).

Dăm în continuare lista indicatorilor de formatare:

Indicator	Descriere
ios::skipws	Elimină spațiile goale din buffer-ul fluxului de intrare
ios::left	Aliniaza la stânga într-o scriere formatată
ios::right	Aliniaza la dreapta într-o scriere formatată
ios::internal	Formatează după semn (+/-) sau indicatorul bazei de numerație
ios::scientific	Pregătește afișarea exponențială a numerelor reale
ios::fixed	Pregătește afișarea zecimală a numerelor reale (fără exponent)
ios::dec	Pregătește afișarea în baza 10 a numerelor întregi
ios::hex	Pregătește afișarea în baza 16 a numerelor întregi
ios::oct	Pregătește afișarea în baza 8 a numerelor întregi
ios::uppercase	Folosește litere mari la afișarea numerelor (lietra 'e' de la exponent și cifrele în baza 16)
ios::showbase	Indică baza de numerație la afișarea numerelor întregi
ios::showpoint	Include un punct zecimal pentru afișarea numerelor reale
ios::showpos	Afișează semnul + în fața numerelor pozitive

<code>ios::unitbuf</code>	Golește toate buffer-ele fluxurilor
<code>ios::stdio</code>	Golește buffer-ele lui <i>stdout</i> și <i>stderr</i> după inserare

De exemplu, pentru a afișa o valoare reală fără exponent, cu virgulă, aliniat la dreapta, pe 8 caractere și cu două zecimale exacte procedăm astfel:

```
float x=11;
cout<<setiosflags(ios::fixed|ios::showpoint|ios::right);
cout<<setw(8)<<setprecision(2)<<x;
```

Menționăm faptul că în exemplul de mai sus formaterile date de *setw* și de *setprecision* se pierd după afișarea valorii *x*, iar formaterile specificate prin intermediul manipulatorului *setiosflags* rămân valabile.

Dăm în continuare un alt exemplu ilustrativ pentru modul de utilizare al manipulatorilor și al indicatorilor de formatare:

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

void main()
{
    int i=100;
    cout<<setfill('.');

    cout<<setiosflags(ios::left);
    cout<<setw(10)<<"Baza 8";
    cout<<setiosflags(ios::right);
    cout<<setw(10)<<oct<<i<<endl;

    cout<<setiosflags(ios::left);
    cout<<setw(10)<<"Baza 10";
    cout<<setiosflags(ios::dec | ios::right);
    cout<<setw(10)<<i<<endl;

    cout<<setiosflags(ios::left);
    cout<<setw(10)<<"Baza 16";
    cout<<setiosflags(ios::right);
    cout<<setw(10)<<hex<<i<<endl;
}
```

În urma execuției programului de mai sus, pe ecranul monitorului se vor afișa:

```
Baza 8.....144
Baza 10.....100
Baza 16.....64
```

Rezumat

Am făcut cunoștință cu primele elemente introduse de C++ în plus față de limbajul C. Astfel, variabilele pot fi declarate aproape oriunde în program (chiar și în instrucțiunea *for*). De asemenea, am văzut cum se citesc date de la tastatură (folosind obiectul *cin*), cum se afișează (cu

cout), cum se face o formatare la citire și respectiv la afișare (folosind manipulatori și indicatori de formatare). Ce este poate cel mai important este faptul că alternativa C++ de lucru cu fluxuri este obiect orientată.

Teme

1. De la tastatură se citește o matrice de valori reale. Să se afișeze pe ecran matricea, astfel încât fiecare element al matricei să fie scris aliniat la dreapta, pe opt caractere și cu două zecimale exacte. Dăm ca exemplu afișarea unei matrici cu trei linii și două coloane:

```

    11.17    2.00
   -23.05   44.10
12345.78    0.00

```

2. De la tastatură se citesc numele și vârstele unor persoane. Să se afișeze tabelar, sortat după nume, aceste date precum și media vârstelor, conform modelului din exemplul de mai jos:

Nr.	NUMELE SI PRENUMELE	Varsta
crt.		
1	Ion Monica	19
2	Ionescu Adrian Ionel	25
3	Popescu Gigel	17
4	Popescu Maria	28
Media varstelor:		22.25

1.7. Alocarea dinamică a memoriei în C++

Obiective

În acest capitol vom studia modul în care se face alocarea și eliberarea memoriei în C++, într-o manieră elegantă și modernă cu ajutorul noilor operatori introduși în C++: *new* și *delete*.

Limbajul C++ oferă o alternativă la funcțiile C *calloc*, *malloc*, *realloc* și *free* pentru alocarea și eliberarea dinamică a memoriei, folosind operatorii **new** și **delete**.

Schema generală de alocare dinamică a memoriei cu ajutorul operatorului **new** este:

```
pointer_catre_tip = new tip;
```

Eliberarea memoriei alocate dinamic anterior se poate face cu ajutorul operatorului **delete** astfel:

```
delete pointer_catre_tip;
```

Prezentăm în continuare alocarea și eliberarea dinamică a memoriei pentru rădăcina unui arbore binar:

```
struct TArbBin
{
    char info[20];
    struct TArbBin *ls,*ld;
    }*rad;

// ....

if (!(rad=new struct TArbBin))
{
    cerr<<"Eroare! Memorie insuficienta."<<endl;
    exit(1); // parasirea program cu cod de eroare
}

//....

delete rad;
```

Alocarea și eliberarea dinamică a memoriei pentru un vector de numere întregi folosind facilitățile C++ se face astfel:

```
int n,*a;

cout<<"Dati lungimea vectorului: ";
cin>>n;
if(!(a = new int[n]))
{
    cerr<<"Eroare! Memorie insuficienta." <<endl;
    exit(1);
}

//....

delete [] a;
```

Pentru a elibera memoria ocupată de un vector se folosește operatorul *delete* urmat de paranteze pătrate, după cum se poate observa în exemplul de mai sus. Dacă nu punem paranteze pătrate nu este greșit, dar este posibil ca în cazul unor compilatoare C++ să nu se elibereze memoria ocupată de vector corect și mai ales complet.

În C++ alocarea și eliberarea dinamică a memoriei pentru o matrice de numere reale se poate face ceva mai ușor decât în C folosind operatorii *new* și *delete*.

Prezentăm în continuare două posibile metode de alocare dinamică a memorie pentru o matrice de ordin 2 (cu m linii și n coloane).

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

```

int main(void)
{
    int m,n;
    float **a;

    cout<<"Dati dimensiunile matricii: ";
    cin>>m>>n;

    if (!(a = new float*[m]))
    {
        cerr<<"Eroare! Memorie insuf." <<endl;
        return 1; // parasirea program cu eroare
    }
    for (int i=0;i<m;i++)
        if(!(a[i] = new float[n]))
        {
            cerr<<"Eroare! Memorie insuf." <<endl;
            return 1;
        }

    //....

    for (i=0;i<m;i++)
        delete [] a[i];
    delete [] a;
    return 0;
}

```

Pentru a intelege mai bine ideea de mai sus de alocare a memoriei pentru o matrice, să vedem ce se întâmplă în memorie.

Prima data se alocă memorie pentru un vector (cu m elemente) de pointeri către tipul *float* (tipul elementelor matricii). Adresa către această zonă de memorie se reține în pointerul a . După prima alocare urmează m alocări de memorie necesare stocării efective a elementelor matriciei. În vectorul de la adresa a (obținut în urma primei alocări) se rețin adresele către începuturile celor m zone de memorie corespunzătoare liniilor matriciei. Tot acest mecanism de alocare a memoriei este ilustrat în figura 1.

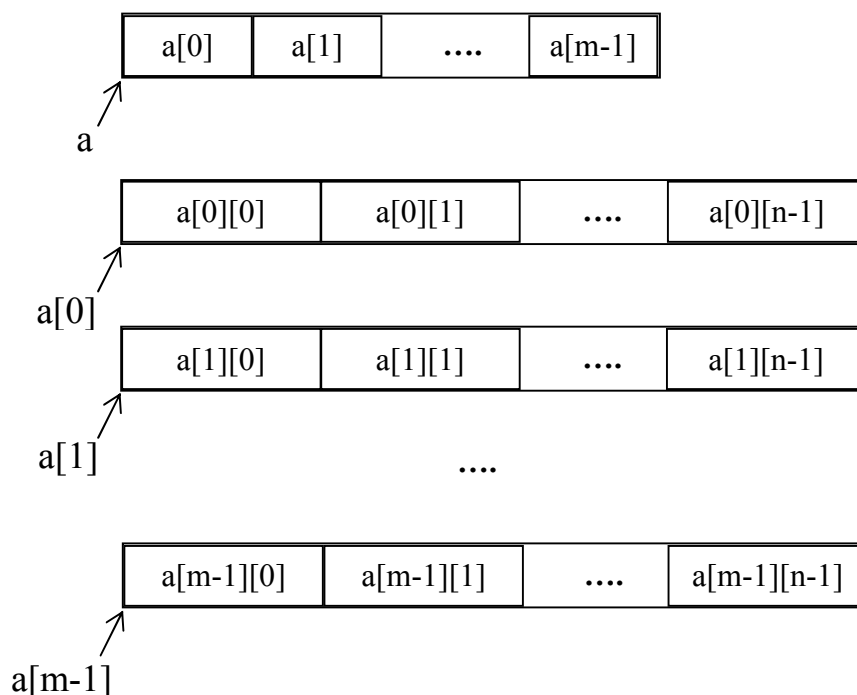


Fig. 1: Prima schemă de alocare a memoriei pentru o matrice

Eliberarea memoriei necesare stocării matricei se face evident tot în $m+1$ pași.

Avantajul alocării dinamice pentru o matrice în stilul de mai sus este dat de faptul că nu este necesară o zonă de memorie continuă pentru memorarea elementelor matricei. Dezavantajul constă însă în viteza scăzută de execuție a programului în momentul alocării și eliberării memoriei (se fac $m+1$ alocări și tot atâtea eliberări de memorie).

Propunem în continuare o altă metodă de alocare a memoriei pentru o matrice cu numai două alocări de memorie (și două eliberări).

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

int main(void)
{
    int m,n;
    float **a;

    cout<<"Dati dimensiunile matriciei: ";
    cin>>m>>n;

    if (!(a = new float*[m]))
    {
        cerr<<"Eroare! Memorie insuf." <<endl;
        return 1;
    }
    if(!(a[0] = new float[m*n]))
    {
        cerr<<"Eroare! Memorie insuf." <<endl;
        return 1;
    }
}
```

```

    for (int i=1;i<m;i++)
        a[i]=a[i-1]+n;

//....

delete [] a[0];
delete [] a;
return 0;
}

```

În cazul celei de a doua metode, întâi alocăm de asemenea memorie pentru a reține cele m adrese de început ale liniilor matricei, după care alocăm o zonă de memorie continuă necesară stocării tuturor celor $m \times n$ elemente ale matricei (întâi vom reține elementele primei linii, apoi elementele celei de a doua linii etc.). Adresa de început a zonei de memorie alocate pentru elementele matricei este reținută în pointerul $a[0]$. În $a[1]$ se reține adresa celei de a $(n+1)$ -a căsuțe de memorie ($a[1]=a[0]+n$), adică începutul celei de-a doua linii a matricei. În general, în $a[i]$ se reține adresa de început a liniei $i+1$, adică $a[i]=a[i-1]+n=a[0]+i*n$. Schema de alocare a memoriei este prezentată în figura 2.

Este evident că al doilea mod de alocare a memoriei este mai rapid decât primul (cu numai două alocări și două eliberări) și, cum calculatoarele din prezent sunt înzestrate cu memorii RAM de capacitate foarte mare, alocarea unei zone mari și continue de memorie nu mai reprezintă un dezavantaj. Așa că în practică preferăm a doua modalitate de alocare dinamică a memorie pentru o matrice.

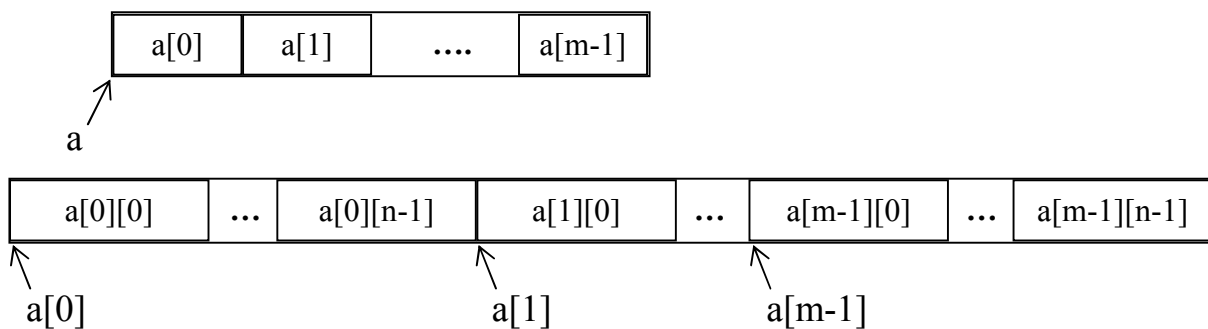


Fig. 2: A doua schemă de alocare a memoriei pentru o matrice

În final trebuie să recunoaștem însă că alocarea dinamică a memoriei pentru o matrice în alte limbaje de programare (cum ar fi Java sau C#) este mai ușoară decât în C++, deoarece ea se poate face printr-o singură utilizare a operatorului **new**.

Rezumat

C++ oferă o alternativă mai elegantă și modernă pentru alocarea dinamică a memoriei, folosind operatorii *new* și *delete*. Cu ajutorul acestor operatori alocăm memorie mai ușor, fără a mai fi nevoie de conversii și fără apeluri de funcții.

Teme

1. Să se aloce dinamic memorie pentru un vector de vectori de elemente de tip *double* cu următoarea proprietate: primul vector are un element, al doilea are două elemente, în general al k -lea vector are k elemente, $k \in \{1, 2, \dots, n\}$. Să se citească de la tastatură n (numărul de vectori) precum și elementele vectorilor. Să se construiască un nou vector format cu mediile aritmetice ale celor n vectori. În final să se elibereze toate zonele de memorie alocate dinamic.
2. De la tastatură se citește un vector a cu n elemente întregi pozitive. Să se aloce memorie pentru un vector de vectori cu elemente întregi. Primul vector are $a[0]$ elemente, al doilea are $a[1]$ elemente, în general al k -lea vector are $a[k-1]$ elemente ($k \in \{1, 2, \dots, n\}$). Să se citească de la tastatură elementele vectorilor. Să se construiască un nou vector (alocat tot dinamic) format cu elementele pătrate perfecte ale celor n vectori. În final se eliberează toate zonele de memorie alocate dinamic.
3. Să se aloce dinamic memorie pentru o matrice tridimensională de dimensiuni m , n și p de elemente întregi, unde m , n și p sunt valori întregi pozitive citite de la tastatură. Să se citească de la tastatură elementele matricii. Să se construiască un vector de triplete (i, j, k) , unde i, j și k ($i \in \{0, 1, \dots, m-1\}$, $j \in \{0, 1, \dots, n-1\}$, $k \in \{0, 1, \dots, p-1\}$) sunt indicii elementelor matricii care sunt cele mai apropiate de un pătrat perfect. În final să se elibereze toate zonele de memorie alocate dinamic.
4. Scrieți funcții pentru introducerea unui element într-o stivă de caractere, scoaterea unui element din stivă, afișarea conținutului stivei și eliberarea memoriei ocupate de stivă. Stiva se va memora dinamic folosind pointeri către tipul *struct*.
5. Scrieți aceleași funcții pentru o coadă.
6. Scrieți aceleași funcții pentru o coadă circulară.
7. Scrieți o funcție pentru introducerea unei valori reale într-un arbore binar de căutare și o funcție pentru parcurgerea în ordine a arborelui binar. Folosiți aceste funcții pentru a sorta un vector de numere reale citit de la tastatură. Pentru memorarea arborelui se vor folosi pointeri către tipul *struct*.

1.8. Funcții în C++

Obiective

În acest capitol ne propunem să studiem elementele introduse de C++ cu privire la modul de scriere al funcțiilor. În C++ putem avea funcții cu același nume și cu parametrii diferiți, valori implicite pentru parametri. Transmiterea parametrilor se poate face și prin referință. Vom prezenta și noțiunile de funcție *inline* și funcție șablon.

1.8.1. Funcții cu același nume și cu parametrii diferiți

În C++ se pot scrie mai multe funcții cu același nume, dar cu parametri diferiți (ca număr sau/și ca tip), în engleza *overloading*. La apelul unei funcții se caută varianta cea mai apropiată de modul de apelare (ca număr de parametri și ca tip de date al parametrilor).

De exemplu, putem scrie trei funcții cu același nume care calculează maximumul dintre două, respectiv trei valori:

```
# include <iostream.h>

int max(int x,int y)
{
    if (x>y) return x;
    return y;
}

int max(int x,int y,int z)
{
    if (x>y)
    {
        if (x>z) return x;
        return z;
    }
    if (y>z) return y;
    return z;
}

double max(double x,double y)
{
    if (x>y) return x;
    return y;
}

void main(void)
{
    int a=1,b=2,c=0,max1,max2;
    float A=5.52f,B=7.1f,max3;
    double A2=2,B2=1.1,max4;

    max1=max(a,b);          // apelul primei functii
    max2=max(a,b,c);        // apelul celei de-a doua fct
    max3=(float)max(A,B);    // apelul functiei 3
    max4=max(A2,B2);        // apelul tot al functiei 3
    cout<<max1<<" ";<<max2<<" ";
    cout<<max3<<" ";<<max4<<endl;
}
```

1.8.2. Transmiterea parametrilor prin referință

În C transmiterea parametrilor în funcție se face prin valoare (pentru cei de intrare) sau prin adresă (pentru cei de ieșire). Transmiterea parametrilor prin adresă este pretențioasă (la apel

suntem obligați în general să utilizăm operatorul adresa &, iar în corpul funcției se folosește operatorul *).

În C++ transmiterea parametrilor de ieșire (care se returnează din funcții), se poate face într-o manieră mult mai elegantă, și anume prin referință. În definiția funcției, fiecare parametru transmis prin referință este precedat de semnul &.

Dăm ca exemplu interschimbarea valorilor a două variabile în ambele forme (transmitere prin adresă și prin referință).

```
# include <iostream.h>

void intersch(int* a, int* b)      // transmitere prin adr.
{
    int c=*a;
    *a=*b;
    *b=c;
}

void intersch(int& a, int& b)      // transmitere prin ref.
{
    int c=a;
    a=b;
    b=c;
}

void main()
{
    int x=1,y=2;

    cout<<"primul nr: "<<x<<"", al doilea nr: "<<y<<endl;
    intersch(&x,&y);      // apelul primei functii
    cout<<"primul nr: "<<x<<"", al doilea nr: "<<y<<endl;
    intersch(x,y);      // apelul celei de-a doua functii
    cout<<"primul nr: "<<x<<"", al doilea nr: "<<y<<endl;
}
```

Ne propunem acum să scriem o funcție care primește ca parametru un vector de valori întregi și care construiește și returnează alți doi vectori formați cu elementele nenegative și respectiv cu cele negative ale vectorului inițial. Mai mult, după separare se eliberează zona de memorie ocupată de vectorul inițial.

Se știe că în C cand alocăm memorie (sau în general atunci când schimbăm adresa reținută într-un pointer primit ca argument), adresa de memorie în general se returnează ca valoare a funcției și nu printre parametrii funcției (vezi de exemplu funcțiile C de alocare a memoriei: *calloc*, *malloc*, *realloc*). Cum procedăm atunci când alocăm mai multe zone de memorie în interiorul funcției și vrem să returnăm adresele spre zonele alocate? Acesta este și cazul problemei propuse mai sus. Singura soluție oferită de C este scrierea unor parametri ai funcției de tip pointer către pointer. Iată rezolvarea problemei în această manieră pentru problema separării elementelor nenegative de cele negative:

```
# include <iostream.h>

void separare(int m,int **a,int *n,int **b,int *p,int **c)
```

```

{
    *n=0;
    *p=0;
    for (int i=0;i<n;i++)
        if ((*a)[i]>=0) (*n)++;
        else (*p)++;
    *b=new int[*n];
    *c=new int[*p];
    int k=0,h=0;
    for (i=0;i<n;i++)
        if ((*a)[i]>=0) (*b)[k++]=(*a)[i];
        else (*c)[h++]=(*a)[i];
    delete [] *a;
}

void main()
{
    int i,n,n1,n2,*a,*a1,*a2;

    cout<<"Nr. de elemente al sirului: ";
    cin>>n;
    a=new int[n];
    cout<<"Dati elemente al sirului:"<<endl;
    for (i=0;i<n;i++)
    {
        cout<<"a["<<(i+1)<<"]="";
        cin>>a[i];
    }
    separare(n,&a,&n1,&a1,&n2,&a2); // transmitere prin adr
    cout<<"Sirul elementelor nenegative: ";
    for (i=0;i<n1;i++)
        cout<<a1[i]<<" ";
    cout<<endl<<"Sirul elementelor negative: ";
    for (i=0;i<n2;i++)
        cout<<a2[i]<<" ";
    delete [] a1;
    delete [] a2;
}

```

Pentru a înțelege obligativitatea folosirii parantezelor în funcția *separare* din exemplul de mai sus trebuie să cunoaștem ordinea de aplicare a operatorilor într-o expresie. Astfel, operatorul de incrementare ++ are prioritatea cea mai mare, el se aplică înaintea operatorului * (valoare de la o adresă) pus în fața pointerului. De asemenea, operatorul [] (de referire la un element al unui șir) are prioritate mai mare decât * aplicat unui pointer. De aceea am folosit paranteze pentru a indica ordinea în care dorim să aplicăm acești operatori în situațiile: (*a)[i] și respectiv (*n)++. Să observăm și faptul că la atribuirea *n=0 nu este necesară utilizarea parantezelor, deoarece operatorul = are prioritate mai mică decât * aplicat pointerilor.

După cum se vede descrierea funcției *separare* cu transmitere a vectorilor prin adresă (stil C) este foarte de pretențioasă. Trebuie să fim foarte atenți la o mulțime de detalii datorate în special necesității utilizării operatorului * când ne referim în corpul funcției la parametrii transmiși prin adresă.

Să vedem în continuare cum putem rescrie mult mai elegant funcția *separare* cu transmitere a vectorilor prin referință către pointer (așa cum este posibil numai în C++):

```
# include <iostream.h>

void separare(int m,int *&a,int &n,int *&b,int &p,int *&c)
{
    n=0;
    p=0;
    for (int i=0;i<m;i++)
        if (a[i]>=0) n++;
        else p++;
    b=new int[n];
    c=new int[p];
    int k=0,h=0;
    for (i=0;i<m;i++)
        if (a[i]>=0) b[k++]=a[i];
        else c[h++]=a[i];
    delete [] a;
}

void main()
{
    int i,n,n1,n2,*a,*a1,*a2;

    cout<<"Nr. de elemente al sirului: ";
    cin>>n;
    a=new int[n];
    cout<<"Dati elemente al sirului:"<<endl;
    for (i=0;i<n;i++)
    {
        cout<<"a["<<(i+1)<<"]=" ";
        cin>>a[i];
    }
    separare(n,a,n1,a1,n2,a2); // transmitere prin referinta
    cout<<"Sirul elementelor nenegative: ";
    for (i=0;i<n1;i++)
        cout<<a1[i]<<" ";
    cout<<endl<<"Sirul elementelor negative: ";
    for (i=0;i<n2;i++)
        cout<<a2[i]<<" ";
    delete [] a1;
    delete [] a2;
}
```

1.8.3. Valori implicite pentru parametrii funcțiilor

În C++ există posibilitatea ca la definirea unei funcții o parte dintre parametri (transmiși prin valoare) să primească valori implicite. În situația în care lipsesc argumente la apelul funcției, se iau valorile implicite dacă există pentru acestea. Numai o parte din ultimele argumente din momentul apelului unei funcții pot lipsi și numai dacă există valorile implicite

pentru acestea în definiția funcției. Nu poate lipsi de exemplu penultimul argument, iar ultimul să existe în momentul apelului funcției.

Dăm un exemplu simplu în care scriem o funcție *inc* pentru incrementarea unei variabile întregi (similară procedurii cu același nume din limbajele Pascal și Delphi):

```
void inc(int &x, int i=1)
{
    x+=i;
}
```

Funcția *inc* poate fi apelată cu unul sau doi parametri. Astfel, apelul *inc(a, 5)* este echivalent cu *x+=5* (în corpul funcției variabila *i* ia valoarea 5, valoare transmisă din apelul funcției). Dacă apelăm însă funcția sub forma *inc(a)*, atunci pentru *i* se ia valoarea implicită 1, situație în care *x* se mărește cu o unitate.

Scrieți o funcție care primește 5 parametri de tip *int* care returnează maximul celor 5 valori. Dați valori implicite parametrilor așa încât funcția să poată fi folosită pentru a calcula maximul a două numere întregi (când se apelează cu 2 parametri), a trei, patru și respectiv cinci valori întregi.

1.8.4. Funcții *inline*

În C++ există posibilitatea declarării unei funcții ca fiind *inline*. Fiecare apel al unei funcții *inline* este înlocuit la compilare cu corpul funcției. Din această cauză funcțiile *inline* se aseamănă cu macrocomenzile. Spre deosebire însă de macrocomenzi, funcțiile *inline* au tip pentru parametri și pentru valoarea returnată. De fapt, ele se declară și se descriu ca și funcțiile obișnuite, numai că în fața definiției se pune cuvântul rezervat *inline*. Modul de apel al macrocomenzilor diferă de cel al funcțiilor *inline*. În acest sens dăm un exemplu comparativ în care scriem o macrocomandă pentru suma a două valori și respectiv o funcție *inline* pentru suma a două valori întregi.

```
# include <iostream.h>

# define suma(a,b) a+b          // macrocomanda

inline int suma2(int a, int b)    // functie inline
{
    return a+b;
}

void main()
{
    int x;
    x=2*suma(5,3);
    cout<<x<<endl;
    x=2*suma2(5,3);
    cout<<x<<endl;
}
```

Pe ecran se vor afișa valorile 13 și respectiv 16. Primul rezultat poate fi pentru unii neașteptat. Dacă suntem familiarizați cu modul de utilizare al macrocomenzilor rezultatul nu mai

este însă deloc surprinzător. La compilare, apelul `suma(5, 3)` este înlocuit efectiv în cod cu `5+3`, ceea ce înseamnă că variabilei `x` i se va atribui valoarea `2*5+3`, adică 13.

Din cauză că apelurile funcției `inline` se înlocuiesc cu corpul ei, codul funcției `inline` trebuie să fie în general de dimensiuni mici. În caz contrar și/sau dacă apelăm des în program funcțiile `inline`, dimensiunea executabilului va fi mai mare.

De reținut este faptul că în cazul compilatorului Borland C++ nu se acceptă instrucțiuni repetitive și nici instrucțiuni *throw* (vezi capitolul dedicat tratării excepțiilor) în corpul funcției `inline`. Dacă încercăm totuși utilizarea lor în corpul unei funcții declarate `inline`, atunci la compilare funcția va fi considerată obișnuită (ignorându-se practic declarația *inline*) și se va genera un *warning*.

Funcțiile `inline` sunt foarte des utilizate în descrierea claselor. Astfel, funcțiile ”mici”, fără cicluri repetitive, pot fi descrise `inline`, adică direct în corpul clasei.

1.8.5. Funcții șablon (template)

Unul dintre cele mai frumoase suporturi pentru programare generică este oferit de limbajul C++ prin intermediul funcțiilor și claselor șablon (*template*). Astfel, în C++ putem scrie clase sau funcții care pot funcționa pentru unul sau mai multe tipuri de date nespecificate. Să luăm spre exemplu sortarea unui vector. Ideea algoritmului este aceeași pentru diverse tipuri de date: întregi, reale, string-uri etc. Fără a folosi șabloane ar trebui să scriem câte o funcție de sortare pentru fiecare tip de date.

Înainte de fiecare funcție șablon se pune cuvântul rezervat *template* urmat de o enumerare de tipuri de date generice (precedate fiecare de cuvântul rezervat *class*). Enumerarea se face între semnele `<` (mai mic) și `>` (mai mare):

```
template <class T1, class T2, ... >
tipret NumeFctSablon(parametri)
{
    // corpul functiei sablon
}
```

`T1, T2, ...` sunt tipurile generice de date pentru care scriem funcția șablon. Este esențial de reținut faptul că toate tipurile de date generice trebuie folosite în declararea parametrilor funcției.

În cele mai multe cazuri se folosește un singur tip generic de date.

Iată câteva exemple simple de funcții șablon:

```
template <class T>
T max(T x, T y)
{
    if (x>y) return x;
    return y;
}

template <class T>
void intersch(T &x, T &y)
{
    T aux=x;
    x=y;
    y=aux;
}
```

```

template <class T1,class T2>
void inc(T1 &x,T2 y)
{
    x+=y;
}

template <class T>
T ma(int n,T *a)
{
    T s=0;
    for (int i=0;i<n;i++) s+=a[i];
    return s/n;
}

```

Dacă în interiorul aceluiași program avem apeluri de funcții șablon pentru mai multe tipuri de date, atunci pentru fiecare dintre aceste tipuri de date compilatorul generează câte o funcție în care tipurile generice de date se înlocuiesc cu tipurile de date identificate la întâlnirea apelului. De asemenea, la compilare se verifică dacă sunt posibile instanțele funcțiilor șablon pentru tipurile de date identificate. De exemplu, compilarea codului funcției șablon *inc* nu generează erori, dar tentativa de apelare a ei cu doi parametri de tip *string* se soldează cu eroare la compilare pentru că cele două string-uri se transmit prin doi pointeri (către *char*), iar operatorul += nu este definit pentru doi operanzi de tip pointer.

Dăm în continuare câteva posibile apeluri ale funcțiilor șablon de mai sus:

```

int i=2,j=0,k;
double m,a[4]={1.5,-5E2,8,0},x=2,y=5.2;
char *s1="Un string",*s2="Alt string",*s;

k=max(i,j);           // T este int
m=max(x,y);           // T este double
intersch(i,j); // T este int
intersch(x,y); // T este double
inc(x,i);              // T1 este double, T2 este int
m=ma(4,a);             // T este double
s=max(s1,s2);          // apel incorect, T nu poate fi char*

```

Funcțiile șablon și clasele șablon alcătuiesc fiecare câte o clasă de funcții, respectiv de clase, în sensul că ele au aceeași funcționalitate, dar sunt definite pentru diverse tipuri de date. O funcție *template* de sortare putem spune că este de fapt o clasă de funcții de sortare pentru toate tipurile de date care suportă comparare. În cazul sortării, tipului nespecificat este cel al elementelor vectorului.

Clasele șablon le vom prezenta într-un capitol următor.

Rezumat

Am studiat principalele elemente introduse de limbajul C++ cu privire la modul de redactare al funcțiilor. Astfel, putem avea funcții cu același nume și parametri diferiți, valori implicite pentru funcții, funcții *inline*, parametri de ieșire pot fi transmiși prin referință într-o manieră mult mai elegantă și mai ușor de redactat. De asemenea, funcțiile șablon (alături de

clasele șablon) oferă unul dintre cele mai frumoase suporturi pentru programarea generică (cod care să funcționeze pentru diverse tipuri de date).

Teme

Propunem aplicativ la transmiterea parametrilor prin referință scrierea a două funcții:

1. Funcție care primește ca primi parametrii lungimea unui vector de numere întregi precum și pointerul către acest vector. Să se construiască un alt vector (alocat dinamic) format cu elementele care sunt numere prime ale vectorului inițial. Să se returneze prin referință lungimea vectorului construit precum și pointerul către noul vector.
2. Funcție care primește ca primi parametri datele despre o matrice de dimensiuni m și n . Pornind de la această matrice să se construiască un vector format cu elementele matricii luate în spirală pornind de la elementul de indici 0 și 0 și mergând în sensul acelor de ceasornic. Funcția va returna prin referință pointerul către vectorul construit. Menționăm faptul că, după construcția vectorului, în interiorul funcției se va elibera memoria ocupată de matrice.

De exemplu, din matricea $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$ se obține vectorul (1, 2, 3, 4, 8, 12, 11,

10, 9, 5, 6, 7).

Să scrie funcții șablon pentru:

1. Minimul a trei valori;
2. Maximul elementelor unui șir;
3. Căutare secvențială într-un șir;
4. Căutare binară într-un șir;
5. Interclasare a două șiruri sortate crescător;
6. Sortare prin interclasare (utilizând funcția de interclasare de mai sus);
7. Produs a două matrici;
8. Ridicare matrice la o putere.

1.9. Supraîncărcarea operatorilor

Obiective

În C++ există posibilitatea supraîncărcării operatorilor (în engleză *overloading*), adică un operator poate fi definit și pentru alte tipuri de date decât cele pentru care există deja definit. Vom vedea că supraîncărcarea operatorilor oferă o alternativă la scrierea unor funcții pentru diverse operații. Un operator este mult mai ușor introdus într-o expresie, oferind eleganță și abstractizare codului.

Nu toți operatorii pot fi supraîncărcați. Operatorii care nu pot fi redefiniți și pentru alte tipuri de date sunt:

1. `x.y` operator acces la membru
2. `x.*y` operator acces la date de la adresa dată de un câmp de tip pointer
3. `x::y` operator de rezoluție
4. `x?y:z` operator ternar ?:
5. `#` operator directivă preprocesor
6. `##` operator preprocesor de alipire token-i

Lista operatorilor care pot fi supraîncărcați este:

1. Operatorii unari de incrementare și decrementare: `++x`, `x++`, `--x`, `x--`
2. Operatorul adresă `&x`
3. Operatorul de redirecționare `*x`
4. Operatorii unari semn plus și minus: `+x`, `-x`
5. Operatorul not la nivel de bit `~x`
6. Operatorul not logic `!x`
7. Operatorul pt. aflarea numărului de octeți pe care se reprezintă valoarea unei expresii `sizeof expresie`
8. Operatorul de conversie de tip `(type) x`
9. Operatorii binari aritmetici: `+`, `-`, `*`, `/`, `%`
10. Operatorii de deplasare (shift-are) pe biți, respectiv pentru introducere, scoatere în / dintr-un flux C++: `x<<y`, `x>>y`
11. Operatorii relaționari: `<`, `>`, `<=`, `>=`, `=`, `!=`
12. Operatorii binari la nivel de bit: `x&y` (și), `x^y` (sau exclusiv), `x|y` (sau)
13. Operatorii logici `&&` (și), `||` (sau)
14. Operatorul de atribuire `=`
15. Operatorul de atribuire combinat cu un operator binar: `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `^=`, `|=`
16. Operatorii de acces la date: `x[y]` (indice), `x->y` (membru y de la adresa x), `x->*y` (referire membru pointer)
17. Operator de tip apel funcție `x(y)`
18. Operatorul virgulă `x,y`
19. Operatorii de alocare și eliberare dinamică a memoriei: `new`, `delete`.

După cum putem observa din lista de mai sus, operatorii care pot fi supraîncărcați sunt unari sau binari, adică au aritatea 1 sau 2. De altfel, singurul operator C/C++ ternar ?: nu poate fi redefinit.

Un operator se definește asemănător cu o funcție. La definirea operatorului trebuie să se țină însă cont de aritatea lui, care trebuie să coincidă cu numărul parametrilor, dacă funcția este externă unei clase, iar în loc de numele funcției apare cuvântul rezervat *operator* urmat de simbolul sau simbolurile care caracterizează acel operator:

```
tipret operator<simbol(uri)>(parametri)
{
    // corpul operatorului
}
```

Să considerăm următoarea structură în care vom memora elementele unei mulțimi de numere întregi:


```
struct Multime
{
    int n,e[1000];
};
```

În câmpul n vom reține numărul de elemente al mulțimii, iar în e vom reține elementele mulțimii.

În C++ la tipul *struct Multime* ne putem referi direct sub forma *Multime*.

Pentru tipul *Multime* vom arăta vom supraîncărca următorii operatori: $+$ pentru reuniunea a două mulțimi și pentru adăugarea unui element la o mulțime, $<<$ pentru introducerea elementelor mulțimii într-un flux de ieșire și $>>$ pentru extragerea elementelor unei mulțimi dintr-un flux de intrare.

Cea mai simplă metodă (de implementat) pentru reuniunea a două mulțimi este:

```
int cautSecv(int x,int n,int *a)
{
    for (int i=0;i<n;i++)
        if (x==a[i]) return 1;
    return 0;
}

Multime operator +(Multime a,Multime b)
{
    Multime c;

    for (int i=0;i<a.n;i++) c.e[i]=a.e[i];
    c.n=a.n;
    for (i=0;i<b.n;i++)
        if (!cautSecv(b.e[i],a.n,a.e))
            c.e[c.n++]=b.e[i];
    return c;
}
```

Complexitatea algoritmului folosit mai sus pentru a reuni mulțimile a și b este evident $O(a.n \times b.n)$. Se poate însă și mai bine. În loc de căutarea secvențială a elementului $b.e[i]$ în vectorul $a.e$ putem aplica o căutare rapidă, bineînțeles dacă sortăm rapid în prealabil vectorul $a.e$. Complexitatea algoritmului devine: $O(a.n \times \log(a.n) + b.n \times \log(a.n)) = O((a.n + b.n) \times \log(a.n))$. Dacă reunim însă mulțimea b cu a (în ordine inversă), atunci complexitatea devine $O((a.n + b.n) \times \log(b.n))$. Este evident că, pentru a îmbunătăți complexitatea algoritmului, vom reuni a cu b dacă $b.n > a.n$ și, respectiv b cu a în situația în care $a.n > b.n$. Complexitatea devine atunci $O((a.n + b.n) \times \log(\min\{b.n, a.n\})) = O(\max\{a.n, b.n\} \times \log(\min\{b.n, a.n\}))$.

Pentru sortarea mulțimii cu mai puține elemente vom aplica algoritmul de sortare prin interclasare pentru că el are în orice situație complexitatea $O(m \times \log(m))$, unde m este numărul de elemente.

Reuniunea rapidă a două mulțimi este:

```
void intercls(int m,int *a,int n,int *b,int *c)
{
    int i=0,j=0,k=0,l;

    while (i<m && j<n)
```

```

        if (a[i]<b[j]) c[k++]=a[i++];
        else c[k++]=b[j++];
    for(l=i;l<m;l++) c[k++]=a[l];
    for(l=j;l<n;l++) c[k++]=b[l];
}

void sortIntercls(int s,int d,int *a,int *b)
{
    if (s==d) {b[0]=a[s]; return;}
    int m=(s+d)/2,*a1,*a2;

    a1=new int[m-s+1];
    a2=new int[d-m];
    sortIntercls(s,m,a,a1);
    sortIntercls(m+1,d,a,a2);
    intercls(m-s+1,a1,d-m,a2,b);
    delete [] a1;
    delete [] a2;
}

int cautBin(int x,int s,int d,int *a)
{
    if (s==d)
    {
        if (a[s]==x) return 1;
        else return 0;
    }
    int m=(s+d)/2;
    if (a[m]==x) return 1;
    if (x<a[m])
        return cautBin(x,s,m,a);
    return cautBin(x,m+1,d,a);
}

Multime operator +(Multime a,Multime b)
{
    int i;
    Multime c;

    if (a.n<b.n)
    {
        if(a.n)sortIntercls(0,a.n-1,a.e,c.e);
        c.n=a.n;
        for (i=0;i<b.n;i++)
            if (!cautBin(b.e[i],0,a.n-1,c.e))
                c.e[c.n++]=b.e[i];
    }
    else
    {
        if(b.n)sortIntercls(0,b.n-1,b.e,c.e);
        c.n=b.n;
        for (i=0;i<a.n;i++)

```

```

        if (!cautBin(a.e[i],0,b.n-1,c.e))
            c.e[c.n++]=a.e[i];
    }
    return c;
}

```

Să implementăm acum operatorul + pentru adăugarea unui element la o mulțime:

```

Multime operator +(Multime a,int x)
{
    Multime b;

    for (int i=0;i<a.n;i++) b.e[i]=a.e[i];
    b.n=a.n;
    if (!cautSecv(x,b.n,b.e)) b.e[b.n++]=x;
    return b;
}

```

Implementarea operatorilor << și >> este:

```

ostream& operator <<(ostream& fl,Multime& a)
{
    for (int i=0;i<a.n;i++) fl<<a.e[i]<<" ";
    return fl;
}

istream& operator >>(istream& fl,Multime& a)
{
    fl>>a.n;
    for (int i=0;i<a.n;i++) fl>>a.e[i];
    return fl;
}

```

Pentru testarea operatorilor care i-am scris pentru tipul *Multime* propunem următoarea funcție principală:

```

void main()
{
    int x;
    Multime a,b,c;

    cout<<"Dati prima multime:"<<endl;
    cin>>a;
    cout<<"Dati a doua multime:"<<endl;
    cin>>b;
    cout<<"Dati un numar intreg:";
    cin>>x;
    c=a+b+x;
    cout<<"Reuniunea este: "<<c<<endl;
}

```

Pentru tipul *Multime* nu putem supraîncărca operatorul = pentru că el este definit deja pentru operanzi de tipul *struct* !

În final prezentăm câteva reguli de care trebuie să ținem seama atunci când supraîncărcăm un operator:

1. Trebuie păstrată aritatea operatorului. Astfel, spre exemplu, aritatea operatorului de adunare + este 2 (operator binar). Dacă supraîncărcăm acest operator, atunci el va trebui gândit și implementat să funcționeze pentru doi operanzi.
2. Modul de folosire al operatorilor se păstrează. De exemplu, operatorul de adunare + va fi folosit numai sub forma *obiect1 + obiect2*, iar operatorul de conversie de tip se va folosi sub forma *(tip)obiect*.
3. Nu pot fi definiți noi operatori, ci pot fi supraîncărcați numai cei enumerați în lista de mai sus. De exemplu nu putem defini operatorul **, care în alte limbaje (cum ar fi Basic) este folosit pentru ridicare la putere.
4. Se păstrează prioritatea de aplicare a operatorilor și după ce aceștia au fost supraîncărcați.
5. Nu se păstrează comutativitatea, astfel $a * b$ nu este același lucru cu $b * a$.
6. Un operand se poate defini cel mult o dată pentru un tip de date (dacă este unar) și cel mult o dată pentru o pereche de tipuri de date (dacă este binar).
7. Un operator nu poate fi supraîncărcat pentru tipuri de date pentru care există deja definit (de exemplu operatorul + nu poate fi redefinit pentru doi operanzi întregi).
8. Pentru a se face distincție între forma prefixată și cea post fixată a operatorilor de incrementare și decrementare, la definirea formei postfixate se pune un parametru suplimentar de tip *int*.
9. Un operator, ca în cazul unei funcții, poate fi definit în 2 feluri: ca fiind membru unei clase sau ca fiind extern unei clase, dar nu simultan sub ambele forme pentru aceleași tipuri de date.
10. Operatorii <<, >> se definesc de obicei ca funcții externe unei clase pentru că primul operand este de obicei un flux.

Facem mențiunea că în cele mai multe cazuri operatorii sunt supraîncărcați pentru obiecte în momentul în care scriem o clasă nouă. Vom ilustra acest lucru la momentul potrivit.

Rezumat

În C++ operatorii pot fi redefiniți și pentru alte tipuri de date decât cele pentru care există deja. Supraîncărcarea unui operator se face foarte asemănător cu descrierea unei funcții. Practic diferența esențială constă în faptul că numele funcției ce descrie operatorul este obligatoriu format din cuvântul rezervat *operator* urmat de simbolurile ce definesc acel operator.

Teme

Propunem cititorului să implementeze pentru tipul *Multime* definit mai sus următorii operatori:

1. operatorul – pentru diferența a două mulțimi și pentru scoaterea unui element dintr-o mulțime
2. operatorul * pentru intersecția a două mulțimi
3. operatorii +=, -=, *=
4. operatorul ! care returnează numărul de elemente al mulțimii

5. operatorii `<=` și `<` pentru a verifica dacă o mulțime este inclusă, respectiv strict inclusă în altă mulțime
6. operatorii `>=` și `>` pentru a verifica dacă o mulțime conține, respectiv conține strict altă mulțime
7. operatorul `<` pentru a verifica dacă un element aparține unei mulțimi
8. operatorii `==` și `!=` pentru a verifica dacă două mulțimi coincid ca și conținut, respectiv sunt diferite.

Scrieți tipul *MulTime*, funcțiile (fără *main*) și operatorii de mai sus într-un fișier cu numele *MulTime.cpp*. În viitor, de fiecare dată când veți avea nevoie într-o aplicație să lucrați cu mulțimi veți putea include acest fișier.

1.10. Tratarea excepțiilor

Obiective

Ne propunem să vedem ce propune C++ în comparație cu limbajul C pentru tratarea situațiilor în care apar excepții (erori) pe parcursul execuției programului. Vom vedea de asemenea cum putem genera propriile excepții cu ajutorul instrucțiunii *throw*.

Excepțiile se referă în general la situații deosebite ce pot apărea în program, în special în cazul erorilor ce apar în urma alocării de memorie, deschiderii unui fișier, calculelor matematice (de exemplu împărțire prin 0 sau logaritm dintr-o valoare care nu este pozitivă) etc. Când este detectată o excepție este dificil de decis ce trebuie făcut în acel moment. În cele mai multe cazuri se preferă afișarea unui mesaj și se părăsește programul.

Excepțiile se împart în excepții **sincrone** și **asincrone**. Excepțiile ce pot fi detectate cu ușurință sunt cele de tip *sincron*. În schimb, excepțiile *asincrone* sunt cauzate de evenimente care nu pot fi controlate din program.

Este mai puțin cunoscut faptul că în limbajul C există posibilitatea tratării excepțiilor folosind funcțiile *setjmp* și *longjmp*:

```
int setjmp(jmp_buf jmbb)
void longjmp(jmp_buf jmbb, int retval)
```

Funcția *setjmp* salvează starea (contextul) execuției (stiva) programului în momentul în care este apelată și returnează valoarea 0, iar funcția *longjmp* utilizează starea salvată cu *setjmp* pentru a se putea reveni cu stiva de execuție a programului din momentul salvării. Starea programului se salvează într-un buffer de tip *jmp_buf*, care este definit în fișierul antet *setjmp.h*. Când se apelează funcția *longjmp*, se face practic un salt cu execuția programului în locul în care s-a apelat funcția *setjmp*. În acest moment funcția *setjmp* returnează valoarea 1. Funcția *longjmp* se comportă asemănător cu un apel *goto* la locul marcat de *setjmp*.

Funcția *longjmp* se apelează cu al doilea parametru având valoarea 1. Dacă se încearcă apelul cu o altă valoare, parametrul *retval* va fi setat tot ca fiind 1.

Dăm un exemplu de tratare a excepțiilor legate de deschiderea unui fișier așa cum se face în C (cu ajutorul celor două funcții):

```
# include <conio.h>
# include <stdio.h>
# include <stdlib.h>
# include <process.h>
# include <setjmp.h>
```

```

void mesaj_eroare()
{
    perror("Au fost erori in timpul executiei!");
    getch();
}

void mesaj_ok()
{
    perror("Program incheiat cu succes!");
    getch();
}

void main()
{
    char numef[100];
    FILE *fis;
    jmp_buf stare;

    if (!setjmp(stare)) // salvare stare
    {
        printf("Dati numele unui fisier: ");
        scanf("%s", numef);
        fis=fopen(numef, "rb");
        if (fis==NULL)
            longjmp(stare,1); // salt la setjmp si
                                // se trece pe ramura else
    }
    else
    {
        perror("Eroare! Nu am putut deschide fis.");
        atexit(mesaj_eroare); // mesaj er. parasire progr.
        exit(1);
    }
    fclose(fis);
    atexit(mesaj_ok); // mesaj parasire program cu succes
}

```

În exemplul de mai sus am folosit funcția *atexit* care specifică funcția ce se va apela imediat înainte de părăsirea programului.

Dacă dorim să semnalăm faptul că un program s-a încheiat cu insucces, putem apela funcția *abort* în locul funcției *exit*. Funcția *abort* părăsește programul și returnează valoarea 3, nu înainte însă de a afișa mesajul *Abnormal program termination*.

În C++ tratarea excepțiilor este mult mai modernă. Tratarea excepțiilor se face cu ajutorul instrucțiunii *try*. După cuvântul rezervat *try* urmează un bloc de instrucțiuni neapărat delimitat de acolade. Se încearcă execuția (de unde și denumirea) instrucțiunilor din blocul *try* și dacă se generează o excepție, atunci execuția instrucțiunilor blocului *try* este întreruptă și excepția este captată (prinsă) și tratată eventual pe una din ramurile *catch* ce urmează după blocul *try*. Facem mențiunea că instrucțiunile ce se execută pe o ramură *catch* sunt de asemenea delimitate obligatoriu de acolade.

Dăm un exemplu de tratare a excepției ce apare în urma împărțirii prin zero:

```
#include<iostream.h>

void main()
{
    int a=1,b=0;

    try
    {
        cout<<(a/b)<<endl;
    }
    catch (...)
    {
        cerr<<"Impartire prin zero."<<endl;
    }
}
```

La prima apariție a unei excepții în interiorul blocului *try* se generează o valoare a cărei tip va face ca excepția să fie tratată pe o anumită ramură *catch*. Între parantezele rotunde ce urmează după cuvântul rezervat *catch* apare o variabilă sau apar trei puncte (. . .). Ramura *catch* cu trei puncte prinde orice excepție generată în blocul *try* care nu a fost prinsă de altă ramură *catch* de deasupra.

În momentul în care detectăm o excepție putem arunca o valoare cu ajutorul instrucțiunii *throw*.

Ne propunem să scriem o funcție pentru scoaterea rădăcinii pătrate dintr-un număr real cu ajutorul metodei numerice de înjumătățire a intervalului. Vom trata eventualele excepții ce pot apărea:

```
#include<iostream.h>

double fct(double x,double v)
{
    return x*x-v;
}

double radical(double v,double precizie)
{
    if (precizie<=0) throw precizie;
    if (v<0) throw "Radical din numar negativ.";
    if (!v || v==1) return v;

    double s,d,m,fs,fd,fm;

    if (v>1)
    {
        s=1;
        d=v;
    }
    else
    {
        s=v;
        d=1;
    }
}
```

```

    }
    fs=fct(s,v);
    fd=fct(d,v);
    while (d-s>precizie)
    {
        m=(s+d)/2;
        fm=fct(m,v);
        if (!fm) return m;
        if (fs*fm<0) {d=m; fd=fm;}
        else {s=m; fs=fm;}
    }
    return m;
}

void main()
{
    double x,p;

    cout<<"Dati un numar real: ";
    cin>>x;
    cout<<"Precizia calcularii radacinii patrute: ";
    cin>>p;
    try
    {
        double r=radical(x,p);
        cout<<"Radical din "<<r<<" este "<<r<<endl;
    }
    catch(char *mesajEroare)
    {
        cerr<<mesajEroare<<endl;
    }
    catch(double prec)
    {
        cerr<<"Precizia nu poate fi: ";
        cerr<<prec<<endl;
    }
}

```

În funcția *radical* parametrul real *precizie* indică distanța maximă între soluția numerică (valoarea aproximativă a rădăcinii pătrate din v) găsită și soluția exactă (în situația noastră valoarea exactă a rădăcinii pătrate din v). Funcția *radical* generează excepții într-una din situațiile: dacă v este negativ sau dacă precizia dată nu este un număr pozitiv.

În situația în care în funcția *radical* trimitem o valoare negativă sau zero pentru precizie, execuția instrucțiunilor din corpul funcției este întreruptă și se generează o valoare reală care reține precizia găsită ca fiind invalidă.

Dacă în funcția *radical* trimitem o valoare v care este negativă, atunci execuția instrucțiunilor din corpul funcției este întreruptă și se generează mesajul “*Radical din numar negativ.*”.

Evident că la apelul funcției *radical*, în situația în care precizia nu este pozitivă, se va intra pe a doua ramură *catch* a instrucțiunii *try*, iar dacă valoarea din care vrem să extragem

rădăcina pătrată este negativă se va ajunge pe prima ramură *catch*. În ambele situații se trimit mesaje în fluxul standard de erori (*cerr*), mesaje ce vor apărea pe ecran.

Dacă folosim ramura *catch* cu trei puncte, ea trebuie pusă ultima. Astfel, numai eventualele excepții care scapă de ramurile *catch* de deasupra ajung pe ramura *catch(...)*. Instrucțiunea *throw* poate să nu returneze nici o valoare, situație în care se ajunge cu execuția programului pe ramura *catch(...)*, dacă există.

După cum am văzut, instrucțiunea *throw* folosită în funcția *radical* are un efect asemănător cu un apel al instrucțiunii *return* în sensul că execuția instrucțiunilor din corpul funcției este întreruptă. Deosebirea este că instrucțiunea *throw* dacă returnează o valoare, atunci ea este interceptată numai în interiorul unui bloc *try*, iar instrucțiunea *return* returnează valoarea funcției.

Pot exista situații în care o excepție este identificată direct în interiorul blocului *throw*. Ea poate fi aruncată spre a fi captată de una dintre ramurile *catch*. În exemplul următor testăm dacă deîmpărțitul este zero înainte de a efectua împărțirea. Dacă deîmpărțitul este zero, aruncăm o excepție prin intermediul unui mesaj:

```
#include<iostream.h>

void main()
{
    int a=1,b=0;

    try
    {
        if (!b) throw "Impartire prin zero.";
        cout<<(a/b)<<endl;
    }
    catch (char *mesajEroare)
    {
        cerr<<mesajEroare<<endl;
    }
}
```

Menționăm faptul că instrucțiunea *try ... catch* nu este implementată în versiunile inferioare Boland C++.

Firma Microsoft a adăugat facilități noi instrucțiunii *try* în Visual C++. Este vorba de ramurile *except* și *finally*. Pentru detalii despre aceste ramuri consultați documentația MSDN pentru Visual Studio.

Rezumat

Am văzut ce este o excepție, cum se tratează o excepție în C și în C++. Tratarea excepțiilor în C++ se poate face într-o manieră modernă și elegantă folosind instrucțiunea *try ... catch*. Eventuala excepție generată în interiorul blocului *try* poate fi captată pe una din ramurile *catch* în funcție de valoarea aruncată de instrucțiunea *throw*.

Temă

Modificați funcțiile legate de tipul *Multime* din capitolul anterior așa încât să fie tratate și eventualele erori ce pot apărea în utilizarea lor.

PARTEA A DOUA

Obiective

În partea a doua ne propunem să studiem programarea orientată pe obiecte dintr-o perspectivă pur teoretică, după care vom vedea cum se poate efectiv programa orientat pe obiecte în C++. Vom studia modul în care se scrie o clasă, o metodă, un constructor, un destructor, cum se instanțiază o clasă, cum se moștenește etc.

2.1. Prezentare teoretică a P.O.O.

La baza programării orientate pe obiecte (POO) stă conceptul de **clasă**. Clasa este o colecție de câmpuri (date) și metode (funcții) care în general utilizează și prelucrează datele clasei. *Metodele* se mai numesc și *funcții membre* clasei. În C++, o clasă se redactează cu ajutorul tipului **class**, care poate fi considerată o îmbunătățire a tipului **struct** din C.

O clasă este un tip abstract de date, la facilitățile căreia avem acces prin intermediul unui **obiect** definit ca o instanță a acelei clase. Obiectul este de fapt o variabilă de tip clasă. Un obiect poate fi comparat cu o “cutie neagră” în care introducem și extragem informații despre care ne asigurăm când proiectăm și redactăm clasa că se prelucrează corect. Obiectele sunt niște “piese” ce le asamblăm pe o “placă de bază” (programul principal) pentru a obține aplicația dorită.

Programarea orientată pe obiecte este mai mult decât o tehnică de programare, ea este un mod de gândire. După cum am mai spus, într-o clasă găsim câmpuri și metode. A programa orientat pe obiecte nu înseamnă însă a scrie o mulțime de date și funcții grupate într-o clasă. Este mult mai mult decât atât. Când se scrie un program trebuie simțită o împărțire firească în module de sine stătătoare a codului așa încât ele să conducă apoi la o îmbinare naturală și facilă.

Programarea orientată pe obiecte (în engleza **object oriented programming - OOP**), pe scurt **POO**, crește modularitatea programului, iar depanarea și modificările programelor ale caror cod este scris orientat pe obiecte se realizează mult mai ușor. De asemenea, codul redactat orientat pe obiecte poate fi oricând refolosit atunci când se scriu programe noi în care apar idei asemănătoare.

POO devine indispensabilă atunci când se scriu programe de dimensiuni cel puțin medii. În cazul acestor programe este necesar aportul mai multor programatori, care pot fi specializați pe diferite domenii. Problema se împarte în probleme mai mici, care sunt repartizate la programatori diferiți. Redactarea obiect orientată permite îmbinarea mult mai ușoară a secvențelor de program, fără a fi nevoie de conversii sau adaptări. Scriind codul orientat pe obiecte creăm o “trusă” de unelte care crește în timp, unelte pe care le putem refolosi ulterior.

În continuare vom prezenta restul conceptelor care stau la baza programării orientate pe obiecte: abstractizarea datelor, moștenirea, polimorfismul, încapsularea datelor.

Abstractizarea datelor, în engleză - **data abstraction**, reprezintă procesul de definire a unui tip de date denumit **tip abstract de date** (în engleză – **abstract data type**, sau pe scurt **ADT**), recurgând și la ascunderea datelor.

Definirea unui tip abstract de date implică specificarea reprezentării interne a datelor pentru acel tip, precum și un set suficient de funcții cu ajutorul cărora putem utiliza acel tip de date fără a fi necesară cunoașterea structurii sale interne. Ascunderea datelor asigură modificarea valorilor acestor date fără a altera buna funcționare a programelor care apelează funcțiile scrise pentru tipul abstract de date.

Abstractizarea datelor nu este un concept legat neapărat de POO. Limbajul C oferă câteva exemple de tipuri abstracte de date. De exemplu tipul FILE este o structură complexă de date scrisă pentru lucrul cu fișiere în C, pentru care nu avem nevoie să cunoaștem câmpurile sale,

atâta timp cât avem definite suficiente funcții care lucrează cu acest tip de date: `fopen`, `fclose`, `fwrite`, `fread`, `fprintf`, `fscanf`, `fgets`, `fputs`, `fgetc`, `fputc`, `feof`, `fseek`, `ftell` etc. Toate aceste funcții realizează o interfață de lucru cu fișiere, la baza căreia stă însă tipul `FILE`.

Clasele sunt de departe însă cele mai bune exemple de tipuri abstracte de date.

Datorită faptului că o clasă este un tip de date deosebit de complex, crearea unui obiect (alocarea memoriei, inițializarea datelor etc.) se face prin intermediul unei funcții membre speciale numite **constructor**.

În majoritatea limbajelor eliberarea memoriei alocate pentru un obiect se face prin intermediul unei alte funcții membre speciale denumite **destructor**.

Când vorbim despre programarea orientată pe obiecte, prin **încapsularea datelor** înțelegem faptul că accesul la date se poate face doar prin intermediul unui obiect. Mai mult, datele declarate ca fiind private (cele care țin de bucătăria internă a unei clase) nu pot fi accesate decât în momentul descrierii clasei.

Moștenirea (în engleză - **inheritance**) este un alt concept care stă la baza reutilizării codului orientat pe obiecte. O clasă poate moșteni caracteristicile uneia sau mai multor clase. Noua clasă poate extinde sau/și particulariza facilitățile moștenite. Clasa moștenită se numește **clasa de bază**, în engleză - **base class**, iar cea care moștenește se numește **clasa derivată**, în engleză - **derived class**. O colecție de clase realizate pe principiul moștenirii se numește **ierarhie de clase**. Într-o ierarhie clasele sunt organizate arborescent, rădăcina arborelui este în general o clasă abstractă (ce nu poate fi instanțiată) care trasează specificul ierarhiei.

După cum am spus, moștenirea este folosită în două scopuri: pentru a particulariza o clasă (de exemplu pătratul moștenește dreptunghiul) sau/și pentru a îmbogăți o clasă prin adăugarea de facilități noi clasei derivate (de exemplu unei clase ce lucrează cu un polinom îi putem adăuga o nouă metodă cum ar fi cea pentru calculul valorii într-un punct). În cele mai multe situații însă clasa derivată particularizează o clasă deja existentă și o îmbogățește în același timp (de exemplu, la clasa dedicată pătratului putem adăuga o metodă pentru calcularea razei cercului înscris, metodă ce nu putea exista în clasa scrisă pentru dreptunghi).

Limbajul C++ este unul dintre puținele limbaje care acceptă **moștenirea multiplă**, adică o clasă poate fi derivată din mai multe clase.

În sens general, prin **polimorfism** înțelegem proprietatea de a avea mai multe forme. Când vorbim însă despre POO, polimorfismul constă în faptul că o metodă cu același nume și aceleași parametri poate fi implementată diferit pe nivele diferite în cadrul aceleiași ierarhii de clase.

În continuare prezentăm o posibilă rețetă de scriere a unei clase noi:

1. Căutăm dacă există ceva facilități într-o clasă sau în mai multe clase deja existente pe care să le putem adapta clasei noastre. Dacă există, atunci tot ceea ce avem de făcut este să moștenim aceste clase.
2. Trebuie să alegem bine încă de la început datele membre clasei. De obicei majoritatea datelor membre clasei (dacă nu chiar toate) se declară private sau protected, în cazul în care clasa este posibil să fie reutilizată în alt scop.
3. Scriem suficienți constructori pentru clasă. Aceștia vor crea obiecte sub diverse forme, inițializând diferit câmpurile clasei.
4. Trebuie scrise suficiente funcții (metode) pentru prelucrarea câmpurilor. Nu trebuie să fim zgârciți în a scrie metode, chiar dacă par a nu fi de folos într-o primă fază. Nu se știe niciodată când va fi nevoie de ele în proiectul nostru sau dacă vom refolosi clasa altă dată. Scriem metode de tipul *set...* și *get...*. Ele modifică valorile câmpurilor și, respectiv, le interoghează. Acest lucru este necesar deoarece, după cum am văzut, în general câmpurile sunt ascunse (private). Prin intermediul lor limităm accesul și ne asigurăm de atribuirea corectă de valori câmpurilor.

5. O parte dintre metode, cele care țin de bucătăria internă a clasei (sunt folosite numai în interiorul clasei), le vom ascunde, adică le vom declara private sau protected.
6. Scriem metode așa încât obiectul clasei să poată interacționa cu alte obiecte. Pregătim astfel posibilitatea de a lega modulul nostru la un proiect (un program).

Urmărind rețeta de mai sus să vedem cum proiectăm o clasă care lucrează cu un număr rațional, în care numărătorul și numitorul sunt memorați separat, în două câmpuri întregi:

1. Clasa nu va fi derivată din alta clasă.
2. Câmpurile clasei sunt numărătorul și numitorul, ambele private și de tip întreg.
3. Este util a scrie cel puțin un constructor cu întrebuițări multiple. El inițializează numărătorul și numitorul cu două valori întregi primite ambele ca parametri. Putem fixa valorile implicite 0 și 1 pentru numărător și respectiv numitor.
4. Scriem funcții care returnează numărătorul și numitorul și eventual funcții pentru modificarea numărătorului și a numitorului.
5. Scriem o funcție ascunsă (privată sau protejată) care simplifică fracția așa încât să se obțină una ireductibilă. Vom folosi această funcție de fiecare dată când va fi nevoie, așa încât în permanență fracția să fie memorată în forma ireductibilă. Metoda va fi folosită practic numai atunci când apar modificări ale numărătorului sau/și a numitorului.
6. Scriem metode pentru adunarea, scăderea, înmulțirea, împărțirea numărului nostru cu un alt obiect rațional și cu un număr întreg etc.

După ce vom avea suficiente cunoștințe vom implementa această clasă ca un prim exemplu.

2.2. Programarea orientată pe obiecte în C++

Obiective

Ne propunem să vedem cum se scrie cod orientat pe obiecte în C++. Mai întâi o să vedem care sunt neajunsurile POO din C folosind tipul *struct* și vom vedea efectiv cum se declară și descrie o clasă în C++, o metodă, un constructor, un destructor, ce este și cum se scrie o funcție prietenă unei clase etc. Ne vom concentra apoi asupra moștenirii din C++, care este una dintre cele mai pretențioase, în special datorită faptului că în C++ există moștenire multiplă.

2.2.1. Neajunsuri ale POO în C

Cu ajutorul tipului *struct* din C putem scrie cod orientat pe obiecte, însă, după cum vom vedea, cod adevărat orientat pe obiecte putem scrie numai în C++ cu ajutorul tipului *class*. Să prezentăm însă pe scurt neajunsurile programării orientate pe obiecte din C, folosind tipul *struct*.

În interiorul tipului *struct*, pe lângă câmpuri putem avea și funcții.

Membrii unei structuri (fie ei date sau funcții) pot fi accesați direct, adică toți se comportă ca fiind publici, declararea unor date sau funcții interne private sau protected fiind imposibilă.

Trebuie avut grijă asupra corectitudinii proiectării unui obiect în C (reținut într-o variabilă de tip *struct*), astfel încât acesta să suporte moștenirea. Este necesară scrierea unor funcții suplimentare pentru a permite unui obiect să-și acceseze corect datele.

Moștenirea comportării obiectului ca răspuns la diferite mesaje necesită funcții suport pentru a se realiza corect.

Toate aceste probleme nu există dacă utilizăm tipul *class* din C++, care răspunde în totalitate cerințelor ridicate de POO.

2.2.2. Declarația unei clase în C++

Declarația unei clase în C++ este:

```
class nume_cls [:[virtual] [public/protected/private] clasa_baza1,
               [virtual] [public/protected/private] clasa_baza2,...]
{
    public:
        // declaratii de date, functii membre publice
    protected:
        // declaratii de date, functii membre protejate
    private:
        // declaratii de date, functii membre private
}[obiect, *obiect2 ... ];
```

Menționăm că tot ceea ce apare între paranteze pătrate este interpretat a fi opțional.

Ca și în cazul tipurilor *struct*, *union* și *enum* din C, descrierea unei clase se încheie cu ; (punct și virgulă). Între acolada închisă și caracterul ; (punct și virgulă) pot fi definite obiecte sau/și pointeri către tipul *class*.

După numele clasei urmează opțional : (două puncte) și numele claselor moștenite despărțite prin virgulă. Vom reveni cu detalii când vom discuta despre moștenire.

Declarațiile datelor și metodele pot fi precedate de unul dintre specificatorii de acces (care sunt cuvinte cheie, rezervate) **public**, **protected** sau **private**. În lipsa unui specificator de acces, datele și metodele sunt considerate a fi implicit private !

Membrii de tip **public** pot fi accesați în interiorul clasei, în interiorul eventualelor clase derivate și prin intermediul unui *obiect* al clasei respective sau al unei clase derivate sub forma *obiect.membru_public*.

Membrii declarați **protected** ai unei clase pot fi accesați în interiorul clasei sau în interiorul unei eventuale clase derivate, dar nu pot fi accesați sub forma *obiect.membru_protected*, unde *obiect* este o instanță a clasei respective sau a unei clase derivate din aceasta.

Datele și metodele de tip **private** pot fi accesate numai din interiorul clasei la care sunt membre.

În general, datele unei clase se declară *private* sau *protected*, iar majoritatea metodelor se declară *publice*. Accesul la datele clasei se va face prin intermediul unor funcții membre special definite în acest scop. Aceste funcții sunt de tipul *set...* (pentru setarea valorilor câmpurilor), respectiv *get...* (care returnează valorile câmpurilor obiectului).

Ordinea definirii membrilor în funcție de modul lor de acces este opțională. Pot apărea mai multe declarații sau nici una de tip *public*, *protected* sau *private* pe parcursul descrierii clasei.

2.2.3. Declarația și descrierea funcțiilor membre

Funcțiile membre unei clase pot fi descrise *inline* sau nu.

Funcțiile care nu conțin instrucțiuni repetitive pot fi descrise în locul în care se definește clasa, în această situație ele fiind considerate a fi *inline* (vezi capitolul dedicat funcțiilor *inline*):

```
class nume_clasa
```

```

{
//....
tip_returnat functie(parametri) // fct. membra inline
{
    // descriere fct. (fara instructiuni repetitive)
}
//....
};

```

Descrierea funcției membre de mai sus este echivalentă cu:

```

class nume_clasa
{
//....
tip_returnat functie(parametri);
//....
};

inline tip_returnat nume_clasa::functie(parametri)
{
    // corpul funcției
}

```

Orice funcție membră poate fi descrisă separat de locul în care este definită clasa, mai mult, funcțiile care conțin instrucțiuni repetitive trebuie neapărat descrise în acest mod:

```

class nume_clasa
{
    //....
    tip_returnat functie(parametri);
    //....
};

tip nume_clasa :: functie(parametri) // descriere metoda
{
    // care nu este inline
    // corpul funcției
}

```

Denumirile parametrilor funcției în momentul descrierii clasei trebuie să coincidă cu denumirile parametrilor din momentul în care descriem funcția. Putem da însă numai tipul parametrilor în momentul definirii funcției în interiorul descrierii clasei și ulterior să dăm efectiv denumire acestor parametri când descriem funcția:

```

class Test
{
    void fct(Test,int);
    void fct2(Test t, int k);
};

void Test::fct(Test t,int n) // aici capatata nume param.
{

```

```

        // corpul functiei
    }

void Test::fct2(Test t,int k) // aceleasi denumiri pt. param.
{
    // corpul functiei
}

```

Dacă o funcție care conține instrucțiuni repetitive este descrisă în momentul definirii clasei, atunci la compilare suntem atenționați că această funcție nu va fi considerată a fi *inline*.

2.2.4. Constructori

În C++ **constructorii** se descriu ca și funcțiile membre obișnuite, dar ei poartă numele clasei și nu au tip returnat, nici măcar *void*:

```

class nume_clasa
{
    //....
    nume_clasa(parametri); // declaratie constructor
    //....
};

```

Pot exista mai mulți constructori definiți pentru o clasă, dar cu parametri diferiți. În lipsa scrierii de constructori de către programator, la compilare se generează **constructori implicați**.

Un obiect se crează static folosind unul dintre constructorii clasei astfel:

```

nume_clasa ob(valori_parametri); // constructie obiect static

```

Dacă folosim constructorul fără parametri al clasei, atunci un obiect se crează static astfel:

```

nume_clasa ob; // creare obiect static folosind constructorul fara parametri

```

Un membru al obiectului creat static se accesează în felul următor:

```

ob.membru

```

Pentru a construi un obiect dinamic definim un pointer către clasă și folosim operatorul *new*:

```

nume_clasa *pob=new nume_clasa(valori_parametri); // constructie ob. dinamic

```

Crearea unui obiect dinamic folosind constructorul fără parametri se realizează astfel:

```

nume_clasa *pob=new nume_clasa; // constructie obiect dinamic

```

Un membru al obiectului creat dinamic se accesează în felul următor:

```

ob->membru

```

Există o categorie specială de constructori denumiți constructori de copiere, despre care vom vorbi mai târziu.

2.2.5. Destructori

Destructorul în C++ poartă numele clasei precedat de semnul ~ (tilda), **nu are parametri și nici tip returnat**:

```
class nume_clasa
{
    //....
    ~nume_clasa(); // definire destructor
    //....
};
```

Orice clasă are un singur destructor.

Dacă nu scriem destructor, atunci la compilare se generează unul implicit.

Destructorul clasei se apelează direct pentru a se distruge obiectele create static (la părăsirea unei funcții se distruge obiectele statice primite ca parametri transmiși prin valoare, de asemenea se distruge obiectele create static în corpul funcției).

Este evident că pentru a putea distruge un obiect când nu mai avem nevoie de el pe parcursul execuției programului, el trebuie creat dinamic. Distrugerea unui obiect creat anterior dinamic se face astfel:

```
delete pob;    // se apeleaza destructorul clasei pentru
               // a distruge obiectul de la adresa pob
```

Prezentăm în continuare o clasă pentru dreptunghiuri cu laturile paralele cu axele de coordonate. Clasa va avea două metode: una pentru calcularea ariei și cealaltă pentru perimetru.

```
# include <math.h>           // fisierul antet ce contine definitia
funcției fabs
# include <iostream.h>

class dreptunghi
{
private:
    float x1,y1,x2,y2; // datele interne clasei ce definesc dreptunghiul.
public:
    // (coordonatele a doua varfuri diagonal opuse)
    dreptunghi(float X1,float Y1,float X2,float Y2) // constructor
    {
        x1=X1; // se initializeaza datele interne clasei: x1, y1, x2 si y2
        x2=X2; // cu valorile primite ca parametri: X1, Y1, X2 si Y2
        y1=Y1;
        y2=Y2;
    }
    float get_x1()
    {
        return x1;
    }
}
```



```

float arie() // metoda descrisa inline
{
    return fabs((x2-x1)*(y2-y1));
}
float perimetru(); // functie membra care va fi descrisa ulterior
};

float dreptunghi::perimetru() // descrierea functiei perimetru,
{
    // membra clasei dreptunghi
    return 2*(fabs(x2-x1)+fabs(y2-y1));
}

```

Funcțiile membre *arie* și *perimetru* se aplică obiectului curent, cel din care se apelează aceste funcții. De aceea nu este nevoie să transmitem ca parametru obiectul *dreptunghi* pentru care vrem să calculăm aria, respectiv perimetrul, așa cum se întâmplă în cazul funcțiilor externe clasei.

Pentru testarea clasei *dreptunghi* propunem următoarea funcție principală:

```

void main(void)
{
    dreptunghi d(20,10,70,50); // se creaza obiectul d folosind constructorul
                                //primeste ca param. X1=20, Y1=10, X2=70, Y2=50
    cout<<"Aria dreptunghiului: " << d.arie() << endl;
    cout<<"Perimetrul dreptunghiului: " << d.perimetru() << endl;
    cout<<"Dreptunghiul are ordonata: " << d.get_x1() << endl;
    // cout<<" Dreptunghiul are ordonata: " << d.x1;
}

```

Dacă se încearcă accesarea uneia dintre datele membre clasei: *x1*, *y1*, *x2* sau *y2*, atunci se obține eroare la compilare, deoarece ele sunt declarate *private*. De aceea afișarea valorii reținute în *x1* din obiectul *d* este greșită (linia comentată). Pentru a putea fi accesată valoarea *x1* direct sub forma *d.x1*, ea ar fi trebuit să fie declarată *public* în interiorul clasei. Datele membre unei clase se declară în general cu unul dintre specificatorii *private* sau *protected*. Dacă dorim să se poată obține valoarea unui câmp privat sau protejat scriem o funcție publică de tip *get...*, așa cum este cazul funcției *get_x1* din clasa *dreptunghi*. Pentru modificarea valorii reținute într-un câmp putem scrie o funcție publică a cărui nume de regulă începe cu *set...*. Pentru câmpul *x1*, funcția membră *set...* (definită în interiorul clasei *dreptunghi*) este:

```

public:
    void set_x1(float X1)
    {
        x1=X1; // campul privat
    }

```

Evident, funcția *set_x1* poate fi apelată în funcția *main* de mai sus astfel:

```

d.set_x1(10);

```

2.2.6. Funcții prietene (*friend*) unei clase

Funcțiile prietene unei clase se declară în interiorul clasei cu ajutorul specificatorului **friend**, care este un cuvânt rezervat și precede definiția funcției. Cu toate că sunt definite în interiorul clasei, **funcțiile prietene sunt externe clasei**. Spre deosebire de funcțiile externe obișnuite, funcțiile declarate *friend* **pot accesa însă datele *private* și *protected*** ale clasei în care sunt definite.

Schema de declarare a unei funcții prietene este:

```
class nume_clasa
{
    //....
    friend tip_retumat functie_prietena(parametri)
    {                                     // functie prietena inline
        // descriere functie
    }
    //....
};
```

Definirea funcției prietenă clasei de mai sus este echivalentă cu:

```
class nume_clasa
{
    //....
    friend tip_retumat functie_prietena(parametri);
    //....
};

inline tip_retumat functie_prietena(parametri)
{                                     // descrere functie prietena inline
    // corpul functiei
}
```

Ca și în cazul funcțiilor membre, funcțiile *friend* care conțin instrucțiuni repetitive trebuie descrise în exteriorul descrierii clasei (adică nu *inline*):

```
class nume_clasa
{
    //....
    friend tip_retumat functie_prietena(parametri);
    //....
};

tip_retumat functie_prietena(parametri)
{
    // descriere functie
}
```

Rescriem în continuare clasa *dreptunghi* de mai sus cu funcții prietene în loc de funcții membre:

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

```

class dreptunghi
{
private:
    float x1,y1,x2,y2;
public:
    dreptunghi(float X1,float Y1,float X2,float Y2)
    {
        x1=X1; x2=X2; y1=Y1; y2=Y2;
    }
    float get_x1(dreptunghi d)
    {
        return d.x1;
    }
    float set_x1(dreptunghi,float);
    friend float arie(dreptunghi d) // descrierea functiei prietena clasei
                                    // dreptunghi. Se primeste ca argument
                                    // un obiect de tipul clasei dreptunghi
    {
        return fabs((d.x2-d.x1)*(d.y2-d.y1));
    }
    friend float perimetru(dreptunghi); // functie prietena ce va fi
                                        // descrisa ulterior
};

float set_x1(dreptunghi d,float X1)
{
    d.x1=X1;
}

float perimetru(dreptunghi d)
{
    return 2*(fabs(d.x2-d.x1)+fabs(d.y2-d.y1));
}

void main(void)
{
    dreptunghi d(20,10,70,50);
    cout<<"Aria dreptunghiului: " << arie(d) << endl;
    cout<<"Perimetrul dreptunghiului: " << perimetru(d) << endl;
    cout<<"Dreptunghiul are ordonata: " << get_x1(d) << endl;
}

```

Într-o funcție prietenă, din cauză că este externă clasei, de obicei transmitem ca parametru și obiectul asupra căruia se referă apelul funcției. Astfel, de exemplu în cazul funcției *set_x1* trimitem ca parametru obiectul *d* pentru care dorim să setăm valoarea câmpului *x1*. Așadar, în general o funcție prietenă unei clase are un parametru suplimentar față de situația în care aceeași funcție ar fi fost scrisă ca fiind membră clasei. Aceași regulă se aplică și supraîncării unui operator ca fiind prieten clasei.

Din cauză că sunt funcții prietene clasei, datele interne private ale obiectului *d* de tip *dreptunghi* au putut fi accesate sub forma: *d.x1*, *d.y1*, *d.x2* și *d.y2*. Evident, dacă funcțiile nu erau prietene clasei, ci erau funcții externe obișnuite (definițiile lor nu apăreau declarate *friend* în interiorul definiției clasei), accesul la datele membre obiectului *d* nu era posibil.

2.2.7. Declarația, descrierea operatorilor pt. o clasă

Operatorii pot fi supraincărcați ca membrii unei clase sau ca fiind prieteni unei clase. Regulile legate de modul de definire și descriere ale unui operator sunt aceleași ca și pentru orice funcție membră, respectiv prietenă unei clase. Un operator nu poate fi însă definit sub ambele forme (interior și exterior clasei) pentru aceleași tipuri de operanzi. Operatorii pot fi definiți de mai multe ori în interiorul aceleiași clase, dar cu tipuri pentru parametri diferite.

Dacă un operator de aritate n este definit ca fiind membru unei clase, atunci el va avea $n-1$ parametri. Așadar, un operator unar nu va avea nici un parametru și se va aplica obiectului care îl apelează. Pentru un operator binar avem un singur parametru. Operatorul binar se aplică între primul obiect, considerat a fi obiectul curent, din care se apelează operatorul, și valoarea primită ca argument (vezi operatorul $+$ din clasa *complex* prezentată mai jos).

Dacă un operator de aritate n este definit ca fiind prieten unei clase, atunci el va avea n parametri.

În general operatorii $<<$, $>>$ vor fi definiți ca fiind prieteni clasei pentru că primul argument este de obicei un flux în care se trimit, respectiv din care se extrag valorile câmpurilor obiectului.

În continuare prezentăm o clasă care lucrează cu un număr complex:

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

class complex
{
private:
    float re,im;
public:
    complex(float x=0,float y=0) // constructor cu parametri cu valori implicite
    {
        re=x;
        im=y;
    }
    complex operator+(complex z2) // operator membru clasei pentru adunare
    {
        complex z;
        z.re=re+z2.re;
        z.im=im+z2.im;
        return z;
    }
    complex operator+(float r) // adunare cu un numar real
    {
        complex z;
        z.re=re+r;
        z.im=im;
        return z;
    }
    friend complex operator+(float r,complex z2) // real+complex
    {
        complex z;
        z.re=r+z2.re;
```

```

        z.im=z2.im;
        return z;
    }
    friend complex operator-(complex z1,complex z2) // operator prieten
    {
        complex z;
        z.re=z1.re-z2.re;
        z.im=z1.im-z2.im;
        return z;
    }
    friend istream& operator>>(istream &fl,complex &z);
    friend istream& operator<<(istream &fl,complex &z);
};

istream& operator>>(istream &fl,complex &z)
{
    fl>>z.re>>z.im;
    return fl;
}

ostream& operator<<(ostream &fl,complex &z)
{
    if (z.im>=0) fl<<z.re<<"+"<<z.im<<"i";
    else fl<<z.re<<z.im<<"i";
    return fl;
}

```

Pentru testarea clasei *complex* propunem următoarea funcție principală:

```

void main(void)
{
    complex z,z1(1),z2(2,5);

    cout<<"Suma este:      "<<(z1+z2)<<endl;
    cout<<"Diferenta este: "<<(z1-z2)<<endl;
    cout<<"Complex+real:   "<<(z1+7)<<endl;
    cout<<"Real+complex:    "<<(8.2+z2)<<endl;
}

```

În exemplul de mai sus, constructorul are valori implicite pentru cei doi parametri. Astfel, pentru obiectul *z* din programul principal datele interne *re* și *im* se vor inițializa cu valorile implicite, adică ambele vor fi 0. Obiectul *z1* din programul principal va avea *re* inițializat cu 1 și *im* cu valoarea implicită 0. În fine, obiectul *z2* va avea *re* inițializat cu 2, iar *im* cu 5 (valorile transmise ca parametri în constructor).

Pentru a supraîncărca operatorii de atribuire corect, și nu numai pentru aceștia, folosim pointerul *this*.

2.2.8. Membri statici

În C (și în C++), o variabilă locală într-o funcție poate fi declarată ca fiind statică. Pentru această variabilă se păstrează valoarea și, când se reapelează funcția, variabila va fi inițializată cu

valoarea pe care a avut-o la apelul anterior. De fapt pentru variabila locală statică este alocată o zonă de memorie de la începutul până la sfârșitul execuției programului, dar accesul la variabilă nu este posibil decât din interiorul funcției în care este declarată. Iată un exemplu simplu:

```
#include<stdio.h>

void fct(void)
{
    static int x=1;
    x++;
    printf("%d ",x);
}

void main(void)
{
    int i;
    for (i=0;i<10;i++) fct();
}
```

Variabila statică *x* este inițializată la începutul execuției programului cu valoarea 1. La fiecare apel al funcției *fct* variabila *x* se incrementează cu o unitate și se afișează noua sa valoare. Așadar pe ecran în urma execuției programului va apărea:

```
2 3 4 5 6 7 8 9 10 11
```

În C++ un membru al unei clase poate fi declarat ca fiind static. Un membru static este folosit în comun de toate obiectele clasei. Pentru un câmp static se alocă o zonă (unică) de memorie încă de la începutul execuției programului. Câmpul static va putea fi interogată și modificat de toate obiectele clasei, modificări ce vor fi vizibile în toate obiectele clasei. Mai mult, un membru static poate fi accesat direct (dacă este vizibil în acel loc) sub forma:

```
NumeClasa::MembruStatic;
```

În continuare vom da un exemplu în care într-un câmp static vom memora numărul de instanțe create pentru o clasă:

```
#include<iostream.h>

class test
{
private:
    static int n;
public:
    test()
    {
        n++;
    }
    static int NrInstante()
    {
        return n;
    }
}
```

```
};

int test::n=0; // definire si initializare camp static

void main()
{
    test t1,t2;
    cout<<test::NrInstante(); // apel metoda statica
}
```

Evident, în urma execuției programului, pe ecran se va afișa numărul 2.

Orice câmp static trebuie definit în exteriorul descrierii clasei. În momentul definirii se folosește operatorul de rezoluție pentru indicarea clasei la care este membru câmpul static (vezi câmpul *x* din exemplul de mai sus).

2.2.9. Pointerul *this*

Pointer-ul *this* (“acesta” în engleză) a fost introdus în C++ pentru a indica adresa la care se află memorat obiectul curent. El este folosit numai în corpul funcțiilor membre, inclusiv în constructori și destructor, pentru a ne putea referi la obiectul din care s-a apelat funcția, respectiv la obiectul care se construiește sau se distruge.

Compilatorul C++ modifică fiecare funcție membră dintr-o clasă astfel:

- 1) Transmite un argument în plus cu numele *this*, care este de fapt un pointer către obiectul din care s-a apelat funcția membră (*this* indică adresa obiectului care a apelat metoda).
- 2) Este adăugat prefixul *this->* tuturor variabilelor și funcțiilor membre apelate din obiectul curent (dacă nu au deja dat de programator).

Apelul unui membru (dată sau funcție) al unei clase din interiorul unei funcții membre se poate face sub forma *this->membru*, scriere care este echivalentă cu *membru* (fără explicitarea *this->*) dacă nu există cumva un parametru al funcției cu numele *membru*.

O instrucțiune de forma *return *this* returnează o copie a obiectul curent (aflat la adresa *this*).

Să vedem în continuare cum se definește corect operatorul = (de atribuire) pentru două numere complexe. După cum se știe, expresia *a=b* are valoarea ce s-a atribuit variabilei *a*. Așadar, o atribuire de numere complexe de forma *z1=z2* va trebui să aibă valoarea atribuită lui *z1*, adică operatorul = definit în clasa *complex* va trebui să returneze valoarea obiectului curent:

```
complex operator=(complex &z) // obiect_curent = z
{
    a=z.a;
    b=z.b;
    return *this;
}
```

Evident, supraîncărcarea operatorului = de mai sus va fi descrisă în interiorul clasei *complex*.

Pentru a înțelege mai bine ceea ce a fost prezentat până acum legat de programarea orientată pe obiecte din C++, dăm un exemplu mai amplu. Este vorba despre o clasă ce “știe” să

lucreze cu un număr rațional, considerat a fi memorat printr-o pereche de numere întregi de tip *long*, reprezentand numărătorul, respectiv numitorul numărului rațional.

```
# include <iostream.h>

class rational
{
private:
    long a,b;
    void simplificare();
public:
    rational(long A=0, long B=1)
    {
        a=A;
        b=B;
        simplificare();
    }
    long numarator()
    {
        return a;
    }
    long numitor()
    {
        return b;
    }
    rational operator+(rational x)
    {
        rational y;
        y.a=a*x.b+b*x.a;
        y.b=b*x.b;
        y.simplificare();
        return y;
    }
    rational operator-(rational x)
    {
        rational y;
        y.a=a*x.b-b*x.a;
        y.b=b*x.b;
        y.simplificare();
        return y;
    }
    rational operator*(rational x)
    {
        rational y;
        y.a=a*x.a;
        y.b=b*x.b;
        y.simplificare();
        return y;
    }
    rational operator/(rational x)
    {
        rational y;
```



```

        y.a=a*x.b;
        y.b=b*x.a;
        y.simplificare();
        return y;
    }
rational operator=(rational&x)
{
    a=x.a;
    b=x.b;
    return *this;
}
rational operator+=(rational x)
{
    return *this=*this+x;
}
rational operator-=(rational x)
{
    return *this=*this-x;
}
rational operator*=(rational x)
{
    return *this=*this*x;
}
rational operator/=(rational x)
{
    return *this=*this/x;
}
int operator==(rational x)
{
    return a==x.a && b==x.b;
}
int operator!=(rational x)
{
    return !(*this==x);
}
rational operator++() // preincrementare ++r
{
    a+=b;
    return *this;
}
rational operator--() // predecrementare --r
{
    a-=b;
    return *this;
}
rational operator++(int) // postincrementare r++
{
    rational c=*this;
    a+=b;
    return c;
}
rational operator--(int) // postdecrementare r--

```

```

        {
            rational c=*this;
            a-=b;
            return c;
        }
    long operator! ()
    {
        return !a;
    }
    friend ostream& operator<<(ostream& fl,rational x)
    {
        fl<<"("<<x.a<<" "<<x.b<<" )";
        return fl;
    }
    friend istream& operator>>(istream& fl,rational &x)
    {
        fl>>x.a>>x.b;
        return fl;
    }
};

void rational::simplificare()
{
    long A=a,B=b,r;

    while (B) // algoritmul lui Euclid pentru c.m.m.d.c. (a,b)
    {
        r=A%B;
        A=B;
        B=r;
    }
    if (A) // simplificare prin A = c.m.m.d.c. (a,b)
    {
        a/=A;
        b/=A;
    }
}

void main(void)
{
    rational x(7,15),y(1,5),z;
    z=x+y;
    cout<<x<<"+"<<y<<"="<<z<<endl;
    cout<<x<<"-"<<y<<"="<<(x-y)<<endl;
    cout<<x<<"*"<<y<<"="<<(x*y)<<endl;
    cout<<x<<"/"<<y<<"="<<(x/y)<<endl;
}

```

Propunem cititorului:

1. Implementarea operatorilor >>, <<, *, /, =, +=, -=, *=, /=, !, == și != ca membri sau prieteni clasei complex.

2. Implementarea operatorilor <, <=, > și >= pentru compararea a două numere raționale.

2.2.10. Constructorul de copiere

Scrierea unui constructor de copiere este necesară numai într-o unei clasă în care există alocare dinamică a memoriei. Constructorul de copiere trebuie să asigure copierea corectă a instanțelor unei clase.

Constructorul de copiere poate fi folosit direct de programator pentru crearea unui obiect nou (ca orice constructor), dar el este apelat în general automat în timpul execuției programului atunci când se transmite un obiect ca parametru într-o funcție prin valoare și la returnarea unui obiect prin valoare dintr-o funcție.

În lipsa unui constructor de copiere definit de programator, compilatorul crează un **constructor de copiere implicit**, dar care nu va ști însă să facă alocare dinamică de memorie. Dacă în clasă avem un câmp de tip pointer, atunci, după copierea unui obiect, pentru ambele obiecte (cel vechi și cel nou construit) câmpurile de tip pointer vor indica aceeași zonă de memorie. Astfel, dacă modificăm ceva la această adresă prin intermediul câmpului pointer al unui obiect, modificarea va fi vizibilă și din celelalte obiect. Acest lucru nu este în general dorit. De aceea programatorul trebuie să scrie constructorul de copiere care va alocă o zonă nouă de memorie pe care o va reține în câmpul de tip pointer al obiectului creat și în această zonă de memorie va copia ce se află la adresa câmpului obiectului care este copiat.

Un constructor de copiere are următoarea structură:

```
nume_clasa (const nume_clasa &);
```

Constructorul de copiere primește ca argument o constantă referință către obiectul care urmează a fi copiat.

Dacă vrem ca o valoarea unui parametru să nu poată fi modificată în interiorul unei funcții, punem în fața parametrului cuvântul rezervat **const**. Cu alte cuvinte un astfel de parametru devine o constantă în corpul funcției.

Spre exemplificare prezentăm o clasă ce lucrează cu un string alocat dinamic.

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

class string
{
private:
    char *s; // sirul de caractere retinut in string
public:
    string(char *st="") // constructor
    {
        s=new char[strlen(st)+1];
        strcpy(s,st);
    }
    string(const string &str) // constructor de copiere
    {
        delete [] s;
        s=new char[strlen(str.s)+1];
        strcpy(s, str.s);
    }
};
```

```

    }
~string()                // destructor
{
    delete [] s;
}
string operator+(string str) // apelare constructor copiere
{
    char *st;
    st=new char[strlen(s)+strlen(str.s)+1];
    string str2(st);
    sprintf(str2.s,"%s%s",s,str.s);
    return str2; // apelare constructor de copiere
}
string operator=(const string &str) // atribuire
{
    delete [] s;
    s=new char[strlen(str.s)+1];
    strcpy(s, str.s);
    return *this; // se apeleaza constructorul de copiere
}
string operator+=(const string &str)
{
    *this=*this+str;
    return *this; // apelare constructor de copiere
}
int operator==(const string &str) // identice ?
{
    if (!strcmp(s,str.s)) return 1;
    return 0;
}
int operator<(string str) // apelare constructor de copiere
{
    if (strcmp(s,str.s)<0) return 1;
    return 0;
}
int operator<=(const string &str)
{
    if (strcmp(s,str.s)<=0) return 1;
    return 0;
}
int operator>(const string &str)
{
    if (strcmp(s,str.s)>0) return 1;
    return 0;
}
int operator>=(const string &str)
{
    if (strcmp(s,str.s)>=0) return 1;
    return 0;
}
void set(char *st) // setararea unui string
{

```

```

        delete [] s;
        s=new char[strlen(st)+1];
        strcpy(s,st);
    }
    void get(char *st) // extragere sir caractere din obiectul string
    {
        strcpy(st,s);
    }
    int operator!() // se returneaza lungimea string-ului
    {
        return strlen(s);
    }
    char operator[] (int i) // returneaza caracterul de pe pozitia i
    {
        return s[i];
    }
    friend ostream& operator<<(ostream &fl,const string &str)
    {
        fl<<str.s;
        return fl;
    }
    friend istream& operator>>(istream &fl,const string &str)
    {
        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 + și <). De asemenea, de fiecare dată când se returnează un obiect de tip *string* (vezi operatorii +, = și +=), este apelat constructorul de copiere definit în interiorul clasei, care spune modul în care se face efectiv copierea (cu alocările și eliberările de memorie aferente).

Pentru a vedea efectiv traseul de execuție 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 execuției programului până se ajunge la linia curentă (pe care se află cursorul) se face apăsând butonul F4. Pentru ca să fie posibilă urmărirea execuției programului, în meniul *Options*, la *Debugger*, trebuie bifat *On in Source Debugging*. În Visual C++ urmărirea execuției pas cu pas a programului se face apăsând butonul F11, iar lăsarea liberă a execuției programului până la linia curentă se face apăsând Ctrl+F10.

Lăsăm plăcerea cititorului de a completa alte și funcții în clasa *string* cum ar fi pentru căutarea unui string în alt string, înlocuirea unui șir de caractere într-un string cu un alt șir de caractere, extragerea unui subșir de caractere dintr-un string etc.

2.2.11. Moștenirea în C++

În C++ o clasă poate să 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 și metodele clasei de bază este restricționat mai mult sau mai puțin în clasa derivată (cel mai puțin prin *public* și cel mai mult prin *private*).

Specificarea modului de derivare este opțională. Implicit, se ia, ca și la definirea datelor și 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 și metodelor clasei de bază în clasa derivată:

Tip date și metode din clasa de bază	Specificator mod de derivare	Modul în care sunt văzute în clasa derivată datele și 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 și 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 moștenirii multiple o să revenim după ce introducem noțiunea de *virtual*.

2.2.12. Funcții virtuale

În clase diferite în cadrul unei ierarhii pot apărea funcții cu aceeași semnătură (același nume și aceiași parametri), în engleză *overriding*. Astfel, putem avea situația în care într-o clasă derivată există mai multe funcții cu aceeași semnătură (unele moștenite din clasele de pe nivele superioare ale ierarhiei și 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 funcție 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 bază 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 așa cum ne-am putea aștepta. Mai mult, dacă funcția există în clasa D și 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 bază *B*, obțineam eroare la compilare pentru că pointerul *p* reține adresa subobiectului și este evident că în această situație la adresa indicată de *p* nu există nimic referitor la clasa *D*. Din aceleași considerente, dacă încercăm referirea la campul *x* sub forma *p->x*, vom obține de asemenea eroare la compilare.

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

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

Să precizăm și faptul că numai funcțiile membre unei clase pot fi declarate ca fiind virtuale.

Declarația unei funcții din clasa de bază ca fiind virtuale se adresează situațiilor de tipul celei de mai sus (clasa *D* este derivată din *B* și *B *p=new D;*). Astfel, dacă în fața declarației metodei *functie* din clasa *B* punem cuvântul rezervat **virtual**, atunci în urma execuției programului de mai sus pe ecran se vor afișa mesajele:

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

Deci, declarația **virtual** a funcției din clasa de bază a ajutat la identificarea corectă a apelului funcției din clasa derivată.

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

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

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

Datorită faptului că apelul unei funcții virtuale este localizat cu ajutorul tabelului VMT, apelarea funcțiilor virtuale este lentă. De aceea preferăm să declarăm funcțiile ca fiind virtuale numai acolo unde este necesar.

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

2.2.13. Destructori virtuali

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

Să vedem pe un exemplu simplu ce se întâmplă când destructorul clasei de bază este și 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 legătură cu obiectul clasei derivate, distrugerea se va face numai la adresa p , adică se va apela numai destructorul clasei B . Pentru a se distruge și obiectul clasei derivate D , trebuie să declarăm destructorul clasei de bază ca fiind virtual. În această situație la execuția programului, pe ecran vor apărea următoarele patru mesaje:

```

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

```

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

2.2.14. Funcții pur virtuale

Într-o ierarhie întâlnim situații în care la un anumit nivel, o funcție nu este implementată. Cu această situație ne întâlnim mai ales în clasele abstracte care pregătesc ierarhia, trasează specificul ei. Funcțiile 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 posibilității declarării funcțiilor pur virtuale, în alte limbaje de programare, pentru metodele neimplementate se dau mesaje.

O funcție pur virtuală se declară ca una virtuală numai că la sfârșit punem `=0`. Compilatorul C++ nu permite instanțierea unei clase care conține metode pur virtuale. Dacă o clasă *D* derivată dintr-o clasă *B* ce conține funcții pur virtuale nu are implementată o funcție 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 instanțiată.

Dăm în continuare un exemplu în care clasa *patrat* este derivată din clasa *dreptunghi*, care la rândul ei este derivată din clasa abstractă *figura*. Pentru fiecare figură dorim să reținem denumirea ei și vrem să putem calcula aria și perimetrul. Dreptunghiul și 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 și 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 și este moștenit în clasele *dreptunghi* și *patrat*. Funcția *getnume* returnează numele figurii și este de asemenea moștenită în clasele derivate.
2. Funcția *arie* nu este definită și în clasa *patrat*. În consecință, apelul *d.arie()* se va răsfrânge asupra metodei din clasa de baza *dreptunghi*.
3. Clasa *figura* conține metode pur virtuale, deci este abstractă. Ea trasează specificul ierarhiei (clasele pentru figuri derivate din ea vor trebui să implementeze metodele *arie* și *perim*). Clasa *figura* nu poate fi instanțiată.

Este evident că în exemplul de mai sus puteam să nu scriem clasa *figura*, câmpul *nume* și funcția *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 și interogată indirect prin intermediul funcției *getnume*) și două metode (pentru *arie* și *perimetru*).

O să dăm în continuare o posibilă aplicație 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 și 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 crescător acest vector de figuri după *arie* și să afișăm 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;
}

```

2.2.15. Moștenire multiplă

Limbajul C++ suportă moștenirea multiplă, adică o clasă D poate fi derivată din mai multe clase de bază B_1, B_2, \dots, B_n ($n > 0$):

```
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 B_1 și B_2 sunt baze pentru o clasă D , iar B_1 și B_2 sunt derivate din aceeași clasă C ? În aceasta situație, așa cum vom vedea în subcapitolul următor, clasa C se declară în general ca fiind virtuală.

2.2.16. Clase virtuale

O clasă C se moștenește virtual dacă poate exista situația ca la un moment dat două clase B_1 și B_2 să fie baze pentru o aceeași clasă derivată D , iar B_1 și B_2 să fie descendente (nu neaparat direct) din aceeași 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* și *B2*, iar *B1* și *B2* sunt ambele derivate din clasa *C*. Astfel, facilitatile oferite de clasa *C* (funcția *fct*) ajung în *D* de două ori: prin intermediul clasei *B1* și prin intermediul clasei *B2*. De fapt, orice obiect al clasei *D* va avea în această situație două sub-obiecte ale clasei *C*. Din cauză că funcția *fct* este apelată în interiorul clasei *D* nu se poate decide asupra cărei dintre cele două funcții din cele două subobiecte se va răsfrânge apelul. Astfel, obținem eroare la compilare. Pentru a elimina această problemă, clasa *C* trebuie să fie declarată virtuală în momentul în care ambele clase, *B1* și *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();
    }
};

```

2.2.17. Constructori pentru clase virtuale

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

Dacă o clasă este derivată din mai multe clase de bază, atunci clasele de bază nevirtuale sunt declarate primele, așa 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 și în final se apelează constructorul clasei derivate D.

Pentru moștenirea multiplă considerăm situația în care clasa *patrat* este derivată din clasele *dreptunghi* și *romb*, iar clasele *dreptunghi* și *romb* sunt derivate ambele din clasa *patrulater*, care este derivată la randul ei din clasa abstractă *figura*. Dreptunghiul și 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 patrati: public dreptunghi, public romb
{
protected:
    float x, y, l;
public:
    patrati(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,"Patrati");
        x=X;
        y=Y;
        l=L;
    }
    virtual float arie() // calcul arie patrati ca fiind dreptunghi
    {
        return dreptunghi::arie();
    }
    virtual float perimetru() // calcul perimetru patrati 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);
    patrati 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 patratiului: " <<p.arie()<<endl;
    cout<<"Perimetrul patratiului: " <<p.perimetru()<<endl<<endl;
}

```

Observații:

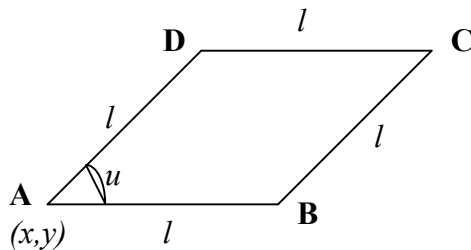
- 1) Patrulaterul este dat prin coordonatele celor 4 vârfuri ale sale (în sens trigonometric sau invers trigonometric): (x_1, y_1) , (x_2, y_2) , (x_3, y_3) și (x_4, y_4) .
- 2) Dreptunghiul (paralel cu axele de coordonate) este considerat ca fiind definit prin două vârfuri diagonal opuse având coordonatele: (x_1, y_1) și (x_2, y_2) .
- 3) Rombul (cu două laturi paralele cu abscisa) este caracterizat prin coordonatele vârfului stânga-sus (x, y) , lungimea laturii sale l și 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) și 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 $A_1 A_2 \dots A_n$ (în cazul nostru $n = 4$) este:

$$S_{A_1 A_2 \dots A_n} = \frac{\left| \sum_{i=1}^n x_i \cdot y_{i+1} - x_{i+1} \cdot y_i \right|}{2}, \text{ unde } A_i(x_i, y_i) \ (i=1 \dots n) \text{ si } A_{n+1} = A_1.$$

- 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:

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

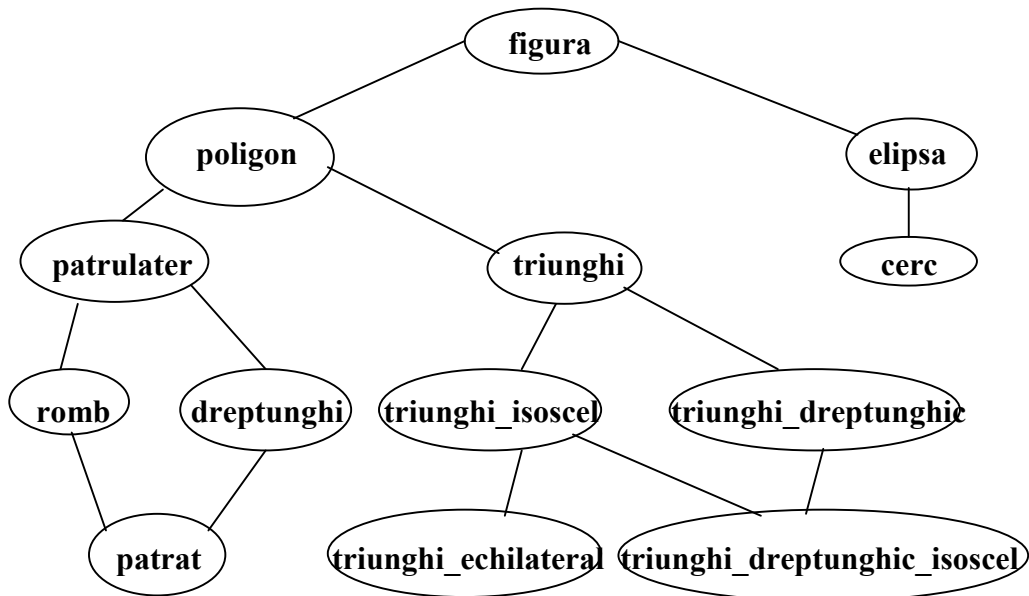
Pentru vârful C al rombului am realizat o translație a punctului D pe axa Ox cu l și am obținut coordonatele $(x+l \cdot \cos(u)+l, y+l \cdot \sin(u))$.

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 cunoștință cu moștenirea din C++ și problemele pe care le ridică moștenirea multiplă. Am văzut cum se rezolvă elegant aceste probleme cu ajutorul claselor virtuale.

Temă

Lăsăm ca exercițiu cititorului implementarea următoarei ierarhii de figuri:



2.2.18. Clase imbricate

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

Dăm un exemplu:

```
class X
{
    //....
    class Y          // clasa Y imbricata in X
    {
        //....
    };
    //....
};

class Z
{
    //....
    class Y          // clasa Y imbricata in Z
    {
```

```

        // ....
    };
// ....
};

```

La instanțierea claselor X sau Z (din exemplul de mai sus) nu se crează instanțe ale claselor Y imbricate în acestea. Instanțierea clasei imbricate trebuie făcută explicit:

```

X::Y y1;    // y1 este obiect al clasei Y imbricate în X
Z::Y y2;    // y2 este obiect al clasei Y imbricate în Z

```

Clasa imbricată nu are acces asupra datelor *private* sau *protected* ale clasei în care este imbricată. De asemenea, o clasă nu are acces asupra datelor *private* sau *protected* ale eventualelor clase imbricate.

2.2.19. Clase șablon (*template*)

Obiective

Ne propunem să reluăm discuția despre șabloane, introducând acum noțiunea de clasă *template*. O să vedem cum alături de funcțiile șablon, clasele șablon permit scrierea de cod generic în C++, cod care se funcționează pentru diverse tipuri de date.

Ca și în cazul unei funcții șablon, unul sau mai multe tipuri de date pot să nu fie explicitate în momentul definirii unei clase șablon. În momentul compilării, în locul unde se instanțiază clasa, se identifică aceste tipuri de date și se înlocuiesc cu tipurile identificate. O clasă șablon se declară astfel:

```

template <class T1, class T2, ... , class Tn>
class nume_clasa
{
    // descriere clasa nume_clasa
};

```

Instanțierea unei clase șablon se face astfel:

```

nume_clasa<tip1, tip2, ... , tipn> obiect;

```

La instanțiere, tipurile generice de date $T1, T2, \dots, Tn$ se înlocuiesc cu tipurile de date specificate între semnele mai mic $<$ și mai mare $>$, adică cu $tip1, tip2, \dots, tipn$.

Fiecare funcție membră unei clase șablon este la rândul ei funcție șablon. Astfel, descrierea fiecărei funcții membre în exteriorul definiției clasei (adică nu *inline*) se va face ca orice funcție șablon obișnuită.

În C++ și tipurile *struct* și *union* pot fi template.

Spre exemplificare, scriem o clasă șablon pentru o stivă:

```

#include<conio.h>
#include<stdlib.h>
#include<iostream.h>

template <class T>
struct NodListaSimpluInlantuita

```

```

{
    T info;
    NodListaSimpluInlantuita<T> *leg;
};

template <class T>
class Stiva
{
private:
    NodListaSimpluInlantuita<T> *cap;
public:
    Stiva()                // constructor
    {
        cap=NULL;
    }
    int operator!()         // verificare stiva goala
    {
        return cap==NULL;
    }
    void operator<<(T x) // introducere in stiva
    {
        NodListaSimpluInlantuita<T> *p=cap;
        cap=new NodListaSimpluInlantuita<T>;
        cap->info=x;
        cap->leg=p;
    }
    void operator>>(T &); // scoatere din stiva
    friend ostream& operator<<(ostream&,Stiva<T>&); // tiparire continut stiva
    void golire();        // golire stiva
    ~Stiva()              // destructor
    {
        golire();
    }
};

template <class T>
void Stiva<T>::operator>>(T &x)
{
    if (cap==NULL) throw "Stiva goala!";
    x=cap->info;
    NodListaSimpluInlantuita<T> *p=cap->leg;
    delete cap;
    cap=p;
}

template <class T>
ostream& operator<<(ostream& fl,Stiva<T>& st)
{
    NodListaSimpluInlantuita<T> *p=st.cap;
    while (p!=NULL)
    {

```

```

        fl<<p->info;
        p=p->leg;
    }
    return fl;
}

template <class T>
void Stiva<T>::golire()
{
    NodListaSimpluInlantuita<T> *p=cap;
    while (cap!=NULL)
    {
        p=cap->leg;
        delete cap;
        cap=p;
    }
}

```

Utilizăm clasa șablon pentru lucra cu o stivă de caractere:

```

void main()
{
    char x;
    Stiva<char> st;

    do
    {
        cout<<endl<<endl;
        cout<<" 1. Introducere element in stiva"<<endl;
        cout<<" 2. Scoatere element din stiva"<<endl;
        cout<<" 3. Afisare continut stiva"<<endl;
        cout<<" 4. Golire stiva"<<endl<<endl;
        cout<<"Esc - Parasire program"<<endl<<endl;
        switch (getch())
        {
            case '1':
                cout<<"Dati un caracter: "<<flush;
                st<<getche();
                break;
            case '2':
                try
                {
                    st>>x;
                    cout<<"Am scos din stiva: "<<x;
                }
                catch (char *mesajeroare)
                {
                    cerr<<"Eroare: "<<mesajeroare;
                }
                break;
            case '3':
                if (!st) cout<<"Stiva este goala!";

```



```

        else cout<<"Stiva contine: "<<st;
        break;
    case '4':
        st.golire();
        cout<<"Stiva a fost golita!";
        break;
    case 27: return;
    default:
        cerr<<"Trebuie sa apasati 1,2,3,4 sau Esc";
    }
    cout.flush();
    getch();
}
while (1);
}

```

După ce avem scrisă clasa șablon, nu trebuie decât să specificăm tipul de date pentru care vrem să o folosim. Astfel, clasa *Stiva* de mai sus o putem folosi pentru orice tip de date.

Rezumat

Am vazut cum se scrie o clasă șablon și cum se folosește pentru diverse tipuri de date. Am dat un exemplu ilustrativ: clasă șablon de lucru cu o stivă. Astfel, această clasă poate fi folosită pentru a lucra cu o stivă de caractere, o stivă de numere întregi, de numere reale etc.

Teme

În finalul prezentării din punct de vedere teoretic al programării orientate pe obiecte din C++, propunem să se scrie:

- 1) Un program care utilizează clasa șablon *Stiva* și pentru alt tip de date decât *char*.
- 2) O clasă șablon de lucru cu o coadă (similară clasei *Stiva*). Să se utilizeze această clasă pentru a rezolva următoarea problemă:

Se deschide un fișier text. Se citesc caracterele fișierului unul câte unul. Când se întâlnește o consoană, se introduce în coadă. Când se întâlnește o vocală, dacă nu este goală coada, se scoate o consoană din listă. Restul caracterelor (cele care nu sunt litere) se vor ignora. La sfârșit să se afișeze conținutul cozii.

- 3) Clasa șablon de lucru cu o mulțime ce reține elementele într-un vector alocat dinamic. Clasa va avea implementați operatorii: +, - și * pentru reuniune, diferență și respectiv intersecție, operatorii << și >> pentru introducerea unui element în mulțime și respectiv scoaterea unui element din mulțime, operatorul << pentru introducerea conținutului mulțimii într-un flux de ieșire, operatorul ! care returnează numărul de elemente al mulțimii, operatorii =, +=, -=, *=, ==, !=, constructor fără parametri, constructor de copiere, destructor etc. Se vor folosi metode rapide pentru reuniune, diferență și intersecție.

- 4) Clasă șablon pentru prelucrarea unui vector. Pentru citire se va folosi >>, pentru scriere <<, atribuirea se va face cu =, compararea de doi vectori cu == și !=, calcularea normei cu !, suma vectorială cu +, diferența vectorială cu -, amplificarea cu scalar cu *, produsul scalar cu *, se vor defini operatorii +=, -= și *=, funcții pentru calcularea mediei aritmetice, pentru sortare etc. Să se deriveze clasa *vector* pentru lucrul cu un vector tridimensional. Să se definească în această clasă în plus operatorul / pentru produs vectorial.
- 5) Clasă șablon de lucru cu *matrici*. Clasa va conține: operatorul >> pentru citire, scrierea se va face cu <<, atribuirea cu =, compararea cu == și !=. Determinantul se va calcula cu !, suma cu +, diferența cu -, produsul cu *. De asemenea, se vor defini operatorii +=, -= și *= și funcții pentru transpusă, ridicarea la putere și inversarea matricii. Să se folosească această clasă pentru calcularea sumei: $A' + (A')^2 + \dots + (A')^n$, unde n este un număr întreg pozitiv.
- 6) Clasă de lucru cu *polinoame*. Clasa va conține operatorii << și >> pentru introducere și scoatere în / din flux, operatorii =, ==, !=, operatorii +, - și * pentru operațiile între două matrici, * pentru amplificare cu o valoare, / pentru câtul împărțirii și % pentru restul împărțirii. De asemenea, se vor descrie operatorii +=, -=, *=, /=, %= și o funcție pentru calcularea valorii polinomului într-un punct. Să se folosească aceasta clasă pentru calcularea celui mai mare divizor comun și a celui mai mic multiplu comun pentru două polinoame.
- 7) Clasă de lucru cu numere întregi mari într-o baza b . Cifrele numărului (valori între 0 și $b-1$) se vor memora într-un șir de numere întregi. Se vor descrie operatorii: << (afișare), >> (citire), = (atribuire); <, >, <=, >=, ==, != pentru comparare; +, -, *, / și % pentru operații aritmetice, operatorii +=, -=, *=, /= și %= . Să se testeze clasa pentru calcularea lui $n!$ pentru un număr întreg pozitiv relativ mare n (de exemplu pentru $n = 20$).
- 8) Clasă pentru transformări elementare aplicate unui punct din plan, de coordonate (x, y) . Cu ajutorul clasei să se poată aplica translații, rotații în jurul originii, simetrii față de axe de coordonate unui punct din plan. Folosind această clasă să se calculeze pentru un punct bidimensional simetricul față de o dreaptă oarecare dată sub forma generală $ax+by+c=0$.

2.3. Fluxuri în C++

Obiective

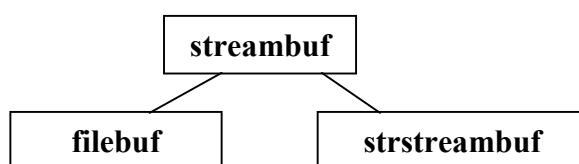
Ne propunem să studiem două ierarhii de clase: *streambuf* și *ios* pentru o mai bună înțelegere a modului de lucru cu fluxuri în C++.

Un limbaj de programare este proiectat să lucreze cu o varietate mare de periferice. Datele se transmit spre echipamentele periferice prin intermediul unei zone de memorie tampon. Gestionarea acestei zone de memorie se face prin intermediul unui așa numit flux. Un flux este un instrument logic care permite tratarea unitară a operațiilor de intrare/ieșire.

Pentru lucrul cu fluxuri în C++ sunt definite în fișierul antet *iostream.h* două ierarhii de clase: una pentru buffer-ul (zona de memorie tampon) cu care lucrează un flux, ierarhie pornită de la clasa **streambuf** și una pentru operațiile pe fluxuri de intrare sau / și de ieșire (clasa **ios** pornește această ierarhie). Ierarhia **ios** reprezintă o alternativă orientată pe obiecte la funcțiile din fișierul antet *stdio.h*.

2.3.1. Ierarhia *streambuf*

Din clasa **streambuf** sunt derivate două clase: **filebuf** (care gestionează buffer-ele pentru fluxurile de fișiere) și **strstreambuf** care gestionează buffer-ele pentru fluxurile de lucru cu string-uri:



Clasa *streambuf* are 2 constructori:

```
streambuf()  
streambuf(char *s, int n)
```

Primul construiește un obiect de tipul clasei *streambuf* cu un buffer vid.

Al doilea constructor specifică adresa *s* spre o zonă de memorie care se va folosi ca buffer pentru flux, iar *n* reprezintă lungimea acestei zone de memorie.

Clasa *streambuf* conține o mulțime de metode pentru prelucrarea cu buffer-ului unui flux. Pentru a lucra cu un flux nu este nevoie să cunoaștem facilitățile clasei ce gestionează buffer-ul asociat fluxului, acestea fiind exploatate indirect de către obiectele de tip ierarhiei *ios*. De aceea nu vom intra în detalii cu privire la conținutul clasei *streambuf* (metode și câmpuri) sau al vreunei clase derivate din aceasta. Dacă se dorește acest lucru, se poate consulta documentația Borland C++ sau/și Visual C++.

Clasa **filebuf** este scrisă pentru buffer-e de fișiere. Ea are 3 constructori:

```
filebuf()  
filebuf(int fd)  
filebuf(int fd, char *s, int n)
```

Primul constructor crează un buffer de fișier, dar nu-l asociază cu nici un fișier.

Al doilea constructor crează un buffer și îl asociază unui fișier având descriptorul *fd*, care este un număr întreg.

Al treilea constructor crează un buffer de caractere al cărui adresă de memorie este *s*, buffer-ul are lungimea *n*, iar constructorul asociază acest buffer unui fișier cu descriptorul *fd*.

Clasa *filebuf* are un destructor care închide eventualul fișier asociat fluxului:

```
~filebuf()
```

Clasa **strstreambuf** este scrisă pentru operații de intrare / ieșire în / din zone de memorie RAM. Clasa are 5 constructori și nici un destructor.

2.3.2. Ierarhia *ios*

Ierarhia de clase pornită de **ios** este scrisă pentru a lucra cu fluxuri de date. Transmiterea și primirea datelor se face prin intermediul unei zone tampon prelucrate prin intermediul unui obiect instanță al unei clase din ierarhia *streambuf*.

Clasa *ios* are 2 constructori:

```
ios(streambuf *buf)
ios()
```

Primul constructor asociază un buffer *buf*, obiect al clasei *streambuf* unui flux, constructorul primește ca argument adresa spre obiectul de tip buffer cu care va lucra fluxul. Al doilea constructor crează un obiect al clasei *ios*, fără a-l lega la un buffer. Legarea se poate face ulterior cu ajutorul funcției *init*, care este membră clasei *ios*.

Clasa *ios* conține următoarele câmpuri:

1) *adjustfield* este o constantă statică de tip *long* transmisă de obicei în al doilea parametru al funcției *setf* membre clasei *ios* pentru a curăța biții de formatare legați de modul de aliniere la stânga, la dreapta și internal (vezi capitolul dedicat indicatorilor de formatare și funcția *setf*). Exemplu: `cout<<setf(ios::right, ios::adjustfield)<<n;`

2) *basefield* tot o constantă statică de tip *long* și este transmisă ca al doilea parametru al funcției *setf* pentru a curăța biții de formatare legați de baza de numerație (8, 10 și 16).

3) *bp* memorează adresa către buffer-ul fluxului, un obiect de tip *streambuff*.

4) *floatfield* este o constantă statică de tip *long* transmisă în al doilea parametru al funcției *setf* pentru a curăța biții de formatare legați de modul de afișare al valorilor în virgulă mobilă (forma fixă și științifică, fără și respectiv cu exponent).

5) *state* este o valoare de tip întreg *int* ce memorează starea buffer-ului de tip *streambuf*.

6) *x_fill* este o valoare întreagă *int* care reține caracterul folosit pentru umplerea spațiilor libere în afișarea formatată.

7) *x_flags* este o valoare de tip *long* care reține biții de formatare la afișarea valorilor numerice întregi (aliniere, baza de numerație etc.).

8) *x_precision* este o valoare de tip *long* care reține precizia la afișarea valorilor în virgulă mobilă (numărul de cifre exacte).

9) *x_width* este o valoare de tip *int* care memorează lungimea (numărul de caractere) pe care se face afișarea.

În clasa *ios* sunt definite următoarele funcții membre de lucru cu fluxuri:

1) *int bad()* returnează o valoare întreagă nenulă dacă a apărut cel puțin o eroare în prelucrarea fluxului. Verificarea se face interogând biții *ios::badbit* și *ios::hardfail* ai valorii din câmpul *state* al clasei *ios*.

2) *void clear(int st=0)*; setează câmpul *state* la valoarea întreagă primită ca argument. Dacă *st* este 0 (valoare de altfel implicită), atunci starea fluxului se inițializează din nou ca fiind bună. Un apel de forma *flux.clear()* readuce fluxul în stare bună (fără erori).

3) *int eof()* returnează o valoare nenulă, dacă nu mai sunt date în flux (sfârșit de fișier). De fapt funcția interoghează bitul *ios::eofbit* al câmpului *state*.

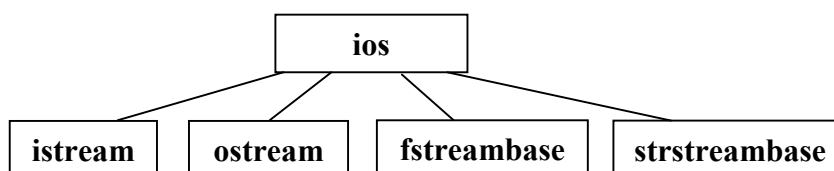
4) *int fail()* returnează o valoare nenulă, dacă o operație aplicată fluxului a eșuat. Se verifică biții *ios::failbit*, *ios::badbit* și *ios::hardfail* ai câmpului *state* (în plus față de funcția *bad* verifică și bitul *ios::failbit*).

5) *char fill()* returnează caracterul de umplere al spațiile libere de la scrierea formatată.

6) *char fill(char c)* setează caracterul de umplere al spațiile goale de la scrierea formatată la valoarea *c* și returnează caracterul folosit anterior în acest scop.

- 7) *long ios_flags()* returnează valoarea reținută în *x_flags*.
- 8) *long ios_flags(long flags)* setează câmpul *x_flags* la valoarea primită ca argument și returnează vechea valoare.
- 9) *int good()* returnează o valoare nenulă dacă nu au apărut erori în prelucrarea fluxului. Acest lucru se realizează consultând biții câmpului *state*.
- 10) *void init(streambuf *buf)*; transmite adresa buffer-ului de tip *streambuf* cu care va lucra fluxul.
- 11) *int precision(int p)*; setează precizia de tipărire a numerelor reale la valoarea primită ca argument și returnează vechea valoare. De fapt câmpul *ios::precision* se setează la valoarea *p* și se returnează vechea sa valoare.
- 12) *int precision()* returnează valoarea preciziei la tipărire a valorilor reale (reținută în câmpul *ios::precision*).
- 13) *streambuf* rdbuf()* returnează adresa către obiectul responsabil cu buffer-ul fluxului.
- 14) *int rdstate()* returnează starea fluxului (valoarea câmpului *state*).
- 15) *long setf(long setbits, long field)*; resetează biții (îi face 0) din *x_flags* pe pozițiile indicate de biții cu valoare 1 ai parametrul *field* și apoi setează biții din *x_flags* la valoarea 1 pe pozițiile în care biții sunt 1 în parametrul *setbits*. Funcția returnează vechea valoare a lui *x_flags*.
- 16) *long setf(long flags)* modifică biții câmpului *x_flags* la valoare 1 pe pozițiile în care biții parametrului *flags* sunt 1 și returnează vechea valoare a lui *x_flags*.
- 17) *void setstate(int st)*; setează biții câmpului *state* la valoarea 1 pe pozițiile în care biții parametrului *st* sunt 1.
- 18) *void sync_with_stdio()*; sincronizează fișierele *stdio* cu fluxurile *iostream*. În urma sincronizării viteza de execuție a programului scade mult.
- 19) *ostream* tie()* returnează adresa către fluxul cu care se afla legat fluxul curent. De exemplu, fluxurile *cin* și *cout* sunt legate. Legătura dintre cele două fluxuri constă în faptul că atunci când unul dintre cele două fluxuri este folosit, atunci mai întâi celălalt este golit. Dacă fluxul curent (din care este apelată funcția *tie*) nu este legat de nici un flux, atunci se returnează valoarea NULL.
- 20) *ostream* tie(ostream* fl)* fluxul *fl* este legat de fluxul curent, cel din care a fost apelată această funcție. Este returnat fluxul anterior legat de fluxul curent. Ca efect al legării unui flux de un alt flux este faptul că atunci când un flux de intrare mai are caractere în buffer sau un flux de ieșire mai are nevoie de caractere, atunci fluxul cu care este legat este întâi golit. Implicit, fluxurile *cin*, *cerr* și *clog* sunt legate de fluxul *cout*.
- 21) *long unsetf(long l)* setează biții din *x_flags* la valoarea 0 pe pozițiile în care parametrul *l* are biții 1.
- 22) *int width()* returnează lungimea pe care se face afișarea formatată.
- 23) *int width(int l)*; setează lungimea pe care se face afișarea formatată la valoarea primită ca argument și returnează vechea valoare.

Din clasa **ios** sunt derivate 4 clase: **istream**, **ostream**, **fstreambase** și **stringstreambase**:



Clasa **istream** realizează extrageri formate sau neformate dintr-un buffer definit ca obiect al unei clase derivate din *streambuf*.

Constructorul clasei *istream* este:

```
istream(strstream *buf);
```

Constructorul asociază un obiect de tip buffer *buf*, primit ca argument, unui flux de intrare.

Funcțiile membre clasei *istream*:

1) *int gcount()* returnează numărul de caractere neformatate ultima dată extrase.

2) *int get()* extrage următorul caracter din flux. Dacă s-a ajuns la sfârșitul fluxului se returnează valoarea EOF, adică -1.

3) *istream& get(signed char *s, int l, char eol='\n')* extrage un șir de caractere interpretate a fi cu semn de lungime maximă *l-1*, pe care îl depune la adresa *s* dacă nu este sfârșitul fluxului sau dacă nu s-a ajuns la caracterul ce delimitează sfârșitul de linie. Delimitatorul de sfârșit de linie este implicit '\n', dar programatorul poate specifica un alt caracter dacă dorește. La sfârșitul șirului de caractere depus în *s* este adăugat '\0' (sfârșit de string). Delimitatorul nu este extras din flux. Se returnează fluxul *istream* din care s-a făcut citirea.

4) *istream& get(unsigned char *s, int l, char eol='\n')* se extrage din flux un șir de caractere fără semn în maniera explicată mai sus.

5) *istream& get(unsigned char &c)* extrage un singur caracter interpretat a fi fără semn și returnează fluxul *istream* din care s-a făcut citirea.

6) *istream& get(signed char &c)* extrage un caracter cu semn și returnează fluxul *istream* din care s-a făcut citirea.

7) *istream& get(strstreambuf &buf, int c='\n')* extrage un șir de caractere până se întâlnește caracterul de delimitare *c* (al cărui valoare implicită este '\n'), șir pe care îl depune în bufferul *buf*.

8) *istream& getline(signed char* s, int l, char c='\n')* extrage un șir de caractere cu semn de lungime maximă *l* pe care îl depune la adresa *s*, fără a pune în *s* delimitatorul, care implicit este '\n'. Delimitatorul este însă scos din flux.

9) *istream& getline(unsigned char* s, int l, char c='\n')* extrage un șir de caractere fără semn în maniera de mai sus.

10) *istream& ignore(int n=1, int delim=EOF)* se ignoră maxim *n* (implicit 1) caractere din fluxul de intrare sau până când delimitatorul *delim* (care implicit este valoarea constantei *EOF*) este întâlnit.

11) *int peek()* returnează următorul caracter din flux, fără a-l scoate.

12) *istream& putback(char c)* pune înapoi în flux caracterul *c*. Se returnează fluxul de intrare.

13) *istream& read(signed char* s, int n)* extrage din fluxul de intrare un număr de *n* caractere interpretate a fi cu semn pe care le depune la adresa *s*.

14) *istream& read(unsigned char* s, int n)* extrage din fluxul de intrare un număr de *n* caractere fără semn pe care le depune la adresa *s*.

15) *istream& seekg(long poz)* se face o poziționare în flux de la începutul acestuia pe poziția *poz*. Se returnează fluxul de intrare.

16) *istream& seekg(long n, int deunde)* se face o poziționare în flux cu *n* caractere de la poziția specificată în al doilea parametru (*deunde*). Se returnează fluxul de intrare. Parametrul *deunde* poate lua valorile: *ios::beg* (de la începutul fluxului), *ios::cur* de la poziția curentă în flux, sau *ios::end* de la sfârșitul fluxului. Pentru valoarea *ios::cur*, *n* poate fi pozitiv (poziționare la dreapta poziției curente) sau negativ (poziționarea se face înapoi de la poziția curentă). Pentru *ios::beg*, *n* trebuie să fie nenegativ, iar pentru *ios::end*, *n* trebuie să fie nepozitiv.

10) *long tellg()* returnează poziția curentă în flux, de la începutul acestuia.

În clasa *istream* este supraîncărcat și operatorul `>>` pentru extragere de date în modul text din fluxul de intrare.

Clasa **ostream** realizează introduceri formate sau neformate într-un buffer definit ca obiect al unei clase derivate din *streambuf*. Clasa are un singur constructor cu următoarea structură:

```
ostream(streambuf* buf);
```

Constructorul asociază un buffer, obiect al unei clase derivate din *streambuf*, fluxului de ieșire.

Metodele clasei *ostream* sunt:

- 1) *ostream& flush()*; forțează golirea buffer-ului. Returnează fluxul de ieșire.
- 2) *ostream& put(char c)*; introduce caracterul primit ca argument în flux. Se returnează fluxul de ieșire.
- 3) *ostream& seekp(long poz)*; se face poziționare pe poziția *poz* de la începutul fluxului. Se returnează fluxul de ieșire.
- 4) *ostream& seekp(long n, int deunde)*; Se mută poziția curentă în flux cu *n* caractere din locul specificat de parametrul *deunde*. Se returnează fluxul de ieșire. *deunde* poate lua valorile *ios::beg*, *ios::cur*, respectiv *ios::end*.
- 5) *long tellp()*; returnează poziția curentă în flux de la începutul fluxului.
- 6) *ostream& write(const signed* s, int n)*; se introduc *n* caractere de la adresa *s* (șirul *s* este de caractere cu semn) în fluxul de ieșire curent (din care este apelată funcția *write*).
- 7) *ostream& write(const signed* s, int n)*; se introduc în flux *n* caractere luate din șirul *s*.

În clasa *ostream* este supraîncărcat operatorul `<<` pentru introducere de valori în modul text în flux.

Clasa **iostream** este derivată din clasele *istream* și *ostream* și are ca obiecte fluxuri care suportă operații de intrare și ieșire. Clasa *iostream* are un singur constructor care asociază unui buffer *buf*, obiect al clasei *streambuf*, un flux, obiect al clasei *iostream*:

```
iostream(streambuf* buf);
```

Clasa *iostream* nu are funcții membre (numai un constructor), ea moștenind metodele claselor *istream* și *ostream*.

Rezumat

Am studiat pe scurt ierarhia *streambuf* pentru buffer-ele fluxurilor și clasele superioare din ierarhia *ios* scrise pentru prelucrarea fluxurilor. Aceste clase sunt baza prelucrării fișierelor și a string-urilor (capitolele următoare). Din aceste clase practic se moștenesc majoritatea datelor și metodelor necesare pentru a lucra cu fișiere și string-uri.

2.4. Fișiere în C++

Obiective

Ne propunem acum să continuăm studiarea ierarhiei *ios* pentru a vedea cum se lucrează în C++ cu fișiere. Majoritatea facilităților de prelucrare a fișierelor este moștenită din clasele superioare ale ierarhiei *ios*, însă pentru a lucra efectiv cu un fișier trebuie să instanțiem una din clasele *ifstream*, *ofstream* și *fstream*.

Tot din clasa *ios* este derivată și clasa ***fstreambase***, specializată pe fluxuri atașate unor fișiere. Clasa ***fstreambase*** are 4 constructori.

Primul constructor crează un obiect al clasei, pe care nu-l atașează nici unui fișier:

```
fstreambase();
```

Al doilea constructor crează un obiect al clasei ***fstreambase***, deschide un fișier și îl atașează obiectului creat:

```
fstreambase(const char* s, int mod, int prot=filebuf::openprot);
```

Parametrul *mod* specifică modul de deschidere al fișierului (text sau pe octeți, pentru scriere sau pentru citire etc.). Pentru modurile de deschidere sunt definite constante întregi în interiorul clasei ***ios***. Acestea sunt:

Constantă mod deschidere	Semnificație
<i>ios::in</i>	Deschidere fișier pentru citire
<i>ios::out</i>	Deschidere fișier pentru scriere
<i>ios::ate</i>	Se face poziționare la sfârșitul fișierului care e deja deschis
<i>ios::app</i>	Deschidere fișier pentru adăugare la sfârșit
<i>ios::trunc</i>	Trunchiază fișierul
<i>ios::nocreate</i>	Deschiderea fișierului se face numai dacă acesta există
<i>ios::noreplace</i>	Deschiderea fișierului se face numai dacă acesta nu există
<i>ios::binary</i>	Deschiderea fișierului se face în modul binar (pe octeți)

Cu ajutorul operatorului | (ori pe biți) se pot compune modurile de deschidere. Astfel, cu modul compus *ios::binary|ios::in|ios::out* se deschide fișierul pentru citire și scriere pe octeți.

Parametrul *prot* corespunde modului de acces la fișier. Valoarea acestui parametru este luată în considerare numai dacă fișierul nu a fost deschis în modul *ios::nocreate*. Implicit, acest parametru este setat așa încât să existe permisiune de scriere și de citire asupra fișierului.

Al treilea constructor crează obiectul și îl leagă de un fișier deschis deja, al cărui descriptor *d* este primit ca argument:

```
fstreambase(int d);
```

De asemenea, există și un constructor care crează obiectul și îl leagă de fișierul al cărui descriptor este *d*. Se specifică buffer-ul *s* cu care se va lucra, împreună cu lungimea *n* a acestuia:

```
fstreambase(int d, char* s, int n);
```

Ca și funcții membre clasei *fstreambase* avem:

1) *void attach(int d)*; face legătura între fluxul curent (din care se apelează funcția *attach*) cu un fișier deschis, al cărui descriptor este *d*.

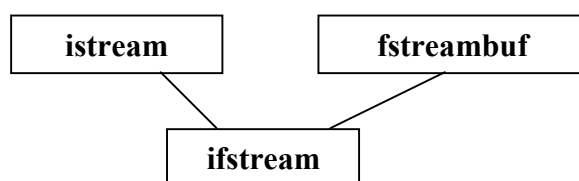
2) *void close()*; închide buffer-ul *filebuf* și fișierul.

3) `void open(const char* s, int mod, int prot=filebuf::openprot);` deschide un fișier în mod similar ca și în cazul celui de-al doilea constructor care are aceeași parametri cu funcția `open`. Atașează fluxului curent fișierul deschis.

4) `filebuf* rdbuf()` returnează adresa către buffer-ul fișierului.

5) `void setbuf(char* s, int n);` setează adresa și lungimea zonei de memorie cu care va lucra obiectul buffer al clasei `filebuf` atașat fișierului.

Din clasele **istream** și **fstreambuf** este derivată clasa **ifstream**, care este o clasă specializată pentru lucrul cu un fișier de intrare (pentru extrageri de date).



Clasa *ifstream* are 4 constructori (asemănători cu cei din clasa *fstreambuf*).

Un prim constructor crează un flux (obiect al clasei **ifstream**) de lucru cu fișiere de intrare, flux pe care nu-l atașează unui fișier:

```
ifstream();
```

Al doilea constructor crează un obiect al clasei *ifstream*, deschide un fișier pentru operații de citire și îl atașează obiectului creat. Pentru ca deschiderea fișierului să se încheie cu succes, trebuie ca fișierul să existe. Semnificația parametrilor este aceeași ca la constructorul al doilea al clasei **fstreambase**:

```
ifstream(const char* s, int mod=în, int prot=filebuf::openprot);
```

Al treilea constructor crează obiectul și îl leagă de un fișier deschis deja, al cărui descriptor *d* este primit ca argument:

```
ifstream(int d);
```

Al patrulea constructor crează obiectul și îl leagă de fișierul al cărui descriptor este *d*. Se specifică adresa de memorie *s* și lungimea *n* acestuia ce va fi folosită ca memorie tampon pentru fișier:

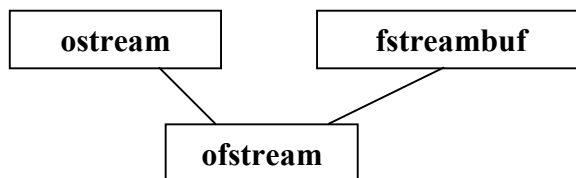
```
ifstream(int d, char* s, int n);
```

Funcții membre clasei *ifstream*:

1) `void open(const char* s, int mod, int prot=filebuf::openprot);` deschide un fișier pentru citire în mod similar cu al doilea constructor al clasei.

2) `filebuf* rdbuf()` returnează adresa către buffer-ul fluxului curent, obiect al clasei *filebuf*.

Din clasele **ostream** și **fstreambuf** este derivată clasa **ofstream** specializată pentru operații de ieșire pe fișiere.



Clasa *ofstream* are, de asemenea, 4 constructori asemănători cu cei din clasa *fstreambuf*. Primul constructor crează un flux pe care nu-l atașează unui fișier:

```
ofstream();
```

Al doilea constructor crează un obiect al clasei *ofstream*, deschide un fișier pentru scriere și îl atașează obiectului creat.

```
ofstream(const char* s, int mod=out, int prot=filebuf::openprot);
```

Semnificația parametrilor este aceeași ca la constructorul al doilea constructor al clasei *fstreambase*.

Al treilea constructor crează obiectul și îl leagă de un fișier deschis pentru operații de scriere, al cărui descriptor *d* este primit ca argument:

```
ofstream(int d);
```

Există și constructorul care crează obiectul și îl leagă de fișierul al cărui descriptor este *d*. Se specifică în plus și adresa zonei de memorie tampon ce se va utiliza precum și lungimea acesteia:

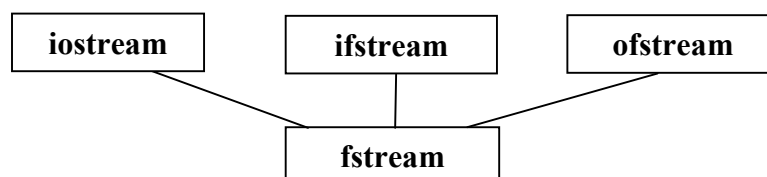
```
ofstream(int d, char* s, int n);
```

Ca funcții membre în clasa **ofstream** avem:

1) *void open(const char* s, int mod, int prot=filebuf::openprot);* deschide un fișier pentru scriere în mod similar cu al doilea constructor al clasei.

2) *filebuf* rdbuf()* returnează adresa către buffer-ul *buf* cu care lucrează fluxul de ieșire.

Din clasele **iostream**, **ifstream** și **ofstream** este derivată clasa **fstream**, care este specializată pentru a lucra cu un fișier în care sunt posibile atât operații de intrare, cât și ieșire:



Clasa *fstream* are, de asemenea, 4 constructori (asemănători cu cei din clasa *fstreambuf*). Un prim constructor crează un flux pe care nu-l atașează nici unui fișier:

```
fstream();
```

Al doilea constructor crează un obiect al clasei **fstream**, deschide un fișier pentru operații de citire și de scriere și îl atașează obiectului creat. Semnificația parametrilor este aceeași ca la constructorul al doilea al clasei **fstreambase**:

```
fstream(const char* s,int mod,int prot=filebuf::openprot);
```

Al treilea constructor crează obiectul și îl leagă de un fișier I/O deschis deja, al cărui descriptor *d* este primit ca argument:

```
fstream(int d);
```

Al patrulea constructor crează obiectul și îl leagă de fișierul al cărui descriptor este *d*. Se specifică adresa și lungimea zonei de memorie tampon:

```
fstream(int d, char* s, int n);
```

Ca funcții membre clasei **fstream** avem:

1) *void open(const char* s, int mod, int prot=filebuf::openprot);* deschide un fișier pentru citire și scriere în mod similar cu al doilea constructor al clasei, care are aceeași parametri cu funcția *open*.

2) *filebuf* rdbuf()* returnează adresa către buffer-ul fișierului.

Pentru a lucra cu un fișier, se instanțiază una dintre clasele: *ifstream*, *ofstream* sau *fstream*. Prelucrarea fișierului se face cu ajutorul metodelor moștenite din clasele: **ios**, **istream**, **ostream**, **fstreambase**.

Pentru a exemplifica modul de lucru cu fișiere dăm listingul câtorva programe:

1) Afișarea pe ecran a conținutului unui fișier text:

```
# include <iostream.h>
# include <fstream.h>
# include <conio.h>
# define lmaxlinie 79 // lungimea maxima a liniei fisierului text

int main()
{
    char numef[100],linie[lmaxlinie+1];
    long n=0;

    cout<<"AFISARE CONTINUT FISIER TEXT"<<endl<<endl;
    cout<<"Dati numele fisierului text: ";
    cin>>numef;
    ifstream fis(numef); // creare obiect si deschidere fisier pentru citire
    if (fis.bad()) // verificare daca fisierul nu a fost deschis cu succes
    {
        cerr<<"Eroare! Nu am putut deschide '"<<numef<<"' pt citire";
        getch();
        return 1;
    }
    while (!fis.eof())
    {
        n++; // numarul de linii afisate
        fis.getline(linie,lmaxlinie); // citire linie din fisier text
```

```

        cout<<linie<<endl; // afisare pe ecran
        if (n%24==0) getch(); // dupa umplerea unui ecran se
    } // asteapta apasarea unei taste
    fis.close();
    cout<<endl<<"AM AFISAT "<<n<<" LINII"<<endl<<flush;
    getch();
    return 0;
}

```

Observații:

- 1) Cu ajutorul funcției *bad()* am verificat dacă fișierul nu a fost deschis cu succes (dacă fișierul nu există pe disc).
- 2) Funcția *getline* (din clasa **istream**) citește o linie de lungime maximă *lmaxlinie*=79 caractere dintr-un fișier text. Dacă linia are mai mult de *lmaxlinie* caractere, atunci sunt citite exact *lmaxlinie* caractere, restul caracterelor de pe linia curentă a fișierului fiind citite data următoare.

2) Program pentru concatenarea a două fișiere:

```

#include <iostream.h>
#include <fstream.h>
#include <conio.h>
#define lbuf 1000 // lungimea buffer-ului de citire din fisier

int main()
{
    ifstream fisis; // fisier pentru operatii de citire
    ofstream fisd; // fisier pentru operatii de scriere
    char numefs1[100], numefs2[100], numefd[100], s[lbuf];
    cout<<"CONCATENARE DE DOUA FISIERE"<<endl<<endl;
    cout<<"Dati numele primului fisier sursa: ";
    cin>>numefs1;
    cout<<"Dati numele celui de-al doilea fisier sursa: ";
    cin>>numefs2;
    cout<<"Dati numele fisierului destinatie: ";
    cin>>numefd;
    fisd.open(numefd, ios::binary); // deschidere fisier pe octeti (mod binar)
    if (fisd.bad()) // verificare daca fisierul nu s-a deschis cu succes
    {
        cerr<<"Eroare! Nu am putut deschide fisierul '"<<numefd<<" pt scriere.";
        getch(); return 1;
    }
    fisis.open(numefs1, ios::binary); // deschidere fisier pe octeti (mod binar)
    if (fisis.bad()) // verificare daca fisierul nu s-a deschis cu succes
    {
        fisd.close();
        cerr<<"Eroare! Nu am putut deschide fisierul '"<<numefs1<<" pt citire.";
        getch(); return 1;
    }
    while (!fisis.eof())

```

```

{
    fiss.read(s,lbuf); // se citesc maxim lbuf baidi din fisier
    fisd.write(s,fiss.gcount()); // se scriu baidii cititi mai sus in fisier
}
fiss.close(); // inchidere fisier
fiss.open(numefs2,ios::binary); // deschidere fisier pe octeti (mod binar)
if (fiss.bad()) // verificare daca fisierul nu s-a deschis cu succes
{
    fisd.close();
    cerr<<"Eroare! Nu am putut deschide fisierul "<<numefs2<<" pt citire.";
    getch(); return 1;
}
while (!fiss.eof())
{
    fiss.read(s,lbuf); // se citesc maxim lbuf baidi din fisier
    fisd.write(s,fiss.gcount()); // se scriu baidii cititi mai sus in fisier
}
fiss.close(); // inchidere fisier
if (fisd.good()) cout<<"Concatenarea s-a incheiat cu succes!"<<endl;
fisd.close(); // inchidere fisier
getch();
return 0;
}

```

Observații:

- 1) Pentru a deschide un fisier în modul binar trebuie specificat explicit acest lucru folosind *ios::binary* în parametrul al doilea al funcției *open*. Dacă nu se face specificarea *ios::binary*, atunci fișierul este deschis în modul text.
- 2) Funcția *gcount()* (din clasa *istream*) returnează numărul de caractere efectiv citite ultima dată cu ajutorul funcției *read*. În programul de mai sus valoarea returnată de *gcount()* este *lbuf*, adică 1000, excepție face ultima citire din fișier.
- 3) Cu ajutorul funcției *good()* am verificat dacă fișierul a fost prelucrat fără să apară erori.

3) Sortarea unui șir citit dintr-un fișier text:

```

# include <iostream.h>
# include <fstream.h>
# include <conio.h>
# include <stdlib.h>

int sort_function(const void *a,const void *b)
{
    float *x=(float *)a,*y=(float *)b;

    if (*x<*y) return -1;
    if (*x==*y) return 0;
    return 1;
}

```

```

int main()
{
    char numef[100];
    int i,n;
    float *a;
    fstream fis;

    cout<<"QUICKSORT"<<endl<<endl;
    cout<<"Dati numele fisierului text cu elementele sinului: ";
    cin>>numef;
    fis.open(numef,ios::in); // deschidere fisier pentru citire
    if (fis.bad())
    {
        cerr<<"Eroare! Nu am putut deschide fisierul '"<<numef<<" pt citire.";
        getch();
        return 1;
    }
    fis>>n; // citirea numarului intreg n din fisier
    a=new float[n];
    if (a==NULL)
    {
        cerr<<"Eroare! Memorie insuficienta.";
        getch();
        return 1;
    }
    for (i=0;i<n;i++)
    {
        if (fis.eof())
        {
            cerr<<"Eroare! Elemente insuficiente in fisier.";
            getch();
            return 1;
        }
        fis>>a[i]; // citirea unui numar real (float)
    }
    fis.close();
    qsort((void *)a, n, sizeof(float), sort_function); // sortare QuickSort
    cout<<"Dati numele fisierului text in care se va depune sinul sortat: ";
    cin>>numef;
    fis.open(numef,ios::out); // deschidere fisier pentru scriere
    if (fis.bad())
    {
        cerr<<"Eroare! Nu am putut crea fisierul '"<<numef<<"'.";
        getch();
        return 1;
    }
    fis<<n<<endl;
    for (i=0;i<n;i++) fis<<a[i]<<" ";
    delete [] a;
    if (fis.good()) cout<<"Am sortat sinul !"<<endl;
    fis.close();
    getch();
}

```

```

    return 0;
}

```

Observații:

- 1) Variabila *fis* este folosită întâi pentru a prelucra un fișier de intrare și apoi pentru unul de ieșire.
- 2) Citirea unor valori dintr-un fișier în modul text a fost realizată cu ajutorul operatorului >>, iar scrierea cu operatorul <<.
- 3) Sortarea șirului am făcut-o cu ajutorul funcției *qsort*, a cărei definiție o găsim în fișierul antet *stdlib.h* sau în *search.h*. Cu ajutorul acestei funcții se pot sorta șiruri de elemente de orice tip. Noi am folosit-o pentru sortarea unui șir de valori de tip *float*. Funcția *sort_function* compară două valori de tipul elementelor din șir. Dacă se dorește o sortare crescătoare, funcția trebuie să returneze o valoare negativă. Se returnează 0, dacă sunt egale valorile și, respectiv, se returnează un număr pozitiv dacă prima valoare e mai mare decât a doua.

4) Gestiunea stocului unei firme:

```

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <process.h>
#include <conio.h>
#include <stdio.h>
#include <string.h>

struct tstoc
{
    char cod_prod[10], den_prod[50];
    double cant, pret;
};

char *numef="stoc.dat"; // numele fisierului in care se retine stocul

void VanzCump() // Vanzare / cumparare dintr-un produs
{
    char *s, cod[10];
    int gasit=0;
    long n=0;
    double cant;
    struct tstoc *st;
    fstream fis;

    s=new char[sizeof(struct tstoc)];
    fis.open(numef, ios::binary|ios::in|ios::out|ios::nocreate);
    if (fis.bad())
    {
        cerr<<endl<<"Eroare! Nu am putut deschide fisierul '"<<numef<<"'.";
        getch(); exit(0);
    }
    cout<<"Dati codul produsului care se vinde / cumpara: ";

```

```

cin>>cod;
while (!fis.eof())
{
    fis.read(s,sizeof(struct tstoc));
    if (fis.good())
    {
        st=(struct tstoc *)s;
        if (!strcmp(cod,st->cod_prod))
        {
            gasit=1;
            cout<<"Denumire:  "<<st->den_prod<<endl;
            cout<<"Cantitate:  "<<st->cant<<endl;
            cout<<"Pret:      "<<st->pret<<endl<<endl;
            cout<<endl<<"Dati cantitatea cu care se modifica stocul: ";
            cin>>cant;
            st->cant+=cant;
            fis.seekp(n*sizeof(struct tstoc),ios::beg);
            fis.write(s,sizeof(struct tstoc));
            cout<<endl<<"Cod produs:  "<<st->cod_prod<<endl;
            cout<<"Denumire:  "<<st->den_prod<<endl;
            cout<<"Cantitate:  "<<st->cant<<endl;
            cout<<"Pret:      "<<st->pret<<endl;
            cout<<"Valoare:    "<<st->pret*st->cant<<endl;
            getch();
        }
    }
    n++;
}
if (!gasit)
{
    cerr<<endl<<"Nu am gasit produs cu acest cod!"<<endl;
    getch();
}
fis.close(); delete [] s;
}

```

void Adaugare() // Introducerea unui nou produs in stoc

```

{
    char *s,*s2;
    int gasit=0;
    struct tstoc *st,*st2;
    fstream fis;

    s=new char[sizeof(struct tstoc)];
    st=(struct tstoc*)s;
    s2=new char[sizeof(struct tstoc)];
    st2=(struct tstoc*)s2;
    fis.open(numef,ios::binary|ios::in);
    if (fis.bad())
    {
        cerr<<endl<<"Eroare! Nu am putut deschide fisierul '"<<numef<<"'.";
        getch(); exit(0);
    }
}

```



```

    }
    cout<<endl<<"Dati codul produsului care se introduce in stoc: ";
    cin>>st->cod_prod;
    while (!fis.eof() && !gasit)
    {
        fis.read(s2,sizeof(struct tstoc));
        if (fis.good())
            if (!strcmp(st->cod_prod,st2->cod_prod)) gasit=1;
    }
    if (!gasit)
    {
        fis.close();
        fis.open(numef,ios::binary|ios::app);
        cout<<"Denumire: ";
        cin>>st->den_prod;
        cout<<"Cantitate: ";
        cin>>st->cant;
        cout<<"Pret: ";
        cin>>st->pret;
        fis.write(s,sizeof(struct tstoc));
    }
    else
    {
        cerr<<"Eroare! Mai exista un produs cu acest cod:"<<endl;
        cout<<endl<<"Cod produs: "<<st->cod_prod<<endl;
        cout<<"Denumire: "<<st->den_prod<<endl;
        cout<<"Cantitate: "<<st->cant<<endl;
        cout<<"Pret: "<<st->pret<<endl;
        cout<<"Valoare: "<<st->pret*st->cant<<endl;
        getch();
    }
    fis.close(); delete [] s2; delete [] s;
}

void AfisProd() // Afisarea unui produs cu un anumit cod
{
    char *s,cod[10];
    int gasit=0;
    struct tstoc *st;
    ifstream fis;
    s=new char[sizeof(struct tstoc)];
    st=(struct tstoc*)s;
    fis.open(numef,ios::binary);
    if (fis.bad())
    {
        cerr<<endl<<"Eroare! Nu am putut deschide fisierul '"<<numef<<"'.";
        getch();
        exit(0);
    }
    cout<<endl<<"Dati codul produsului care se afiseaza: ";
    cin>>cod;
    while (!fis.eof())

```

```

{
    fis.read(s,sizeof(struct tstoc));
    if (fis.good())
    {
        if (!strcmp(st->cod_prod,cod))
        {
            gasit=1;
            cout<<endl<<"Cod produs: "<<st->cod_prod<<endl;
            cout<<"Denumire:  "<<st->den_prod<<endl;
            cout<<"Cantitate: "<<st->cant<<endl;
            cout<<"Pret:      "<<st->pret<<endl;
            cout<<"Valoare:   "<<st->pret*st->cant<<endl;
            getch();
        }
    }
}
if (!gasit)
{
    cerr<<endl<<"Nu am gasit produs cu acest cod!"<<endl;
    getch();
}
fis.close();
delete [] s;
}

void ModifPret() // Modificarea pretului unui produs
{
    char *s,cod[10];
    int gasit=0;
    long n=0;
    double pret;
    struct tstoc *st;
    fstream fis;

    s=new char[sizeof(struct tstoc)];
    fis.open(numef,ios::binary|ios::in|ios::out|ios::nocreate);
    if (fis.bad())
    {
        cerr<<endl<<"Eroare! Nu am putut deschide fisierul '"<<numef<<"'.";
        getch();
        exit(0);
    }
    cout<<"Dati codul produsului pentru care se modifica pretul: ";
    cin>>cod;
    while (!fis.eof())
    {
        fis.read(s,sizeof(struct tstoc));
        if (fis.good())
        {
            st=(struct tstoc *)s;
            if (!strcmp(cod,st->cod_prod))
            {

```

```

        gasit=1;
        cout<<"Denumire:  "<<st->den_prod<<endl;
        cout<<"Cantitate:  "<<st->cant<<endl;
        cout<<"Pret:      "<<st->pret<<endl<<endl;
        cout<<endl<<"Dati noul pret al produsului: ";
        cin>>pret;
        st->pret=pret;
        fis.seekp(n*sizeof(struct tstoc),ios::beg);
        fis.write(s,sizeof(struct tstoc));
        cout<<endl<<"Cod produs: "<<st->cod_prod<<endl;
        cout<<"Denumire:  "<<st->den_prod<<endl;
        cout<<"Cantitate:  "<<st->cant<<endl;
        cout<<"Pret:      "<<st->pret<<endl;
        cout<<"Valoare:    "<<st->pret*st->cant<<endl;
        getch();
    }
}
n++;
}
if (!gasit)
{
    cerr<<endl<<"Nu am gasit produs cu acest cod!"<<endl;
    getch();
}
fis.close();
delete [] s;
}

void Stergere() // Stergerea unui produs cu un anumit cod din stoc
{
    char *s,cod[10];
    int gasit=0;
    struct tstoc *st;
    ifstream fis1;
    ofstream fis2;

    s=new char[sizeof(struct tstoc)];
    fis1.open(nume_f,ios::binary);
    if (fis1.bad())
    {
        cerr<<endl<<"Eroare! Nu am putut deschide fisierul '"<<nume_f<<"'.";
        getch();
        exit(0);
    }
    cout<<"Dati codul produsului care se sterge: ";
    cin>>cod;
    fis2.open("stoc.tmp",ios::binary);
    if (fis2.bad())
    {
        cerr<<endl<<"Eroare! Nu am putut deschide fisierul 'stoc.tmp'.";
        getch(); exit(0);
    }
}

```

```

while (!fis1.eof())
{
    fis1.read(s, sizeof(struct tstoc));
    if (fis1.good())
    {
        st=(struct tstoc *)s;
        if (strcmp(st->cod_prod, cod))
            fis2.write(s, sizeof(struct tstoc));
        else
        {
            gasit=1;
            cout<<endl<<"S-a sters produsul:"<<endl<<endl;
            cout<<"Cod produs: "<<st->cod_prod<<endl;
            cout<<"Denumire: "<<st->den_prod<<endl;
            cout<<"Cantitate: "<<st->cant<<endl;
            cout<<"Pret: "<<st->pret<<endl;
            cout<<"Valoare: "<<st->pret*st->cant<<endl;
            getch();
        }
    }
}
fis2.close();
fis1.close();
if (!gasit)
{
    remove("stoc.tmp");
    cerr<<endl<<"Nu am gasit produs cu acest cod!"<<endl;
    getch();
}
else
{
    remove(numef);
    rename("stoc.tmp", numef);
}
}

void Cautare() // Cautarea unui produs dupa un sir de caractere ce apare in denumire
{
    char *s, nume[50];
    int gasit=0;
    struct tstoc *st;
    ifstream fis;

    s=new char[sizeof(struct tstoc)];
    st=(struct tstoc*)s;
    fis.open(numef, ios::binary);
    if (fis.bad())
    {
        cerr<<endl<<"Eroare! Nu am putut deschide fisierul '"<<numef<<"'.";
        getch();
        exit(0);
    }
}

```

```

cout<<endl<<"Dati sirul de caractere ce apare in numele produsului cautat: ";
cin>>nume;
while (!fis.eof())
{
    fis.read(s,sizeof(struct tstoc));
    if (fis.good())
    {
        if (strstr(st->den_prod,nume) != NULL)
        {
            gasit++;
            cout<<endl<<"Cod produs: "<<st->cod_prod<<endl;
            cout<<"Denumire:  "<<st->den_prod<<endl;
            cout<<"Cantitate: "<<st->cant<<endl;
            cout<<"Pret:      "<<st->pret<<endl;
            cout<<"Valoare:   "<<st->pret*st->cant<<endl;
            getch();
        }
    }
}
if (!gasit)
{
    cerr<<endl<<"Nu am gasit nici un produs!"<<endl;
    getch();
}
else
{
    cout<<endl<<"Am gasit "<<gasit<<" produse!"<<endl;
    getch();
}
fis.close();
delete [] s;
}

```

void AfisStoc() // Afisarea tuturor produselor din stoc

```

{
    char *s;
    int gasit=0;
    struct tstoc *st;
    ifstream fis;

    s=new char[sizeof(struct tstoc)];
    st=(struct tstoc*)s;
    fis.open(numef,ios::binary);
    if (fis.bad())
    {
        cerr<<endl<<"Eroare! Nu am putut deschide fisierul '"<<numef<<".";
        getch(); exit(0);
    }
    cout<<"Stocul este alcatuit din:"<<endl<<endl;
    while (!fis.eof())
    {
        fis.read(s,sizeof(struct tstoc));

```

```

        if (fis.good())
        {
            gasit++;
            cout<<endl<<"Cod produs: "<<st->cod_prood<<endl;
            cout<<"Denumire:  "<<st->den_prood<<endl;
            cout<<"Cantitate:  "<<st->cant<<endl;
            cout<<"Pret:       "<<st->pret<<endl;
            cout<<"Valoare:    "<<st->pret*st->cant<<endl;
            getch();
        }
    }
    if (!gasit)
    {
        cerr<<endl<<"Nu exista nici un produs in stoc!"<<endl;
        getch();
    }
    else
    {
        cout<<endl<<"Am afisat "<<gasit<<" produse!"<<endl;
        getch();
    }
    fis.close();
    delete [] s;
}

void ValStoc() // Afisarea valorii totale a stocului
{
    char *s;
    double total=0;
    struct tstoc *st;
    ifstream fis;

    s=new char[sizeof(struct tstoc)];
    st=(struct tstoc*)s;
    fis.open(numef,ios::binary);
    if (fis.bad())
    {
        cerr<<endl<<"Eroare! Nu am putut deschide fisierul '"<<numef<<"'.";
        getch();
        exit(0);
    }
    while (!fis.eof())
    {
        fis.read(s,sizeof(struct tstoc));
        if (fis.good()) total+=st->cant*st->pret;
    }
    fis.close();
    cout<<"Valoarea totala a stocului este: "<<total<<endl;
    getch();
    delete [] s;
}

```

```

void main()
{
    char c;
    do
    {
        cout<<"GESTIONAREA STOCULUI UNEI FIRME"<<endl<<endl;
        cout<<" 1) Vanzare / cumparare marfa"<<endl;
        cout<<" 2) Adaugare produs in stoc"<<endl;
        cout<<" 3) Afisare produs dupa cod"<<endl;
        cout<<" 4) Modificare pret"<<endl;
        cout<<" 5) Stergere produs din stoc"<<endl;
        cout<<" 6) Cautare produs dupa nume"<<endl;
        cout<<" 7) Afisare stoc"<<endl;
        cout<<" 8) Valoare totala stoc"<<endl;
        cout<<endl<<"Esc - Exit"<<endl<<endl;
        c=getch();
        switch (c)
        {
            case '1': VanzCump(); break;
            case '2': Adaugare(); break;
            case '3': AfisProd(); break;
            case '4': ModifPret(); break;
            case '5': Stergere(); break;
            case '6': Cautare(); break;
            case '7': AfisStoc(); break;
            case '8': ValStoc(); break;
        }
    }
    while (c!=27);
}

```

Observație:

Fluxurile C++ pentru fișiere lucrează numai cu caractere și șiruri de caractere. De aceea, când am folosit funcțiile *read*, respectiv *write* pentru citirea, respectiv scrierea unei variabile de tipul *struct tstoc*, am fost nevoiți să facem conversii de la șir de caractere la tipul adresă către un tip *struct tstoc*. De fapt, în variabilele *s* (de tip adresă către un șir de caractere) și *st* (pointer către tipul *struct tstoc*) s-a reținut aceeași adresă de memorie. Am citit și scris șirul de caractere de lungime *sizeof(struct tstoc)*, șir de caractere aflat la aceeași adresă către care pointează și *st*. Cu alte cuvinte, scrierea și citirea datelor la adresa *st* s-au făcut prin intermediul variabilei *s*.

Rezumat

Am studiat clasele specializate pe lucru cu fișiere: *fstreambase*, *ifstream*, *ofstream* și *fstream* din ierarhia *ios*. Am văzut cum se deschide un fișier în C++, cum se prelucrează și cum se închide folosind facilitățile ierarhiei *ios*. Am putut urmări în finalul prezentării câteva exemple ilustrative la modul de lucru cu fișiere din C++.

2.5. Prelucrarea string-urilor în C++

Obiective

Ne propunem să studiem clasele din ierarhia *ios* specializate pe prelucrarea string-urilor (șiruri de caractere NULL terminate): *istream*, *ostream* și *stringstream*.

În ierarhia de fluxuri *ios* există clase scrise pentru a prelucra string-uri (șiruri de caractere încheiate cu '\0') într-o manieră orientată pe obiecte.

Clasele pentru lucrul cu string-uri sunt definite în fișierul antet **strstream.h**.

Buffer-ele pentru string-uri sunt obiecte ale clasei *strstreambuf*, care este o clasă derivată din *streambuf*.

Clasa **strstreambase** specializează clasa *ios* pentru lucrul cu string-uri, specificându-se faptul că se va lucra cu un buffer de tip **strstreambuf**.

Clasa *strstreambase* are 2 constructori:

```
strstreambase();  
strstreambase(const char* buf, int n, char* start);
```

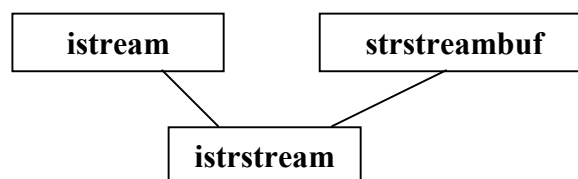
Primul constructor crează un obiect al clasei *strstreambase*, fără a specifica însă șirul de caractere cu care se va lucra. Legătura se va face dinamic cu un șir de caractere prima dată când obiectul va fi utilizat.

Al doilea constructor crează un obiect de *strstreambase*, obiect ce va folosi șirul de caractere aflat la adresa *buf*, care are lungimea *n*, al cărui poziție de pornire este *start*.

Funcția membră *rdbuf* clasei *strstreambase* va returna adresa buffer-ului (obiect al clasei *strstreambuf*), cu care lucrează fluxul de tip string:

```
strstreambuf* rdbuf();
```

Din clasele **istream** și **strstreambase** este derivată clasa **istream**, care este specializată (după cum îi spune numele și clasele din care este derivată) pentru a lucra cu fluxuri de intrare de tip string:



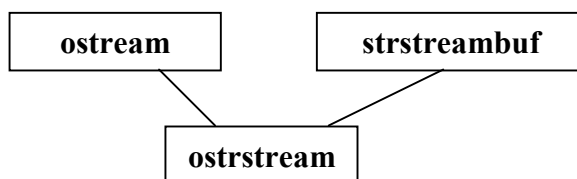
Clasa *istream* are doi constructori:

```
istream(const char* s);  
istream(const char* s, int n);
```

Primul constructor crează un obiect al clasei *istream* și specifică faptul că acesta va lucra cu string-ul *s*.

Al doilea constructor, în plus față de primul, limitează la *n* numărul de caractere din șirul *s* cu care se va lucra. Cu alte cuvinte, string-ul *s* nu va putea fi mai lung de *n* caractere.

Din clasele **ostream** și **stringstream** este derivată clasa **ostringstream**, care este specializată pentru a lucra cu fluxuri de ieșire de tip string (string-ului nu i se vor putea aplica decât operații de scriere).



Clasa **ostringstream** are doi constructori:

```

ostringstream();
ostringstream(char* s, int n, int poz=ios::out);
  
```

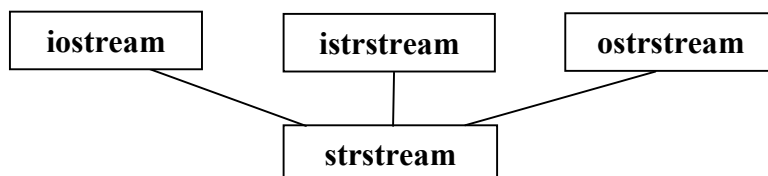
Primul constructor crează un obiect de tipul **ostringstream**, obiect care va lucra cu un șir de caractere alocat dinamic.

Al doilea constructor crează un obiect al clasei **ostringstream** și specifică faptul că acesta va lucra cu șirul de caractere *s* de lungime maximă *n*. Poziționarea în string-ul *s* se face implicit la începutul acestuia. Dacă ultimul parametru este specificat ca având valoarea *ios::app* sau *ios::ate*, atunci poziționarea în string-ul *s* se va face pe ultima poziție a acestuia, adică pe poziția pe care se afla caracterul '\0' (delimitatorul de sfârșit de string).

Pe lângă cei doi constructori, clasa **ostringstream** mai are două funcții membre:

- 1) *int pcount()* returnează numărul de caractere reținute în buffer.
- 2) *char* str()* returnează adresa către șirul de caractere cu care lucrează fluxul.

Din clasele **iostream**, **istream** și **ostringstream** este derivată clasa **stringstream**, care este o clasă de fluxuri ce lucrează cu string-uri ce suportă operații atât de intrare, cât și de ieșire:



Clasa *stringstream* are doi constructori:

```

stringstream();
stringstream(char* s, int n, int poz=ios::out);
  
```

Primul constructor crează un obiect de tipul *stringstream*, obiect care va lucra cu un șir de caractere alocat dinamic.

Al doilea constructor crează un obiect al clasei *stringstream* și specifică faptul că acesta va lucra cu șirul de caractere *s* de lungime maximă *n*. Poziționarea în string-ul *s* se face implicit la începutul acestuia. Dacă valoarea ultimului parametru este *ios::app* sau *ios::ate*, atunci poziționarea în string-ul *s* se va face pe ultima poziție a acestuia, adică pe poziția pe care se afla caracterul '\0' (delimitatorul de sfârșit de string).

Pe lângă cei doi constructori, în clasa *stringstream* există și funcția membră *str()*, care returnează adresa către șirul de caractere cu care lucrează fluxul.

Când se dorește a se lucra cu fluxuri de tip string se instanțiază una dintre clasele: **istream**, **ostream** și **stringstream**. Prelucrarea string-ului se face utilizând metodele din clasele superioare: **ios**, **istream**, **ostream**, **stringstreambase**.

Spre exemplificare, prezentăm următorul program care efectuează operații aritmetice cu numere reale (citirea expresiei matematice se face dintr-un string):

```
# include <stdio.h>
# include <conio.h>
# include <math.h>
# include <iostream.h>
# include <string.h>

void main(void)
{
    char s[100],c=0,op;
    float nr1,nr2;
    stringstream *si;

    do
    {
        cout<<" Calcule matematice (operatiile: +, -, *, / si ^)"<<endl;
        cout<<"_____)"<<endl;
        cout<<endl<<"<numar_real> <operator> <numar_real>: ";
        gets(s);
        cout<<endl;
        si=new stringstream(s); // construire obiect
        *si>>nr1>>op>>nr2;    // extragere real, caracter, real din string-ul s
        if (!si->bad())        // verificare daca extragerea a fost cu succes
            switch (op)
            {
                case '+':
                    cout<<nr1<<op<<nr2<<"="<<(nr1+nr2);
                    break;
                case '-':
                    cout<<nr1<<op<<nr2<<"="<<(nr1-nr2);
                    break;
                case '*':
                    cout<<nr1<<op<<nr2<<"="<<(nr1*nr2);
                    break;
                case '/':
                    if (nr2) cout<<nr1<<op<<nr2<<"="<<(nr1/nr2);
                    else cout<<"Eroare! Impartire prin 0.";
                    break;
                case '^': // ridicare la putere
                    if (nr1>0) cout<<nr1<<op<<nr2<<"="<<pow(nr1,nr2);
                    else cerr<<"Eroare! Primul numar negativ.";
                    break;
                default:
                    cerr<<"Eroare! Operator necunoscut";
            }
        else cerr<<"Eroare! Utilizare incorecta.";
        cout<<endl<<endl<<"Apasati Esc pentru a parasi programul,";
        cout<<" orice alta tasta pentru a continua.";
    }
```

```

        c=getch();
        delete si; // distrugere obiect
    }
    while (c!=27);
}

```

Rezumat

Pentru prelucrarea string-urilor în C++ instanțiem una din clasele: *istrstream*, *ostrstream* sau *strstream*. Practic, un șir de caractere NULL terminat se “îmbracă” într-un obiect pentru a fi prelucrat, urmând ca în final el să poată fi extras din obiect cu metoda *str()* care returnează adresa șirului.

2.6. Clasa *complex* din C++

Obiective

Ne propunem să studiem în final o altă clasă interesantă care se instalează odată cu mediul de programare C++. Este vorba de o clasă scrisă pentru numere complexe. Această clasă ne oferă un mare număr de operatori și funcții pentru numere complexe. Practic, aproape tot ce există pentru numere reale (operatori și funcțiile din fișierul antet *math.h*) este aplicabil și numerelor complexe.

C++ oferă o clasă de lucru cu un număr complex. În această clasă sunt supraîncărcăți o serie de operatori. De asemenea există o mulțime de metode prietene clasei pentru numere complexe.

În continuare vom prezenta clasa **complex** așa cum există ea în Borland C++. Pentru a lucra cu ea trebuie inclus fișierul antet “*complex.h*”.

Clasa *complex* conține două date de tip *double*. Este vorba de *re*, care reține partea reală a numărului complex și de *im*, partea imaginară a numărului complex. Aceste date sunt private.

Clasa *complex* are 2 constructori:

```

complex();
complex(double Re, double Im=0);

```

Primul constructor crează un obiect fără a inițializa partea reală și cea imaginară a numărului complex.

Al doilea constructor inițializează partea reală și pe cea imaginară cu cele două valori primite ca argumente. Pentru al doilea argument există valoarea implicită 0. Evident, în cazul în care valoarea pentru partea imaginară este omisă, la construcția obiectului complex ea se va inițializa cu 0 (vezi capitolul dedicat valorilor implicite pentru parametrii funcțiilor în C++).

Există o mulțime de funcții (matematice) definite ca fiind prietene clasei *complex*:

- 1) *double real(complex &z)* returnează partea reală a numărului complex *z*.
- 2) *double imag(complex &z)* returnează partea imaginară a numărului *z*.
- 3) *double conj(complex &z)* returnează conjugatul numărului complex *z*.
- 4) *double norm(complex &z)* returnează norma numărului complex *z*, adică:

$$\sqrt{re^2 + im^2}$$

5) *double arg(complex &z)* returnează argumentul numărului complex *z* (măsura unui unghi în radiani în intervalul $[0, 2\pi)$).

6) *complex polar(double r, double u=0)*; crează un obiect de tip *complex* pornind de la norma și argumentul acestuia. Numărul complex care se crează este $r \cdot \cos(u) + r \cdot \sin(u)i$. Se returnează obiectul creat.

7) *double abs(complex &z)* returnează modulul numărului complex *z*, adică $re^2 + im^2$ (pătratul normei).

8) *complex acos(complex &z)* returnează arccosinus din numărul complex *z*.

9) *complex asin(complex &z)* returnează arcsinus din numărul complex *z*.

10) *complex atan(complex &z)* returnează arctangentă din numărul complex *z*.

11) *complex cos(complex &z)* returnează cosinus din numărul complex *z*.

12) *complex cosh(complex &z)* returnează cosinus hiperbolic din *z*.

13) *complex exp(complex &z)* returnează e^z .

14) *complex log(complex &z)* returnează logaritm natural din numărul *z*.

15) *complex log10(complex &z)* returnează logaritm zecimal (în baza 10) din *z*.

16) *complex pow(double r, complex &z)* ridică numărul real *r* la puterea *z*.

17) *complex pow(complex &z, double r)* ridică numărul complex *z* la puterea *r*.

18) *complex pow(complex &z1, complex &z2)* ridică *z1* la puterea *z2*.

19) *complex sin(complex &z)* returnează sinus din numărul complex *z*.

20) *complex sinh(complex &z)* returnează sinus hiperbolic din *z*.

21) *complex sqrt(complex &z)* returnează radical din numărul complex *z*.

22) *complex tan(complex &z)* returnează tangentă din numărul complex *z*.

23) *complex tanh(complex &z)* returnează tangentă hiperbolică din *z*.

În clasa *complex* sunt supraîncărcați operatorii aritmetici: +, -, *, / de câte 3 ori. Astfel ei funcționează între două numere complexe, între un *double* și un *complex* și între un *complex* și un *double*.

Există definiți operatorii de tip atribuire combinat cu un operator aritmetic: +=, -=, *=, /=. Acești operatori sunt definiți sub două forme: între două numere complexe și între un *complex* și un *double*.

Două numere complexe se pot compara cu ajutorul operatorilor == și !=.

Operatorii unari pentru semn (+, respectiv -) sunt de asemenea definiți în clasa *complex* (acești operatori permit scrierea +z, respectiv -z).

Extragerea, respectiv introducerea unui număr complex dintr-un / într-un flux se poate face cu ajutorul operatorilor <<, respectiv >>, care sunt definiți ca fiind externi clasei *complex*, ei nu sunt nici măcar prieteni clasei. Citirea și afișarea numărului complex se face sub forma unei perechi de numere reale (*re*, *im*) (partea reală și partea imaginară despărțite prin virgulă, totul între paranteze rotunde). De exemplu, perechea (2.5, 7) reprezintă numărul complex $2.5 + 7i$.

Propunem în continuare o aplicație de tip calculator pentru numere complexe pentru a exemplifica modul de lucru cu numere complexe:

```
# include <stdio.h>
# include <conio.h>
# include <complex.h>
# include <iostream.h>
# include <strstrea.h>

void main(void)
{
    char s[100], c=0, op;
    complex nr1, nr2; // doua numere complexe (obiecte ale clasei complex)
```

```

    istrstream *si;

do
{
    cout<<" Calcule cu numere complexe (+, -, *, / si ^)"<<endl;
    cout<<"-----)"<<endl;
    cout<<endl<<"<numar_complex> <operator> <numar_complex>: ";
    gets(s);
    cout<<endl;
    si=new istrstream(s);
    *si>>nr1>>op>>nr2;
    if (si->good())
        switch (op)
        {
            case '+':
                cout<<nr1<<op<<nr2<<"="<<(nr1+nr2);
                break;
            case '-':
                cout<<nr1<<op<<nr2<<"="<<(nr1-nr2);
                break;
            case '*':
                cout<<nr1<<op<<nr2<<"="<<(nr1*nr2);
                break;
            case '/':
                if (abs(nr2)) cout<<nr1<<op<<nr2<<"="<<(nr1/nr2);
                else cout<<"Eroare! Impartire prin 0.";
                break;
            case '^': // ridicare la putere
                cout<<nr1<<op<<nr2<<"="<<pow(nr1,nr2);
                break;
            default:
                cerr<<"Eroare! Operator necunoscut.";
        }
    else cerr<<"Eroare! Utilizare incorecta.";
    cout<<endl<<endl<<"Apasati Esc pentru a parasi programul,";
    cout<<" orice alta tasta pentru a continua.";
    c=getch();
    delete si;
}
while (c!=27);
}

```

Dacă rulăm programul de mai sus și introducem de la tastatură: $(2.5, 7) + (1.31, -2.2)$, pe ecran se va afișa: $(2.5, 7) + (1.32, -2.3) = (3.82, 4.7)$.

Rezumat

În finalul fisuței noastre despre programarea orientată pe obiecte din C++ am prezentat clasa *complex* scrisă pentru lucrul cu numere complexe. Nu prea avem membri în clasa *complex* ci mai mult funcții prietene pentru a apropia mai mult modul de lucru cu numere complexe de cel cu numere reale din C.

Indicații și răspunsuri

Cap. 1.5. și 1.6.

Problemele 1. și 2. Se vor folosi manipulatorii *setprecision*, *setw* și *setiosflags*, ultimul cu indicatorii de formatare: *ios::left* (pentru texte), respectiv *ios::right* (pentru valori numerice), *ios::fixed* și *ios::showpoint* (pentru numere reale).

Cap. 1.7.

Problemele 1., 2. și 3. Este de preferat (este mai ușor) să se adapteze prima metodă de alocare dinamică a memorie pentru o matrice la aceste probleme (cea cu $m+1$ alocări).

Cap. 1.8.5.

Problema 1. Un număr natural n mai mare ca doi ($n > 2$) este prim dacă nu se împarte la 2 și la nici un număr întreg impar între 3 și parte întreagă din radical din n . Atenție! 0 și 1 nu sunt prime.

Cap. 1.9.

Se va folosi metoda prezentată mai sus pentru reuniune: sortarea unuia dintre vectori și căutarea rapidă în vectorul sortat.

Cap. 1.10.

Posibile excepții ce pot fi tratate sunt: scoaterea unui element care nu aparține unei mulțimi și reuniunea a două mulțimi care conduce la o mulțime cu mai mult de 1000 de elemente.

Cap. 2.2. (la sfârșitul subcapitolului 2.2.19.)

Problema 3. Vezi exemplul de la capitolul dedicat supraincărcării operatorilor

Problema 4. Produsul vectorial a doi vectori tridimensionali este:

$$\vec{a} \times \vec{b} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$$

Problema 5. Calcularea sumei se poate face mai rapid grupând termenii astfel:

$$A^t + (A^t)^2 + \dots + (A^t)^n = A^t + A^t(A^t + A^t(A^t + \dots + A^t \cdot A^t)).$$

Problema 6. Pentru cel mai mare divizor comun se va folosi algoritmul lui Euclid, iar cel mai mic multiplu comun se poate găsi cu formula:

$$c.m.m.m.c.(P,Q) = \frac{P \cdot Q}{c.m.m.d.c.(P,Q)}.$$

Problema 8. Se aplică dreptei d o translație și o rotație așa încât dreapta să se suprapună peste axa Ox , se găsește simetricul față de Ox , și în final se aplică inversele rotației și a translației inițiale. Se obține că simetricul unui punct P față de dreapta d este dat de compunerea următoarelor transformări elementare:

$$T_{X_p, Y_p} \circ R_\alpha \circ S_{Ox} \circ R_{-\alpha} \circ T_{-X_p, -Y_p}, \text{ unde } \alpha \text{ este măsura unghiului dintre axa } Ox \text{ și}$$

dreapta d . Avem:

$$\begin{cases} \cos(\alpha) = \frac{a}{\sqrt{a^2 + b^2}} \\ \sin(\alpha) = \frac{b}{\sqrt{a^2 + b^2}} \end{cases}.$$

ANEXA 1 - Folosirea mouse-ului sub DOS

Mediul de programare Borland C/C++ pentru DOS nu oferă facilități directe de utilizare a mouse-ului. Propunem în continuare câteva funcții pentru utilizarea mouse-lui atât în modul text cât și în modul grafic.

Scrieți următorul cod într-un fișier cu numele *mouse.h*. Includeți acest fișier de fiecare dată când aveți nevoie să scrieți programe în care se folosește mouse-ul.

```
# include <dos.h>
# include <stdio.h>

typedef struct graphtype
{
    char screenmask[16];
    char cursormask[16];
    int xactive, yactive;
}graphshapetype;

# define screenmask
{ \
    0xFEFF, 0xFEFF, 0xFEFF, 0xFEFF, 0xFEFF, 0xFEFF, 0xFEFF, 0x0001, \
    0xFEFF, 0xFEFF, 0xFEFF, 0xFEFF, 0xFEFF, 0xFEFF, 0xFEFF, 0xFEFF \
};

# define cursormask
{ \
    0x0100, 0x0100, 0x0100, 0x0100, 0x0100, 0x0100, 0x0100, 0xFFFE, \
    0x0100, 0x0100, 0x0100, 0x0100, 0x0100, 0x0100, 0x0100, 0x0000 \
};

# define waitdoubleclick 300;

typedef struct ppe
{
```

```

    char ofs,seg;
    }ptrtype;

struct REGPACK r;

void clicknumber(int *buttons,int *clicks,int *x,int *y);
void defgraphlocator(graphshapetype shape);
void defsensitivity(int deltax,int deltax);
void deftextlocator(int styleflag,char scrmask,char cursmask);
void defxrange(int xmin,int xmax);
void defyrange(int ymin,int ymax);
void getbuttonpress(int *button,int *n,int *x,int *y);
void getbuttonrelease(int *button,int *n,int *x,int *y);
void getmotion(int *deltax,int *deltay);
void getmouse(int *button,int *x,int *y);
void hidelocator();
void resetmouse(int *foundmouse,int *buttons);
void setdoublespeed(int speed);
void setmouse(int x,int y);
void showlocator();
int foundmouse();

void resetmouse(int *foundmouse,int *buttons) // activeaza mouse-ul
{
    r.r_ax = 0;
    intr(0x33,&r);
    *buttons=r.r_bx;
    *foundmouse=(! r.r_ax==0);
}

void showlocator() // face sa apara cursorul mouse-lui
{
    r.r_ax=1;
    intr(0x33,&r);
}

void hidelocator() // ascunde cursorul mouse-lui
{
    r.r_ax=2;
    intr(0x33,&r);
}

void getmouse(int *button,int *x,int *y) // returneaza pozitia mouse-lui
{
    // si combinatia de butoane apasate
    r.r_ax=3;
    intr(0x33,&r);
    *button=r.r_bx;
    *x=r.r_cx;
    *y=r.r_dx;
}

void setmouse(int x,int y) // pozitioneaza mouse-ul pe ecran la coordonatele (x,y)

```



```

    {
        r.r_ax=4;
        r.r_cx=x;
        r.r_dx=y;
        intr(0x33,&r);
    }

void getbuttonpress(int *button,int *n,int *x,int *y)
    {
        r.r_ax=5;
        r.r_bx=*button;
        intr(0x33,&r);
        *button=r.r_ax;
        *n=r.r_bx;
        *x=r.r_cx;
        *y=r.r_dx;
    }

void getbuttonrelease(int *button,int *n,int *x,int *y) // returneaza butoanele apasate
    {
        r.r_ax=6;
        r.r_bx=*button;
        intr(0x33,&r);
        *button=r.r_ax;
        *n=r.r_bx;
        *x=r.r_cx;
        *y=r.r_dx;
    }

void clicknumber(int *buttons,int *clicks,int *x,int *y) // returneaza nr. de click-uri
    {
        getmouse(buttons,x,y);
        if (*buttons==1)
            {
                delay(300);
                *buttons=0;
                getbuttonpress(buttons,clicks,x,y);
            }
        else *clicks=0;
    }

void defxrange(int xmin,int xmax) // defineste limitele inferioare si
    {                                     // superioare pe orizontala ecranului
        r.r_ax=7;
        r.r_cx=xmin;
        r.r_dx=xmax;
        intr(0x33,&r);
    }

void defyrange(int ymin,int ymax) // defineste limitele inferioare si
    {                                     // superioare pe verticala ecranului
        r.r_ax=8;

```

```

r.r_cx=ymin;
r.r_dx=ymax;
intr(0x33,&r);
}

```

```

void defgraphlocator() // defineste cursorul în modul grafic
{
    r.r_ax=9;
    r.r_bx=1;//activ x
    r.r_cx=1;//activ y
    r.r_dx=0xfe;
    r.r_es=0x01;
    intr(0x33,&r);
}

```

```

void deftextlocator(int styleflag,char scmask,char cursmask) // defineste cursorul
                                                             // în modul text
{
    r.r_ax=10;
    if (styleflag) r.r_bx=0; else r.r_bx=1;
    r.r_cx=scmask;
    r.r_dx=cursmask;
    intr(0x33,&r);
}

```

```

void getmotion(int *deltax,int *deltay) // returneaza pasul de miscare
                                         // pe orizontala si pe verticala
{
    r.r_ax=11;
    intr(0x33,&r);
    *deltax=r.r_cx;
    *deltay=r.r_dx;
}

```

```

void defsensivity(int deltax,int deltay) // defineste sensibilitatea la miscare
                                         // pe orizontala si pe verticala
{
    r.r_ax=15;
    r.r_cx=deltax;
    r.r_dx=deltay;
    intr(0x33,&r);
}

```

```

void setdoublespeed(int speed)
{
    r.r_ax=19;
    r.r_dx=speed;
    intr(0x33,&r);
}

```

Ca aplicație la utilizarea mouse-ului în modul grafic propunem desenarea de cercuri, pătrate și elipse la apăsarea butoanelor stânga, dreapta, respectiv stânga împreună cu dreapta (simultan):

```
# include <conio.h>
```

```

# include <string.h>
# include <graphics.h>
# include "mouse.h" // includere fisier cu functiile pentru mouse de mai sus

# define r 40 // dimensiune figuri (cercuri si patrate)

void main(void)
{
    char s[10];
    int gd=DETECT, gm, buton, x, y, dx, dy;
    initgraph(&gd, &gm, "");
    showlocator(); // face sa apara cursorul mouse-ului
    do
    {
        getmouse(&buton, &x, &y); // returnare buton si pozitie mouse
        getmotion(&dx, &dy); // returnare valori deplasare mouse
        if (dx || dy) // verificare daca s-a miscat mouse-ul
        {
            setfillstyle(SOLID_FILL, RED);
            setcolor(WHITE);
            bar(0, 0, 56, 10);
            sprintf(s, "%3d/%3d", x, y);
            outtextxy(1, 2, s); // afisare pozitie cursor mouse
        }
        switch (buton)
        {
            case 1: // click buton stanga mouse
                setcolor(YELLOW);
                circle(x, y, r);
                break;
            case 2: // click buton dreapta mouse
                setcolor(LIGHTCYAN);
                rectangle(x-r, y-r, x+r, y+r);
                break;
            case 3: // click butoane stanga+dreapta mouse
                setcolor(LIGHTGREEN);
                ellipse(x, y, 0, 360, r, 2*r);
                break;
        }
    } while (!kbhit());
    getch(); closegraph();
}

```

Aplicația pe care o propunem pentru utilizarea mouse-ului în modul text este afișarea caracterelor *x* și *o* la apăsarea butoanelor stânga, respectiv dreapta:

```

# include <conio.h>
# include <stdio.h>
# include "mouse.h" // includere fisier cu functiile pentru mouse de mai sus

void main(void)
{

```

```

int buton,x,y,dx,dy;
textbackground(0); clrscr();
showlocator(); // face sa apara cursorul mouse-lui
do
{
    getmouse(&buton,&x,&y); // returnare buton si pozitie mouse
    getmotion(&dx,&dy); // returnare valori deplasare mouse
    if (dx || dy) // verificare daca s-a miscat mouse-ul
    {
        textcolor(WHITE);
        textbackground(RED);
        gotoxy(1,1);
        cprintf("%3d/%3d",x,y); // afisare pozitie cursor mouse
    }
    switch (buton)
    {
        case 1: // click buton stanga mouse
            textbackground(0);
            textcolor(YELLOW);
            gotoxy(x/8+1,y/8+1);
            cprintf("o");
            break;
        case 2: // click buton dreapta mouse
            textbackground(0);
            textcolor(LIGHTCYAN);
            gotoxy(x/8+1,y/8+1);
            cprintf("x");
            break;
    }
}
while (!kbhit()); // cand se apasa buton de la tastatura se paraseste programul
getch();
}

```

Observație: Poziția mouse-lui în modul text este dată de către funcția *getmouse* tot în puncte (ca și în modul grafic). Rezoluția ecranului în modul text obișnuit *co80* este 640x200. De aceea, pentru a afla poziția mouse-lui în coordonate text, trebuie ca la poziția în puncte împărțită la 8 să se adauge 1. Astfel, obținem coordonatele text ale cursorului mouse-lui $(X,Y) = (x/8+1,y/8+1)$. Evident obținem că $X \in \{1, 2, \dots, 80\}$ și $Y \in \{1, 2, \dots, 25\}$, pornind de la coordonatele în puncte (x,y) returnate de funcția *getmouse*, unde $x \in \{0, 8, 16, \dots, 632\}$ și $y \in \{0, 8, 16, \dots, 192\}$.

Scrieți un program C în care se citesc coordonatele vârfurilor unui poligon. Translați și rotiți poligonul pe ecran cu ajutorul mouse-lui.

ANEXA 2 - Urmărirea execuției unui program. Rularea pas cu pas.

Pentru a vedea efectiv traseul de execuție și modul în care se își modifică variabilele valorile într-un program, putem rula pas cu pas. Acest lucru se face în Borland C/C++ cu ajutorul butoanelor **F7** sau **F8**, iar în Visual C++ cu **F11**, combinații de taste care au ca efect rularea liniei curente și trecerea la linia următoare de execuție.

Execuția programului până se ajunge la o anumită linie se face apăsând pe linia respectivă butonul **F4** în Borland C/C++ și respectiv **Ctrl+F10** în **Visual C++**.

În Borland C/C++ pentru a fi posibilă urmărirea execuția unui program, în meniul Options, la Debugger, trebuie selectat On în Source Debugging ! În lipsa acestei setări, dacă se încearcă execuția pas cu pas, se afișează mesajul de atenționare **WARNING: No debug info. Run anyway?**. Este bine ca la *Display Swapping* (în fereastra *Source Debugging*) să se selecteze opțiunea *Always*, altfel fiind posibilă alterarea afișării mediului de programare. Reafișarea mediului de programare se poate face cu *Repaint desktop* din meniul \equiv . Este bine de știut că informațiile legate de urmărirea execuției sunt scrise în codul executabil al programului ceea ce duce la o încărcare inutilă a memoriei când se lansează în execuție aplicația. Așa că programul, după ce a fost depanat și este terminat, este indicat să fie compilat și link-editat cu debugger-ul dezactivat.

Pentru ca execuția programului să se întrerupă când se ajunge pe o anumită linie (*break*), se apasă pe linia respectivă **Ctrl+F8** în Borland C/C++ și **F9** în Visual C++. Linia va fi marcată (de obicei cu roșu). Pentru a anula un *break* se apasă tot **Ctrl+F8**, respectiv **F9** pe linia respectivă.

Execuția pas cu pas a unui program poate fi oprită apăsând **Ctrl+F2** în Borland C/C++ și **F7** în Visual C++.

Dacă se dorește continuarea execuției programului fără *Debugger*, se poate apăsa **Ctrl+F9** în Borland C/C++ și **F5** în Visual C++.

În orice moment, în Borland C/C++ de sub DOS rezultatele afișate pe ecran pot fi vizualizate cu ajutorul combinației de taste **Alt+F5**.

În Borland C/C++ valorile pe care le iau anumite variabile sau expresii pe parcursul execuției programului pot fi urmărite în fereastra **Watch**, pe care o putem deschide din meniul *Window*. Adăugarea unei variabile sau a unei expresii în **Watch** se face apăsând **Ctrl+F7**, sau **Insert** în fereastra **Watch**. Dacă se apasă **Enter** pe o expresie sau variabilă din fereastra **Watch**, aceasta poate fi modificată.

În Visual C++ valoarea pe care o are o variabilă pe parcursul urmăririi execuției unui program poate fi aflată mutând cursorul pe acea variabilă. În timpul urmăririi execuției programului putem vedea rezultatul unei expresii apăsând **Shift+F9**, după ce acea expresie este introdusă.

BIBLIOGRAFIE

1. A. Deaconu, *Programare avansată în C și C++*, Editura Univ. "Transilvania", Brașov, 2003.

2. J. Bates, T. Tompkins, *Utilizare Visual C++ 6*, Editura Teora, 2000.
3. Nabajyoti Barkakati, *Borland C++ 4. Ghidul programatorului*, Editura Teora, 1997.
4. B. Stroustrup, *The C++ Programming Language*, a doua ediție, Addison-Wesley Publishing Company, Reading, MA, 1991..
5. T. Faison, *Borland C++ 3.1 Object-Oriented Programming*, ediția a doua, Sams Publishing, Carmel, IN, 1992.
6. R. Lafore, *Turbo C++ - Getting Started*, Borland International Inc., 1990.
7. R. Lafore, *Turbo C++ - User Guide*, Borland International Inc., 1990.
8. R. Lafore, *Turbo C++ - Programmer's Guide*, Borland International Inc., 1990.
9. O. Catrina, I. Cojocaru, *Turbo C++*, ed. Teora, 1993.
10. M. A. Ellis, B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company, Reading, MA, 1990.
11. S. C. Dewhurst, K. T. Stark, *Programming in C++*, Prentice Hall, Englewood Cliffs, NJ, 1989.
12. E. Keith Gorlen, M. Sanford, P. S. Plexico, *Data Abstraction and Object-Oriented-Programming in C++*, J. Wiley & Sons, Chichester, West Sussex, Anglia, 1990.
13. B. S. Lippman, *C++ Primer*, ediția a doua, Addison-Wesley, Reading, MA, 1991.
14. C. Spircu, I. Lopatan, *Programarea Orientată spre Obiecte*, ed. Teora.
15. M. Mullin, *Object-Oriented Program Design with Examples in C++*, Addison-Wesley, Reading, MA, 1991.
16. I. Pohl, *C++ for C Programmers*, The Benjamin/Cummings Publishing Company. Redwood City, CA, 1989.
17. T. Swan, *Learning C++*, Sams Publishing, Carmel, IN, 1992.
18. K. Weiskamp, B. Flaming, *The Complete C++ Primer*, Academic Press, Inc., San Diego, CA, 1990.
19. J. D. Smith, *Reusability & Software Construction: C & C++*, John Wiley & Sons, Inc., New York, 1990.
20. G. Booch, *Object-Oriented Design with Applications*, The Benjamin/Cummings Publishing Company. Redwood City, CA, 1991.
21. B. Meyer, *Object-Oriented Software Construction*, Pretice Hall International (U.K.) Ltd. Hertfordshire, Marea Britanie, 1988.

22. L. Pinson, R. S. Wiener, *Applications of Object-Oriented Programming*, Addison-Wesley Publishing Company, Reading, MA, 1990.
23. J. Rambaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modelling and Design*, Prentice Hall, Englewood-Cliffs, NJ, 1991.
24. L. A. Winblad, S. D. Ewards, D. R. King, *Object-Oriented Software*, Addison-Wesley Publishing Company, Reading, MA, 1990.
25. R. Wirfs-Brock, B. Wilkerson, L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ, 1990.
26. D. Claude, *Programmer en Turbo C++*, Eyrolles, Paris, 1991.