

## 1. Să ne reamintim ... (noțiuni de programare obiectuală)

În anii '60 s-au dezvoltat tehnicile de programare structurată. Conform celebrei ecuații a lui Nicklaus Wirth:

$$\text{algoritmi} + \text{structuri de date} = \text{programe}$$

un program este format din două părți total separate :

- un ansamblu de proceduri și funcții;
- un ansamblu de date asupra cărora acționează practic;

Procedurile sunt prezente ca și cutii negre, fiecare având de rezolvat o anumită sarcină (de făcut anumite prelucrări). Această modalitate de programare se numește *programare structurată*. Evoluția calculatoarelor și a problemelor de programare, a făcut ca în aproximativ 10 ani, programarea structurată să devină inefficientă.

Astfel, într-un program bine structurat în proceduri, este posibil ca o schimbare relativ minoră în structura datelor, să provoace o deranjare majoră a procedurilor.

Programarea structurată este tehnica pe care ați abordat-o, începând de la conceperea primei organigrame și terminând cu cele mai complexe programe pe care le-ați elaborat până în acest moment, presupunând bineînțeles că, citind aceste rânduri, nu sunteți încă fani ai programării obiectuale.

Scopul acestui capitol este de a vă deprinde cu o tehnică nouă, mult mai “tânără” decât programarea structurată, tehnică ce permite o concepție mult mai apropiată de natural a programelor. Aceasta este **Programarea Orientată pe Obiecte (POO)**.

Dar înainte de a începe să ridicăm ceața ce înconjoară această tehnică nouă, să facem cunoștință cu un nou mediu de programare, care va fi utilizat de acum înainte pentru implementarea programelor. Acest mediu este Microsoft Visual C++ 6.0.

### 1.1. Un mediu de programare nou? Oare cum arată?

Visual C++ 6.0 face parte dintr-un ansamblu de programe, numit Visual Studio 6.0. Acesta este implementarea firmei Microsoft și pentru alte limbaje pe lângă C++, cum ar fi FoxPro, Basic, Java, etc. Utilizarea acestui mediu de programare are mai multe avantaje:

- posibilitatea de creare de programe în diferite limbaje, care să *interacționeze* între ele;
- utilizarea unor obiecte *native* Windows, ceea ce duce la crearea de programe executabile de dimensiuni relativ mici;
- posibilitatea utilizării de biblioteci complexe, ce pun la dispoziția programatorului metode de rezolvare a mării majorități a problemelor;
- posibilitatea utilizării de programe vrăjitor (Wizard) care ghidează programatorul în implementarea programului.

Cum pornim Visual C++? Dacă aveți mediul de programare instalat pe calculatorul Dvs., veți putea lansa mediul de programare de la **Start->Programs->Microsoft Visual Studio 6.0->Microsoft Visual C++ 6.0** (fig. 1.1);

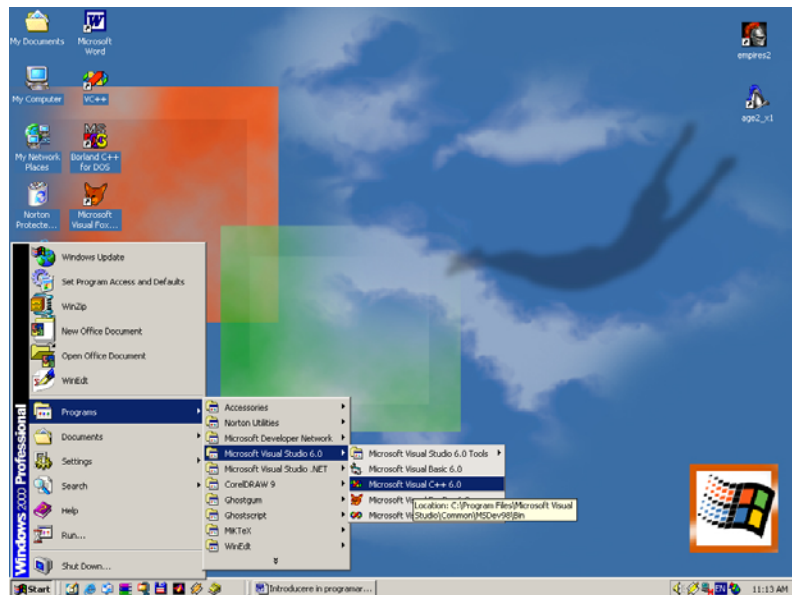


Fig. 1.1 Lansarea în execuție a Visual C++ 6.0

O dată lansat programul, acesta se prezintă și vă afișează o casetă cu diferite “șmecherii” utile în scrierea programelor sau utilizarea mediului (fig. 1.2). Ca să puteți lucra, va trebui (eventual după ce citiți o serie de astfel de informații) să închideți caseta apăsând butonul **OK**.

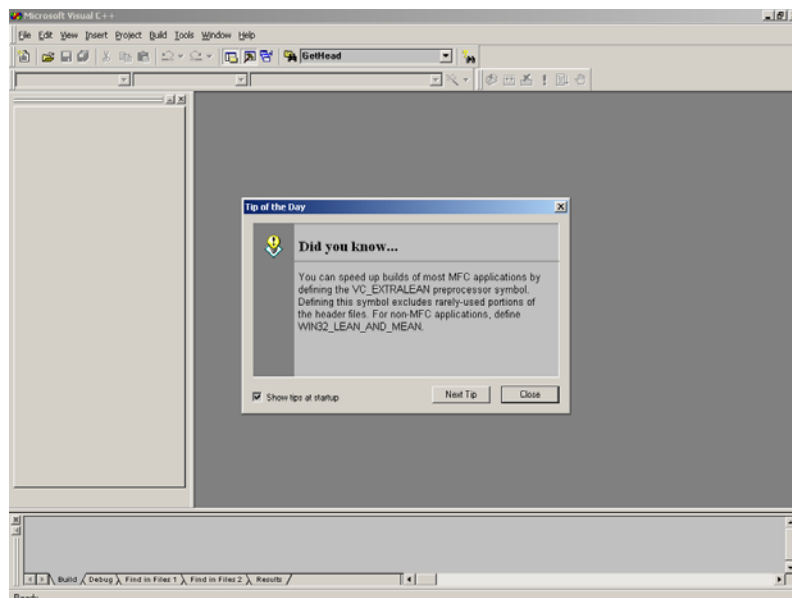


Fig. 1.2. Așa începe totul ...

În Visual C++, orice program executabil rezultă în urma compilării și editării legăturilor în cadrul unui *Proiect* (Project). Un proiect este o entitate constituită din mai multe fișiere header, sursă, de resurse, etc, care conțin toate informațiile necesare generării programului executabil. Acesta, va rezulta ca un fișier cu același nume cu numele proiectului.

O altă noțiune nouă introdusă de mediul de programare este *Spațiul de lucru* (Workspace). Acesta este constituit din totalitatea fișierelor, utilităților, dispozitivelor, etc, puse la dispoziția programatorului în cadrul ferestrei Visual C++, pentru a-l ajuta la crearea programului. O să constatați că, dacă ați lucrat la un program și ați închis mediul de lucru, la revenirea în același program totul va fi identic cu momentul în care l-ați închis. Aceasta deoarece spațiul de lucru este salvat într-un fișier (cu extensia *.dsw*) și informațiile din acesta reconstituie în mod identic spațiul de lucru la orice nouă accesare. În mod normal, fiecare spațiu de lucru are în interiorul lui un singur proiect, dar se pot adăuga mai multe proiecte la același spațiu de lucru.

Cum deschidem deci un proiect nou? Foarte simplu: în meniul **File** alegeți opțiunea **New** și veți fi invitați să alegeți tipul de proiect dorit (fig. 1.3).

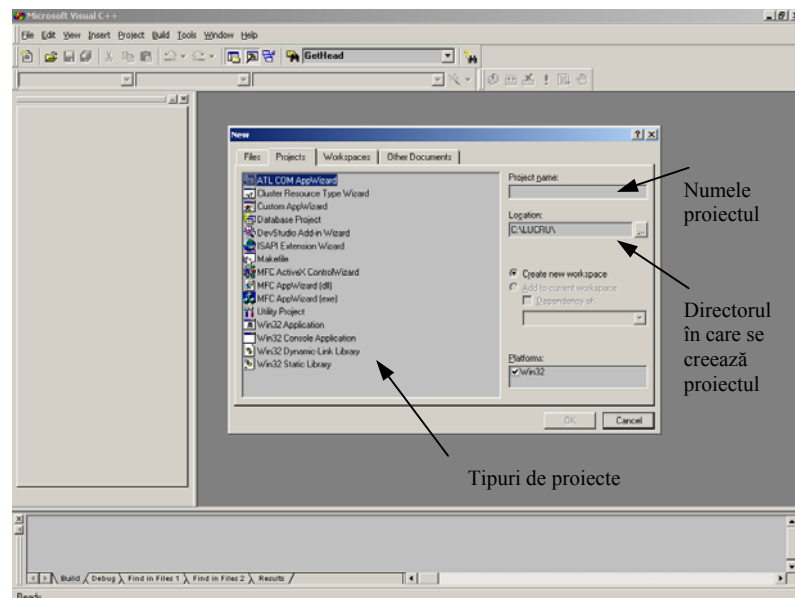


Fig. 1.3. Crearea unui nou proiect

După cum se poate vedea, există mai multe tipuri de proiecte ce pot fi create. Ne vom referi în acest moment la două, pe care le vom utiliza în capitolele ce urmează.

### 1.1.1 Win32 Console Application. Țsta nu e DOS?

Vom încerca întâi să creem un proiect consolă. Asta înseamnă că vom obține un program care să ruleze într-un ecran MsDos, cu toate că nu se respectă regulile de compunere a adreselor din acest bătrân sistem de operare, adică ceea ce știm: pointeri pe 16 biți. Acum pointerii sunt pe 32 de biți, dar îi vom folosi la fel ca înainte. Pentru aceasta, vom alege opțiunea **Win32 Console Application**, și haideți să scriem în caseta **Project name** numele proiectului: *Primul*.

O dată ales numele proiectului și apăsând pe butonul **OK**, mediul de programare va dori să știe cum trebuie să creeze proiectul: să ne lase pe noi să luăm totul de la 0, sau să ne ofere niște șabloane de programare, cu anumite biblioteci gata incluse (fig. 1.4). Vom alege opțiunea **An empty project**, adică va trebui să populăm noi proiectul cu diferite fișiere înainte de a putea compila ceva.

Apoi apăsăm butonul **Finish** pentru a încheia procesul de generare a proiectului. Înainte de final, mediul ne va mai afișa o pagină de informații, în care ne va spune ce

am lucrat până în acest moment, adică aproape nimic. După apăsarea butonului **OK**, proiectul este gata să fie utilizat.

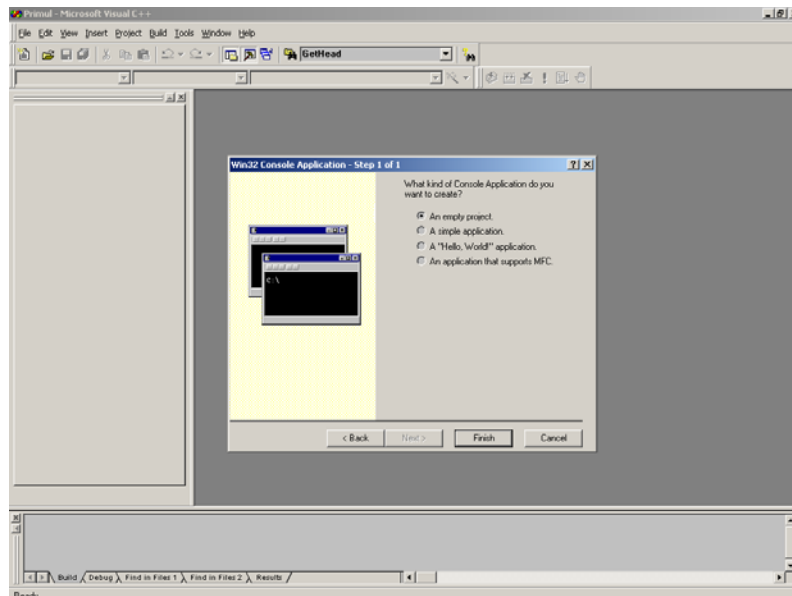


Fig. 1.4. Creem un proiect gol. O să avem ceva de lucru...

Dacă o să vă uitați în acest moment la structura de directoare, veți constata că în directorul ales de Dvs. în caseta **Location**, a fost creat un director cu numele *Primul*, care conține mai multe fișiere asociate proiectului. Nu veți găsi nici un fișier sursă (.cpp) sau header (.h). Acestea trebuie inserate în proiect.

Pentru aceasta, din nou în meniul **File** vom alege opțiunea **New**. De această dată, mediul ne va permite să populăm proiectul cu diferite fișiere. Vom alege opțiunea **C++ Source File** și vom da numele fișierului sursă tot *Primul* (fig. 1.5). Trebuie să avem grijă ca opțiunea **Add to project**: să fie validată, în caz contrar fișierul nou creat nu va fi adăugat proiectului și va trebui să-l adăugăm manual.

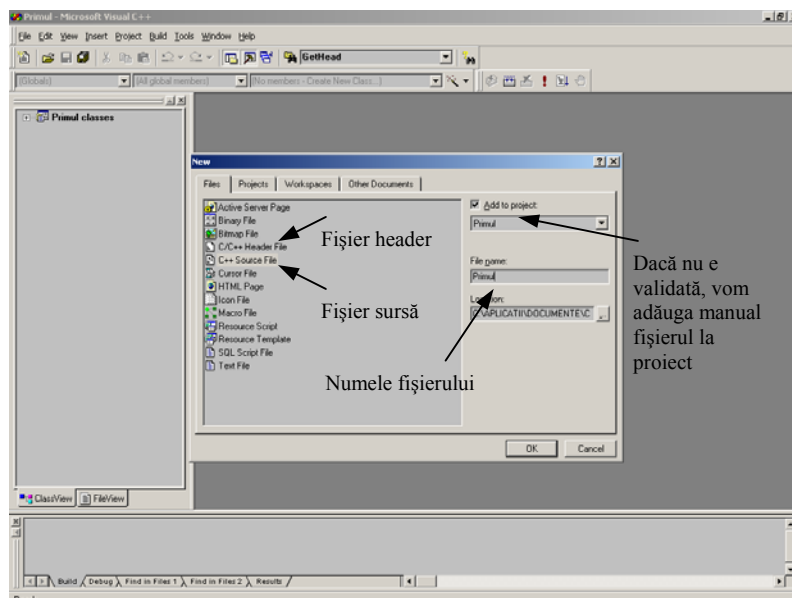


Fig. 1.5. Vom adăuga și un fișier sursă

Gata! Dacă apăsăm **OK**, fișierul *Primul.cpp* e adăugat directorului *Primul* și așteaptă să fie completat cu codul sursă.

Haideți să scriem următorul program:

```
#include <iostream.h>

void main()
{
    char nume[20];
    char prop1[]="\n Salut ";
    char prop2[]="\n Ai scris primul program VC++!\n\n";

    cout << "\n Cum te numesti : " ;
    cin >> nume;
    cout << prop1 << nume << prop2;
}
```

E un program care nu diferă cu nimic de ce știm până acum. Va trebui să-l compilăm. Pentru acesta, vom alege în meniu opțiunea **Build->Rebuild All** (fig. 1.6), iar dacă nu avem erori (ar trebui să nu avem!) vom lansa programul în execuție apăsând pe butonul **!**.

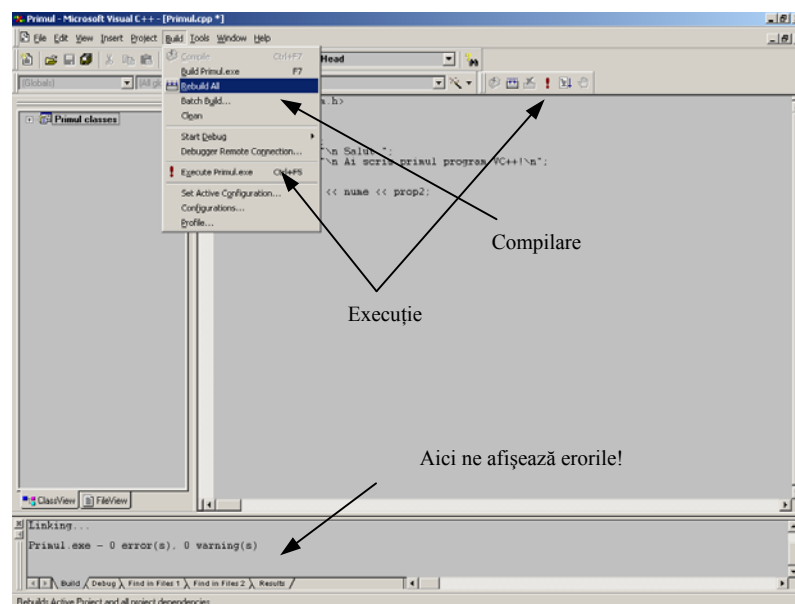


Fig. 1. 6. Compilăm și lansăm în execuție

Execuția programului se face într-o fereastră DOS ce arată ca în fig. 1.7.

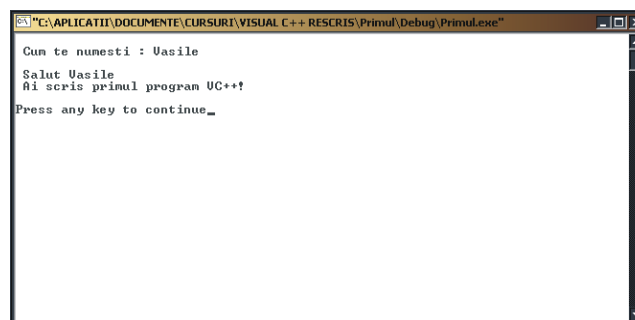


Fig. 1. 7. Așa arată un program în consolă

### 1.1.2 MFC Application Wizard. E altceva!

Vom implementa acum un program Windows, fără să intrăm în amănunte cu privire la tehnologia folosită. Pentru aceasta, să închidem proiectul *Primul* (**File->Close Workspace** și apoi **OK** la întrebarea dacă să se închidă toate documentele) și să creăm un nou proiect, pe care să-l numim *Al doilea*. De data aceasta, vom alege un proiect de tip **MFC AppWizard (exe)**. După apăsarea butonului **OK**, vom putea alege unul din cele 3 șabloane de programe Windows pe care le putem utiliza. Vom alege **Dialog based** (fig. 1.8) și vom apăsa **Finish** și apoi **OK** în pagina de prezentare.

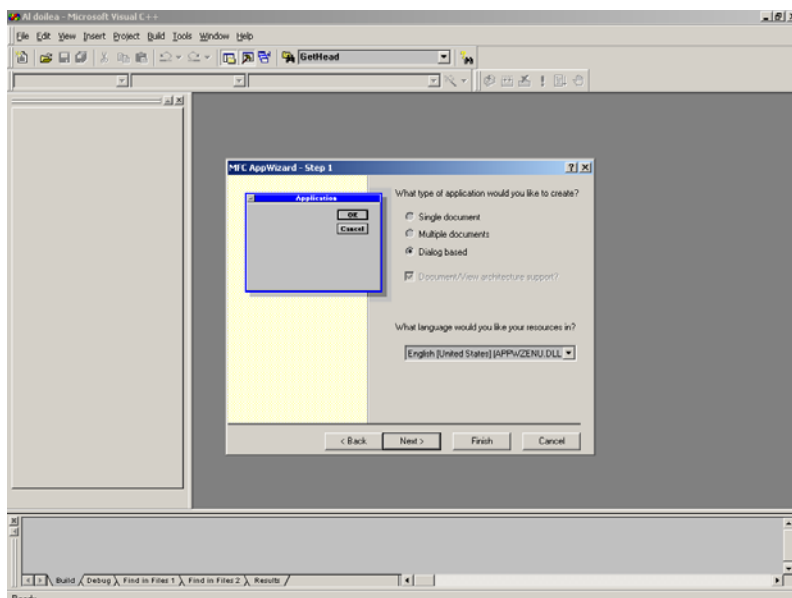


Fig. 1.8. Crearea unui proiect de tip dialog

Un astfel de proiect ne va pune la dispoziție o machetă, pe care putem construi cu ajutorul controalelor din bara de instrumente, interfața utilizator dorită.

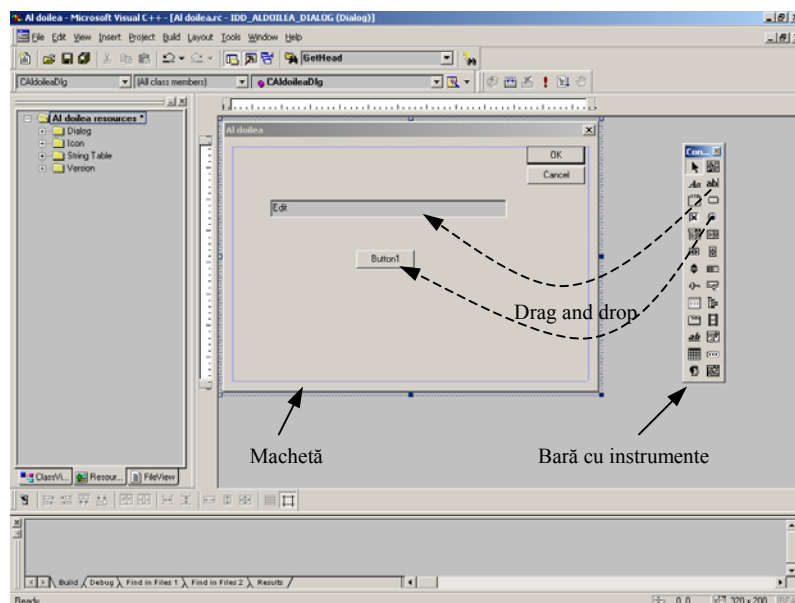


Fig. 1.9. Să compunem interfața

Pentru interfața programului nostru, vom șterge textul "TODO: Place dialog controls here" (facem click cu tasta dreaptă a mouse-ului și apoi apăsăm tasta

*Delete*) și vom insera un buton de comandă (*Button*) și o casetă de editare (*Edit Box*) din bara de instrumente (fig. 1.9). Inserarea se face ușor, suntem deja familiarizați cu tehnica *drag and drop* caracteristică sistemelor Windows.

Acum, apăsăm dublu-click pe butonul `Button1` și acceptăm numele `OnButton1` propus de mediul de programare (adică apăsăm butonul **OK**). Mediul de programare ne va poziționa într-o funcție, cu implementarea de mai jos:

```
void CAlDoileaDlg::OnButton1()
{
    // TODO: Add your control notification handler code here
}
```

În această funcție, vom adăuga codul nostru:

```
void CAlDoileaDlg::OnButton1()
{
    // TODO: Add your control notification handler code here
    CString nume, afisare;
    GetDlgItem(IDC_EDIT1) -> GetWindowText(nume);
    afisare += " Salut "+nume+
        "\n Acesta este primul tau program Windows!";
    AfxMessageBox(afisare);
}
```

Să facem o convenție. Tot codul ce trebuie adăugat de noi, va fi scris de acum înainte cu caractere bold. Acum vom compila și lansa în execuție programul (similar proiectului **Win32 Console Application**), și va fi afișată macheta din fig. 1.10. În această machetă, vom utiliza caseta de editare pentru introducerea numelui, iar la apăsarea butonului, va fi afișat mesajul din fig. 1.11.

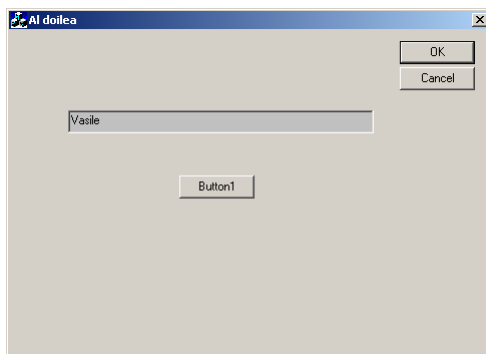


Fig. 1.10. Așa arată macheta

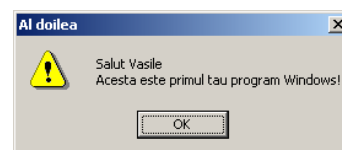


Fig. 1.11. Mesajul afișat

Asta este! Am implementat și primul program Windows! De fapt, am dorit doar să facem cunoștință cu mediul de programare. Cu tipul de proiect de mai sus ne vom întâlni abia mai târziu, în capitolele următoare. Deocamdată, ne vom mulțumi cu proiecte **Win32 ConsoleApplication**.

## 1.2 Să revenim la oile noastre... adică, POO!

De ce programare obiectuală și nu programare structurată? Să încercăm să înțelegem dintr-un exemplu simplu.

Să presupunem că într-un program, avem de manipulat ca informație puncte în plan. Ar trebui deci să declarăm o structură de forma:

```
struct punct_plan {
    int coordx;
    int coordy;
};
punct_plan punct1;
```

Să presupunem că logica programului cere ca toate punctele utilizate să fie de ordonată pozitivă, deci deasupra axei reale (fig. 1.12).

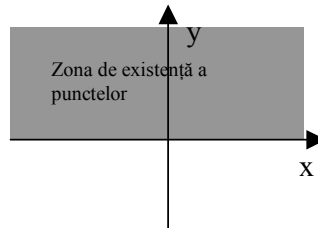


Fig. 1.12. Aici avem puncte...

Cu alte cuvinte, pentru orice punct introdus în program, ordonata va trebui înlocuită cu valoarea ei absolută. Va trebui să scriem de fiecare dată, o secvență de forma (evident, fără să uităm să includem bibliotecile *iostream.h* și *math.h*)

```
cin >> punct1.coordx >> punct1.coordy;
punct1.coordy= abs(punct1.coordy);
```

Dar cine garantează că a doua linie de program este introdusă întotdeauna? Poate că o soluție mai bună ar fi citirea ordonatei prin intermediul unei funcții, care să returneze întotdeauna valoarea ei absolută. Vom putea avea spre exemplu în program declarată funcția

```
int citescoordy ()
{
    int inputy;
    cin >> inputy;
    return (abs(inputy));
}
```

iar secvența de program de citire a punctului ar putea fi

```
cin >> punct1.coordx ;
punct1.coordy=citescoordy();
```

Dar, din nou, cine garantează că undeva în program nu se strecoară și una din liniile

```
cin >> punct1.coordy;
```

sau

```
punct1.coordy=7;
```



Nu avem în acest moment la îndemână nici o tehnică prin care să fim obligați să folosim doar funcția de citire pentru atribuirea de valori ordonatei unui punct, sau să ne oblige să atribuim doar valori pozitive acesteia.

### 1.2.1 Ce este o clasă? Ce este un obiect?

Din cele arătate mai sus, apare ideea introducerii unui nou „tip”, care să încapsuleze o structură de date și un set de funcții de interfață care acționează asupra datelor din structură. În plus, noul tip trebuie să asigure diferite niveluri de acces la date, astfel încât anumite date să nu poată fi accesate decât prin intermediul unor funcții de interfață și nu în mod direct. Acest nou tip este denumit **clasă**.

În limbajul C++ clasele se obțin prin completarea structurilor uzuale din limbajul C, cu setul de funcții necesare implementării interfeței obiectului. Aceste funcții poartă denumirea de **metode**.

Pentru realizarea izolării reprezentării interne de restul programului, fiecărui membru (dată din cadrul structurii, sau metodă) i se asociază nivelul de încapsulare **public** sau **private** (fig. 1.13).

Un membru **public** corespunde din punct de vedere al nivelului de accesibilitate, membrilor structurilor din limbajul C. El poate fi accesat din orice punct al programului, fără să se impună vre-o restricție asupra lui.

Membrii **private** sunt accesibili doar *în domeniul clasei*, adică în clasa propriu-zisă și în toate funcțiile membre.

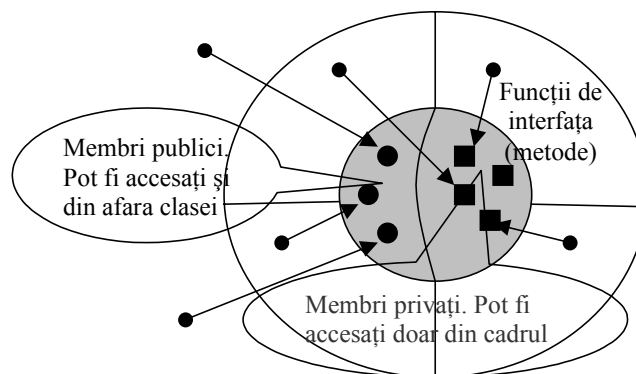


Fig. 1.13. Accesibilitatea membrilor clasei

Sintaxa folosită pentru declararea unei clase este următoarea:

```
class Nume_clasa {
[ [private :] lista_membri_1 ]
[ [public :] lista_membri_2 ]
};
```

Cuvântul cheie **class** indică faptul că urmează descrierea unei clase, având numele *Nume\_clasa*. Numele clasei poate fi orice identificator (orice nume valid de variabilă), dar trebuie să fie unic în cadrul domeniului de existență respectiv.

Descrierea clasei constă din cele două liste de membri, prefixate eventual de cuvintele cheie **private** și **public**.

**Observație:** Dacă în declarația unei clase apare o listă de membri fără nici un specificator de acces, acești membri vor fi **implicit privați**.

În exemplu nostru, este evident că va trebui să declarăm o clasă care să conțină ca și date două valori întregi. Deoarece ordonata poate fi doar pozitivă, ea nu trebuie să poată fi accesată direct din program, ci doar prin intermediul unei metode care să îi atribuie o valoare, dar numai pozitivă. Deci, va trebui să o declarăm ca membru privat. Cum ordonata nu poate fi accesată direct din program, va trebui să adăugăm clasei și o metodă care să permită citirea valorii ei.

Uzual, declararea unei clase se face într-un fișier header. Nu este obligatoriu, dar o să vedem în capitolele următoare că acest mod de implementare a programului duce la o mai mare claritate a acestuia.

Acestea fiind spuse, să ne facem curaj și să declarăm prima noastră clasă. Vom deschide un proiect nou, de tip **Win32 Console Application**. Haideți să-l numim *Prima\_Clasa*. Acestui proiect îi vom adăuga în fișierul header *Prima\_Clasa.h*, în care vom declara clasa astfel:

// fișierul *Prima\_Clasa.h*

```
class punct_plan
{
    int coordy;
public:
    int coordx;
    void setcoordy(int cy) {coordy=abs(cy);};
    int getcoordy() {return coordy;};
};
```

Am declarat astfel clasă pe care o vom utiliza în continuare. Se poate observa că variabila `coordy` este privată, deci va putea fi accesată în afara clasei, doar prin intermediul metodelor puse la dispoziție de clasă. În cadrul metodelor, deci din interiorul clasei, variabila poate fi accesată în mod direct.

Va trebui acum să declarăm niște “variabile” de tipul clasei. O astfel de variabilă poartă denumirea de **obiect** și reprezintă în fapt o instanțiere (concretizare) a clasei respective. Putem da acum și o definiție pentru clasă:

**Definiția 1.1:** *O clasă este un tip de date care descrie un ansamblu de obiecte cu aceeași structură și același comportament.*

Obiectele de tipul clasei, le vom declara în fișierul sursă *Prima\_Clasa.cpp*, pe care îl vom adăuga proiectului (Ați uitat cum? Simplu: **File ->New-> C++ Source File**), în care vom scrie instrucțiunile de mai jos:

// fișierul *Prima\_Clasa.cpp*

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

void main()
{
    punct_plan punct, *ppunct;
    int valy;

    cout << "\n Introduceți abscisa : ";
    cin >> punct.coordx;
```

```

cout << "\n Introduceți ordonata : ";
cin >> valy;
punct.setcoordy(valy);
cout << "\n Valorile pentru abscisa si ordonata sunt "
    << punct.coordx << " si " << punct.getcoordy();
ppunct=&punct;
cout << "\n Acum cu pointer "
    << ppunct->coordx << " si " << ppunct->getcoordy() << "\n\n";
}

```

Vom compila și executa programul obținut. Rezultatul execuției programului este prezentat în fig. 1.14.

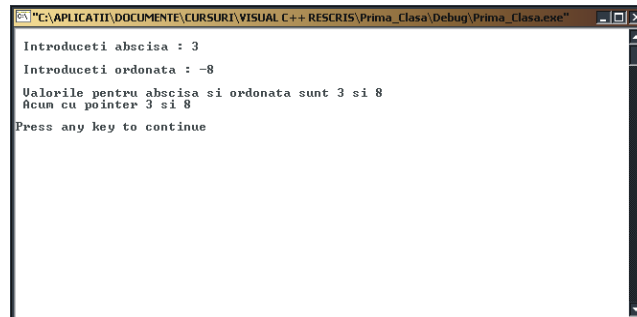


Fig. 1.14. Asta am obținut!

Ce observații putem face dacă ne uităm la program:

- în primul rând, membrul privat al clasei poate fi accesat din `main()` (adică din afara clasei) doar prin intermediul metodelor clasei. Dacă am fi scris o instrucțiune de forma

```
cin >> punct.coordy;
```

am fi obținut o eroare încă din faza de compilare. Cu alte cuvinte, problema noastră, de a forța valori pozitive pentru ordonată e rezolvată!

- și în cazul obiectelor, ca și în cazul variabilelor de un tip standard sau utilizator, se poate face atribuirea

```
ppunct=&punct
```

unde prin `&punct` înțelegem (din nou) adresa obiectului `punct`. Deci, formalismul lucrului cu pointeri este neschimbat;

- accesul la componentele unui obiect, fie elemente ale structurii de date, fie metode, se face ca și în cazul structurilor. Putem observa că și în cazul obiectului `punct` și în cazul pointerului `ppunct`, accesul se face în forma deja cunoscută;

Un membru privat al unei clase, poate fi totuși accesat și din afara clasei. Dar funcția care-l accesează trebuie să fie o funcție *prietenă*. O funcție este prezentată clasei ca fiind *prietenă* prin intermediul cuvântului rezervat `friend`:

```

class punct_plan
{
    friend void functie_prietena(punct_plan);
    int coordy;
public:

```

```

int coordx;
void setcoordy(int cy){coordy=abs(cy);};
int getcoordy() {return coordy;};
};

```

Funcția, cu toate că nu aparține clasei, ci este o funcție globală, va putea accesa direct variabila privată din clasă:

```

#include <iostream.h>
#include <math.h>
#include "Prima_Clasa.h"

void functie_prietena(punct_plan pct)
{
    cout << "\n coordy " << pct.coordy;
}

void main()
{
    ...
    cout << "\n Acum cu pointer "
        << ppunct->coordx << " si " << ppunct->getcoordy() << "\n\n";
    functie_prietena(punct);
}

```

Să revenim puțin la mediul de programare. În partea stângă, se poate observa o fereastră ce conține 2 etichete: **Class View** și respectiv **File View**. În fereastra **Class View** (fig. 1.15), mediul ne afișează toate componentele programului: clasa punct\_plan, marcată cu o pictogramă de tip arbore, metodele setcoordy() și getcoordy() marcate prin paralelipede de culoare roșie, variabilele coordy și coordx, marcate prin paralelipede albastre, precum și mărimile globale, în cazul nostru, doar funcția main(). Putem observa de asemenea, că în dreptul variabilei coordy este desenat un lacăt, marcând faptul că aceasta este o mărime privată. Un dublu click asupra oricărei componente, va afișa și fixa prompterul în zona de implementare corespunzătoare în fereastra programului.

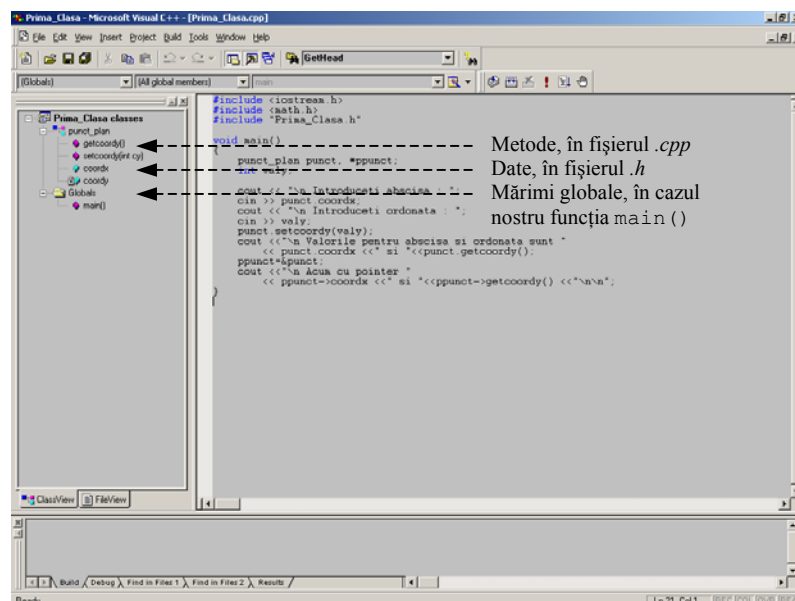


Fig. 1.15. Mărimile din program sunt figurate în Class View!

Pentru eticheta **File View** (fig. 1.16) fereastra din stânga va conține toate fișierele care compun proiectul, în funcție de tipul lor. Din nou, un dublu click asupra unui nume de fișier, va face ca în fereastra programului să se afișeze conținutul fișierului respectiv.

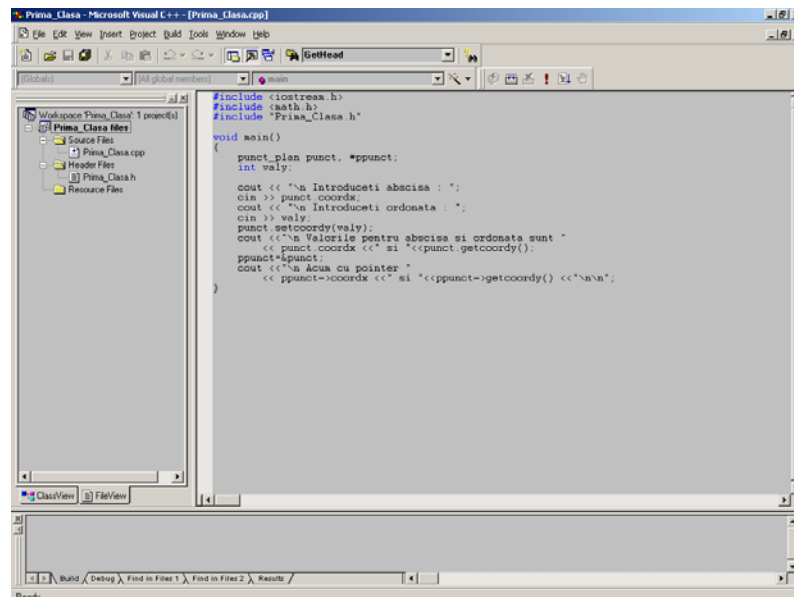


Fig. 1.16 . Așa arată File View

### 1.2.2 Funcții inline. La ce or fi bune?

Să modificăm declarația clasei `punct_plan` ca și în codul de mai jos:

```
class punct_plan
{
    int coordy;
public:
    int coordx;
    inline void setcoordy(int cy) {coordy=abs(cy);};
    inline int getcoordy() {return coordy;};
};
```

Ce am modificat? Am introdus cuvintele `inline` în fața definirii metodelor, transformându-le astfel în funcții **inline**. Dacă compilăm și executăm programul, constatăm că nimic nu s-a schimbat. Atunci, ce este de fapt o funcție inline?

Să ne reamintim care este mecanismul care se pune în mișcare, atunci când într-un program se face un apel de funcție (fig. 1.17):

- la întâlnirea unui apel de funcție, se salvează în stivă adresa din memorie a codului următoarei instrucțiuni executabile, precum și valorile parametrilor funcției;
- se sare din secvența normală de instrucțiuni și se execută prima instrucțiune din funcție, aflată la o adresă cunoscută din memorie;
- se execută toate instrucțiunile funcției, iar la sfârșit se extrage din stivă adresa următoarei instrucțiuni executabile din programul apelant;
- se continuă execuția normală a programului.

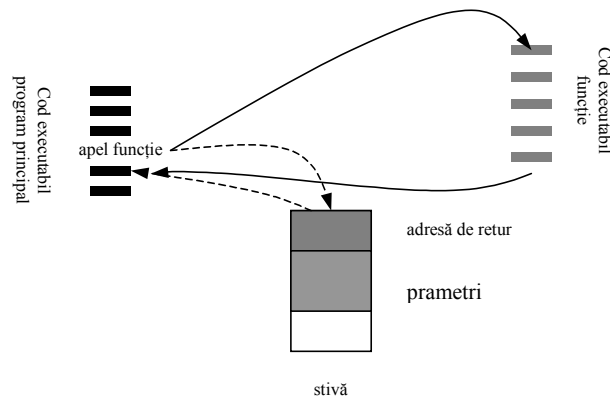


Fig. 1.17. Așa se pelează o funcție

Acest mecanism asigură o dimensiune redusă a codului executabil, pentru că toate codurile executabile asociate funcțiilor vor apare o singură dată în codul programului. Dar, fiecare apel de funcție înseamnă respectarea mecanismului descris mai sus. Fiecare operație durează un interval de timp, timp care poate fi chiar mai mare decât timpul de execuție al codului funcției apelate, dacă acesta este scurt.

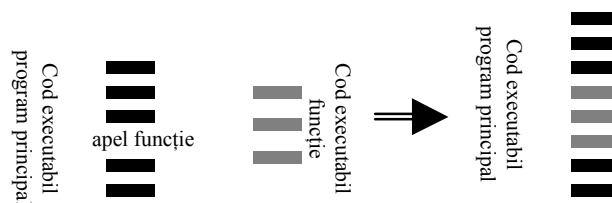


Fig. 1.18. Funcții inline

În cazul funcțiilor cu puține instrucțiuni, este uneori utilă să le declarăm **inline**. În acest caz, nu se mai generează un apel normal al funcției, cu tot mecanismul aferent, ci pur și simplu, codul funcției este inserat în locul în care a fost apelată (fig. 1.18). Se obține astfel un cod executabil mai lung, dar timpul de execuție al programului este scurtat. Declararea unei funcții inline este absolut la latitudinea noastră. Depinde dacă urmărim un cod executabil mai redus ca dimensiune, sau un timp de execuție mai scurt.

### 1.2.3. Nume calificat. Operator de domeniu

Metodele clasei `punct_plan` au instrucțiuni foarte puține, deci nu a fost o problemă să le implementăm în momentul declarării și chiar să le definim `inline`. Dar, în marea majoritate a cazurilor, în fișierele header se face doar declararea metodelor, iar implementarea lor este făcută în fișierele `.cpp`, având astfel loc o separare clară a implementării unei clase de interfața ei.

Pentru a respecta cele arătate mai sus, să rescriem conținutul fișierului *Prima\_Clasa.h* ca mai jos:

```
class punct_plan
{
    int coordy;
public:
    int coordx;
```

```

    void setcoordy(int cy);
    int getcoordy();
};

```

Cum definirea metodelor nu este făcută, acestea vor trebui implementate în fișierul *Prima\_Clasa.cpp*. Să modificăm acest fișier ca mai jos:

```

#include <iostream.h>
#include <math.h>
#include "Prima_Clasa.h"

void setcoordy(int cy)
{
    coordy=abs(cy);
}

int getcoordy()
{
    return(coordy);
}

void main()
{
    punct_plan punct, *ppunct;
    int valy;

    cout << "\n Introduceți abscisa : ";
    cin >> punct.coordx;
    cout << "\n Introduceți ordonata : ";
    cin >> valy;
    punct.setcoordy(valy);
    cout << "\n Valorile pentru abscisa si ordonata sunt "
        << punct.coordx << " si " << punct.getcoordy();
    ppunct=&punct;
    cout << "\n Acum cu pointer "
        << ppunct->coordx << " si " << ppunct->getcoordy() << "\n\n";
}

```

Aparent totul e corect. Dacă însă vom compila programul, vom obține erori. Compilatorul, în cazul nostru, nu va ști cine este variabila `coordy`. Aceasta se întâmplă pentru că, din modul de definire, funcțiile `getcoordy()` și `setcoordy()` sunt interpretate de compilator ca și funcții globale (fig. 1.19) și nu aparținând clasei `punct_plan`, iar variabila `coordy` nu este declarată nicăieri în `main()`. Ea este o proprietate intrinsecă a clasei `punct_plan`, iar compilatorul nu are de unde să știe că cele două funcții sunt metode ale clasei. De altfel, am putea avea declarate mai multe clase, fiecare conținând câte o metodă `setcoordy()` și respectiv `getcoordy()`, având implementări complet diferite. De unde știm că o funcție este declarată într-o clasă sau în alta?

Pentru clarificarea problemei, va trebui ca în momentul definirii unei funcții, pe lângă numele ei, să precizăm compilatorului și clasa din care face parte aceasta. Prin adăugarea numelui clasei la numele funcției se obține **numele complet** (sau **numele calificat**) al funcției. Adăugarea numelui clasei se face cu ajutorul operatorului `::`, numit **operator de domeniu**. Deci, de exemplu, numele calificat al funcției `getcoordy()` va fi `punct_plan::getcoordy()`. Astfel, funcții cu același nume, dar aparținând la clase diferite, vor fi identificate corect de compilator.

Pentru ca programul nostru să nu mai aibă erori de compilare, vom modifica fișierul *Prima\_Clasa.cpp* ca mai jos:

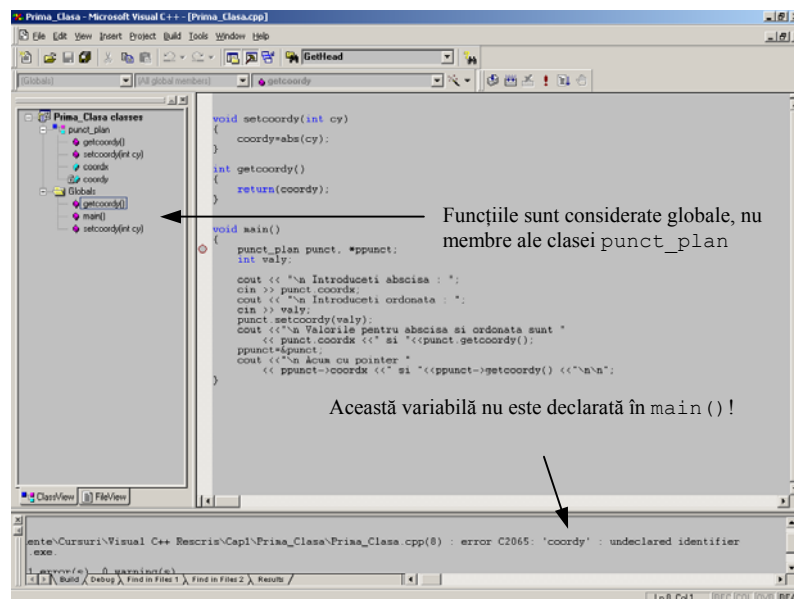


Fig. 1.19. Funcțiile sunt considerate globale!

```

#include <iostream.h>
#include <math.h>
#include "Prima_Clasa.h"

void punct_plan::setcoordy(int cy)
{
    coordy=abs(cy);
}

int punct_plan::getcoordy()
{
    return(coordy);
}

void main()
{
    ...
}
  
```

#### 1.2.4. Pointerul ascuns this. Uf, ce încurcătură

Haideți să modificăm din nou fișierul *Prima\_Clasa.cpp*, astfel încât să avem două obiecte de clasă *punct\_plan*, respectiv *punct1* și *punct2*:

```

#include <iostream.h>
#include <math.h>
#include "Prima_Clasa.h"

void punct_plan::setcoordy(int cy)
{
    coordy=abs(cy);
}

int punct_plan::getcoordy()
{
    return(coordy);
}
  
```



```

}

void main()
{
    punct_plan punct1, punct2;
    int valy;

    cout << "\n Introduceți abscisa 1: ";
    cin >> punct1.coordx;
    cout << "\n Introduceți ordonata 1: ";
    cin >> valy;
    punct1.setcoordy(valy);
    cout << "\n Introduceți abscisa 2: ";
    cin >> punct2.coordx;
    cout << "\n Introduceți ordonata 2: ";
    cin >> valy;
    punct2.setcoordy(valy);

    cout << "\n Valorile pentru abscisa1 si ordonata1 sunt "
        << punct1.coordx << " si " << punct1.getcoordy()
        << "\n iar pentru abscisa2 si ordonata2 sunt "
        << punct2.coordx << " si " << punct2.getcoordy() << "\n\n";
}

```

Dacă executăm programul, vom observa că funcția `setcoordy()` modifică valoarea `punct1.coordy` când este apelată de `punct1` și respectiv `punct2.coordy` când este apelată de `punct2`. Similar și funcția `getcoordy()` returnează corect valorile variabilelor `coordy` pentru cele 2 obiecte. Dar, dacă ne uităm la implementarea funcțiilor, fiecare dintre ele manipulează o variabilă întreagă `coordy`, fără a se face vre-o referire la obiectul căreia îi aparține acea mărime. De unde știe totuși programul să atribuie în mod corect mărimea `coordy` unui obiect sau altuia?

Pentru a identifica obiectul implicit asupra căruia se operează, compilatorul generează un pointer ascuns, numit `this`, care se încarcă înaintea oricărei operații cu adresa obiectului curent. Codul corect în accepțiunea compilatorului este de fapt:

```

void punct_plan::setcoordy(int cy)
{
    this->coordy=abs(cy);
}
int punct_plan::getcoordy()
{
    return(this->coordy);
}

```

astfel programul putând ști exact variabila de la ce adresă de memorie să o modifice.

Este foarte simplu să verificăm faptul că acest pointer se încarcă cu adresa obiectului curent. Este suficient pentru aceasta, să modificăm programul astfel încât, în funcția `setcoordy()` de exemplu, să afișăm adresa conținută de `this`, iar în programul principal adresele obiectelor utilizate pentru apelul funcției.

```

#include <iostream.h>
#include <math.h>
#include "Prima_Clasa.h"

void punct_plan::setcoordy(int cy)
{
    this->coordy=abs(cy);
}

```

```

        cout << " this= " << hex << this << "\n\n";
    }

int punct_plan::getcoordy()
{
    return(this->coordy);
}

void main()
{
    punct_plan punct1, punct2;
    cout << "\n &punct1= " << hex << &punct1
        << "\n punct1.setcoordy() ";
    punct1.setcoordy(10);
    cout << "\n &punct2= " << hex << &punct2
        << "\n punct2.setcoordy() ";
    punct2.setcoordy(10);
}

```

Dacă executăm acest program, vom obține rezultatul din fig. 1.20.

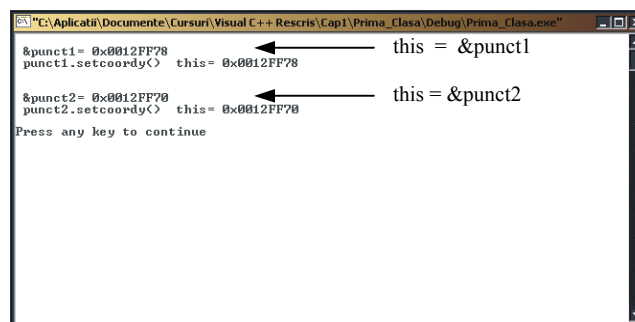


Fig. 1.20. Așa lucrează de fapt compilatorul...

Ce observăm? Că la apelul `punct1.setcoordy()`, `this` se încarcă cu adresa obiectului `punct1`, iar la apelul `punct2.setcoordy()`, `this` se încarcă cu adresa obiectului `punct2`. Apoi, funcția va modifica valoarea câmpului `coordy` a obiectului pointat de `this`.

### 1.2.5 Membri statici. Ce mai sunt și ăștia?

Să modificăm declarația clasei `punct_plan` ca mai jos:

```

class punct_plan
{
    int coordy;
public:
    static int coordx;
    void setcoordy(int cy);
    int getcoordy();
    void inccoordy();
    void inccoordx();
};

```

Am declarat două noi funcții, care urmează să fie definite, dar lucru nou, am adăugat cuvântul rezervat `static` în fața declarării câmpului `coordx`. Astfel, `coordx` devine o variabilă statică. O variabilă statică este un atribut propriu și nu fiecărui obiect

în parte. Dacă pentru o variabilă obișnuită, se rezervă câte o zonă de dimensiunea `sizeof()` în cazul declarării fiecărui obiect, o variabilă statică are o unică zonă de rezervare. Ea apare ca și o zonă de memorie comună tuturor obiectelor (fig. 1.22). Ea există și dacă nu a fost declarat nici un obiect din clasa respectivă!

O variabilă statică **trebuie neapărat inițializată** și nu este vizibilă decât în interiorul fișierului în care a fost declarată.

Vom completa fișierul *Prima\_Clasa.cpp* ca mai jos:

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

void punct_plan::setcoordy(int cy)
{
    this->coordy=abs(cy);
    cout << " this= " << hex << this << "\n\n";
}
int punct_plan::getcoordy()
{
    return(this->coordy);
}

void punct_plan::inccoordx()
{
    coordx++;
    cout << " coordx= " << dec << coordx;
}
void punct_plan::inccoordy()
{
    coordy++;
    cout << " coordy= " << dec << coordy;
}

int punct_plan::coordx=10; // am inițializat variabila statică!

void main()
{
    cout << "\n Variabila statica : " << punct_plan::coordx << "\n";
    punct_plan punct1, punct2;

    cout << "\n &punct1= " << hex << &punct1
        << "\n punct1.setcoordy() ";
    punct1.setcoordy(10);
    cout << "\n &punct2= " << hex << &punct2
        << "\n punct2.setcoordy() ";
    punct2.setcoordy(10);
    cout << "\n Pentru punct1 avem : ";
    punct1.inccoordx();
    punct1.inccoordy();
    cout << "\n Pentru punct2 avem : ";
    punct2.inccoordx();
    punct2.inccoordy();
    cout << "\n\n";
}
```

Ce observăm? Funcțiile `inccoordx()` și `inccoordy()` nu fac altceva decât să incrementeze câmpul corespunzător al structurii de date. De asemenea, se observă că

variabila statică este inițializată înainte de folosire și mai mult, valoarea ei poate fi afișată înainte de declararea unui obiect al clasei. Rezultatul execuției programului este prezentat în fig. 1.21.

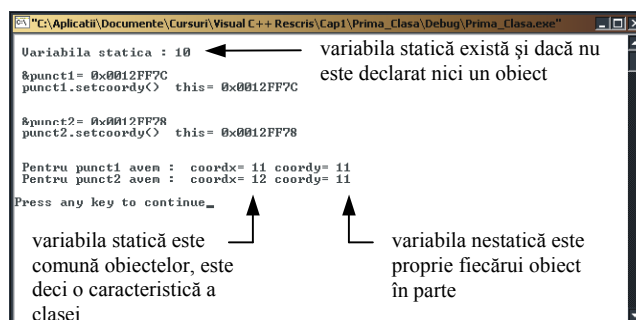


Fig. 1.21. Variabilele statice sunt atribute ale clasei

De ce `coordx` ia valoarea 12 după operațiile de incrementare? Pentru că ea este un atribut al clasei! Variabila statică va fi incrementată o dată prin apelarea funcției de incrementare prin `punct1` și o dată prin `punct2`. Variabila nestatică este, după cum se vede un atribut al obiectului, ea este instanțiată și incrementată separat pentru fiecare obiect în parte (fig. 1.22).

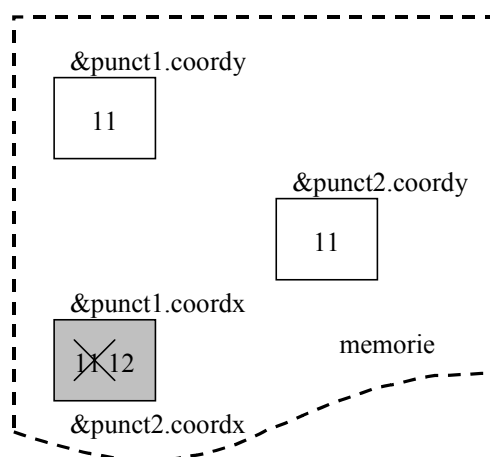


Fig. 1.22 Variabila statică este comună obiectelor

Acum, să mai facem o modificare în fișierul de declarare al clasei. Să declarăm funcția `inccoordx()` ca funcție statică:

```
class punct_plan
{
    int coordy;
public:
    static int coordx;
    void setcoordy(int cy);
    int getcoordy();
    static void inccoordy();
    void inccoordx();
};
```

Dacă compilăm programul, vom obține erori de compilare (fig. 1.23).

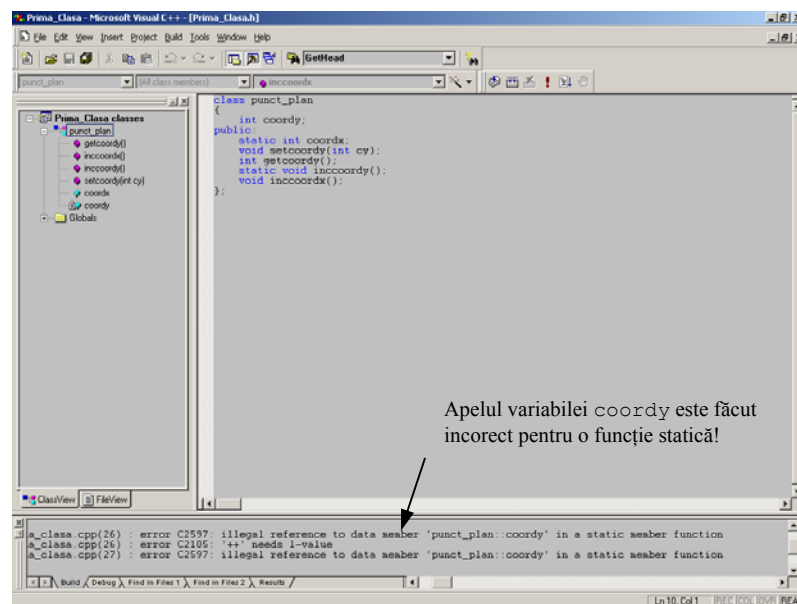


Fig. 1.22. Funcția statică nu poate accesa coordy. Nu există this!

Eroarea se datorează faptului că, o funcție statică, fiind de asemenea un atribut al clasei, nu poate instanția corect variabila `coordy`. Pur și simplu nu știe cărui obiect îi aparține, deoarece funcțiile statice nu primesc ca argument pointerul `this`, la fel ca și funcțiile nestatice. Va trebui ca funcției statice să-i transmitem explicit un pointer spre obiectul curent:

```
class punct_plan
{
    int coordy;
public:
    ...
    static void inccoordy(punct_plan* ptr);
    void inccoordx();
};
```

Modificările în fișorul sursă vor fi:

```
#include <iostream.h>
...
void punct_plan::inccoordy(punct_plan* ptr)
{
    ptr->coordy++;
    cout << " coordy= " << dec << ptr->coordy;
}

int punct_plan::coordx=10;

void main()
{
    ...
    punct_plan punct1, punct2, *pointer;
    ...
    punct1.inccoordx();
    pointer=&punct1;
    punct1.inccoordy(pointer);
}
```

```

cout <<" \n Pentru punct2 avem : ";
punct2.inccoordx();
pointer=&punct2;
punct2.inccoordy(pointer);
cout << "\n\n";
}

```

### 1.2.6 Constructori, destructori ...

Am învățat că la declararea unei variabile de un anumit tip, se rezervă în memorie o zonă de dimensiune `sizeof(tip)` pentru stocarea valorilor variabilei. Și în cazul obiectelor trebuie rezervată o zonă de memorie pentru stocarea valorilor structurii de date ce compune clasa. Această rezervare este făcută de o funcție specială, numită **constructor**. Constructorul unei clase este generat implicit de compilator, dar poate fi redefinit de programator.

Un constructor este *o funcție care are același nume ca și clasa din care face parte*. Spre deosebire de o funcție obișnuită, un constructor nu returnează nici o valoare. Rolul constructorului este de a inițializa un obiect.

O clasă poate avea mai mulți constructori, care diferă între ei prin semnătură (lista parametrilor formali). Ca și în cazul unei funcții obișnuite, unii parametri pot fi specificați implicit. **Un constructor, trebuie să fie întotdeauna public, în caz contrar la declararea unui obiect în afara clasei, el nu va fi accesibil!**

O problemă importantă apare în cazul claselor care conțin membri de tip pointer, pentru care trebuie alocată dinamic memorie. Această operațiune este efectuată de obicei în constructorul clasei. Alocarea dinamică a memoriei se poate face folosind funcția `malloc`, totuși limbajul C++ aduce o nouă facilitate în acest sens: operatorul `new`.

Să luăm un exemplu: să deschidem un nou proiect de tip **Win32 Console Application**, numit *Constructori* și să-l populăm cu fișiere cu același nume.

În fișierul header vom declara clasa `elem_lista`, ca mai jos:

```

class elem_lista
{
    int val;
    int* urmatorul;
public:
    elem_lista(int);
    elem_lista(int,int);
    void afisez();
};

```

Ce am declarat? O clasă care are ca structură de date un element de tip listă simplu înlănțuită. Clasa declară de asemenea doi constructori, care pot fi deosebiți prin lista de argumente și o funcție de afișare.

Să implementăm acum conținutul fișierului sursă.

```

#include <iostream.h>
#include "Constructori.h"

inline elem_lista::elem_lista(int valoare)
{
    val=valoare;
    urmatorul=NULL;
}

```

```

}
inline elem_lista::elem_lista(int valoare1, int valoare2)
{
    val=valoare1;
    urmatorul=new int(valoare2);
}
void elem_lista::afisez()
{
    if (urmatorul)
        cout << "\n val= " << val << " *urmatorul= " << *urmatorul
            << " urmatorul= " << hex << urmatorul;
    else
        cout << "\n val= " << val << " urmatorul= "
            << hex << urmatorul;
}

void main()
{
    elem_lista element1(5);
    element1.afisez();
    elem_lista element2(7,9);
    element2.afisez();
    elem_lista *pelement=new elem_lista(3,5);
    pelement->afisez();
    cout << "\n\n";
}

```

Cei doi constructori creează structuri de date ca în fig. 1.23.

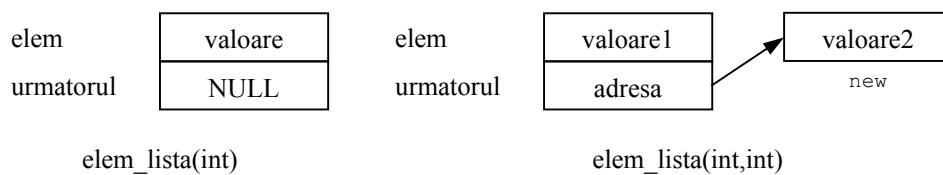


Figura 1.23. Constructorii creează structuri de date diferite

Structura `if - else` din funcția de listare este necesară, deoarece o încercare de afișare a conținutului adresei 0 (`p=NULL`) duce în Windows la violarea protecției și în consecință, terminarea anormală a programului.

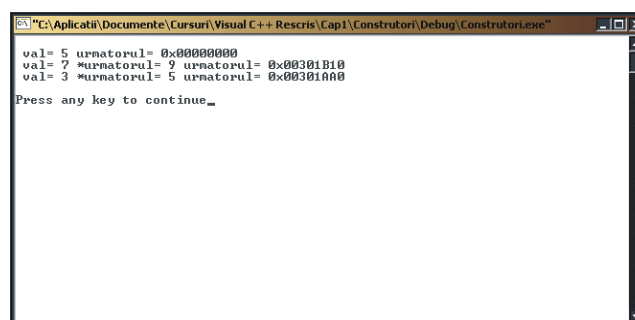


Figura 1.24. Rezultatul execuției programului

În fig. 1.24 putem vedea rezultatul execuției programului. Putem observa că apelul primului constructor completează doar structura declarată în clasă, iar apelul celui de al doilea constructor, fie la construirea unui obiect al clasei, fie la construirea unui pointer la clasă, alocă în mod dinamic memorie pentru cel de-al doilea întreg. În acest

caz, va trebui ca la terminarea domeniului de existență al obiectului, zona de memorie alocată dinamic să fie eliberată. Acest lucru se face de o altă funcție specială, numită **destructor**.

O clasă poate declara destructorul, care este apelat automat de compilator în momentul distrugerii obiectelor din acea clasă. Funcția destructor nu returnează nici o valoare, iar numele ei este format cu construcția `~nume_clasă`. ***Destructorul nu primește nici un parametru. O clasă poate avea un sigur destructor.*** De obicei, rolul destructorului este de a dealoca memoria alocată dinamic în constructor.

Pentru eliberarea memoriei alocate, se poate folosi în continuare funcția `free`, dar se recomandă folosirea operatorului `delete`.

Să modificăm declarația clase ca mai jos:

```
class elem_lista
{
    int val;
    int* urmatorul;
public:
    elem_lista(int);
    elem_lista(int,int);
    ~elem_lista();
    void afisez();
};
```

Funcția nou adăugată este destructorul clasei. Implementarea lui va fi:

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

...
inline elem_lista::elem_lista(int valoare1, int valoare2)
{
    val=valoare1;
    urmatorul=new int(valoare2);
}

inline elem_lista::~elem_lista()
{
    if (urmatorul != NULL) delete urmatorul;
}

...
}
```

Destructorul va fi apelat automat la terminarea programului

Am învățat că în C++ o variabilă poate fi inițializată la declarare. În mod absolut similar și un obiect poate fi inițializat la declarare. Putem scrie de exemplu, în cazul clasei declarate în primul program,

```
punct_plan punct1;
punct_plan punct2=punct1;
```

În acest caz, cel de-al doilea obiect al clasei va fi construit în mod identic cu primul. Pentru construirea obiectului `punct2`, compilatorul apelează (și creează implicit) un nou constructor, numit **constructor de copiere**. Constructorul de copiere nu face altceva decât să rezerve memorie pentru structura de date al celui de-al doilea



obiect și apoi să copieze bit cu bit valorile din câmpurile primului obiect în câmpurile celui de al doilea.

Dacă vom modifica însă fișierul *Constructori.h* astfel încât structura de date a clasei să fie publică (pentru a putea afișa direct în programul principal valorile câmpurilor):

```
class elem_lista
{
public:
    int val;
    int* urmatorul;
    elem_lista(int);
    elem_lista(int,int);
    ~elem_lista();
    void afisez();
};
```

și apoi vom modifica fișierul *Constructori.cpp* ca mai jos,

```
#include <iostream.h>
...
void main()
{
    elem_lista element1(6,7), element2=element1;
    cout << "\n val1= " << element1.val << " *urmatorul1= "
        << *element1.urmatorul << " urmatorul1= " << hex
        << element1.urmatorul;
    cout << "\n val2= " << element2.val << " *urmatorul2= "
        << *element2.urmatorul << " urmatorul2= " << hex
        << element2.urmatorul;
    cout << "\n\n";
}
```

vom obține eroarea de execuție din fig. 1.25:

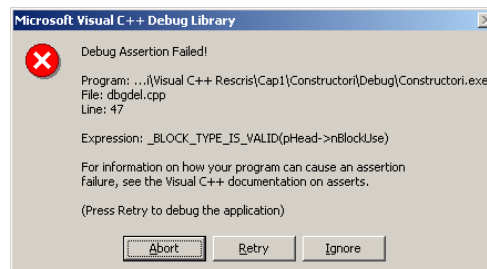


Fig. 1.25. Obținem o eroare de asertie!

De fapt, ce am făcut în program. Am declarat obiectul `element1`, pentru care a fost creată structura de date, conform celui de al doilea constructor. Apoi, la declararea obiectului `element2`, a intrat în funcțiune constructorul de copiere, care construiește cel de-al doilea obiect copiind bit cu bit valorile din structura de date asociată. Adică, pointerul `element2.urmatorul` va fi o copie bit cu bit a pointerului `element1.urmatorul`. Adică, cele două obiecte vor pointa spre aceeași adresă creată dinamic în timpul execuției (fig. 1.26). La terminarea programului (sau a domeniului de vizibilitate a obiectelor), destructorul eliberează zona alocată dinamic și distruge ultimul obiect creat. Deci, în primul obiect, va rămâne un pointer încărcat cu o adresă ce nu mai aparține programului.

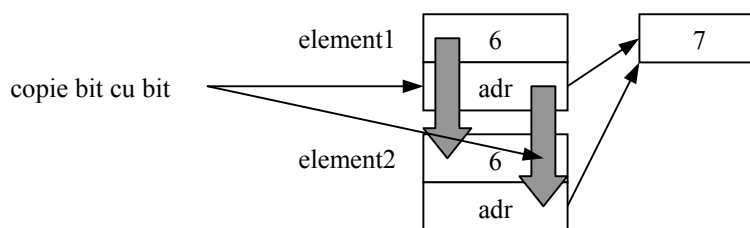


Fig. 1.26. al doile obiect este o copie a primului

***Dacă structura de date a clasei conține pointeri ce alocă dinamic memorie, constructorul de copiere va trebui declarat explicit.*** Constructorul de copiere este o funcție cu același nume cu cel al clasei, care primește ca argument o referință la un obiect de tipul clasei. În cazul nostru, va trebui să declarăm un constructor de copiere:

```
class elem_lista
{
public:
    int val;
    int* urmatorul;
    elem_lista(int);
    elem_lista(int,int);
    elem_lista(elem_lista&);
    ~elem_lista();
    void afisez();
};
```

și să-l implementăm

```
#include <iostream.h>
...
elem_lista::elem_lista(elem_lista& obiectsursa)
{
    this->val=obiectsursa.val;
    this->urmatorul=new int(*obiectsursa.urmatorul);
}
inline elem_lista::~~elem_lista()
{
    if (urmatorul != NULL) delete urmatorul;
}
...
void main()
{
    ...
}
```

Ce face constructorul de copiere în acest caz? În primul rând, primește ca argument o referință spre obiectul sursă. Apoi face atribuire valoarea câmpului `val` din obiectul sursă, câmpului `val` al noului obiect creat. Câmpul `urmatorul` al acestui obiect se încarcă apoi cu adresa unui întreg creat dinamic și în care se depune conținutul adresei pointate de câmpul `urmatorul` din obiectul sursă. Vor rezulta astfel două obiecte care conțin valori identice, dar la adrese diferite, lucru care se vede cu ușurință în fig. 1.27.

Ca o observație, pointerul `this` este scris explicit în acest exemplu, doar din motive didactice, el este oricum creat ascuns de către compilator și fără să fie scris.

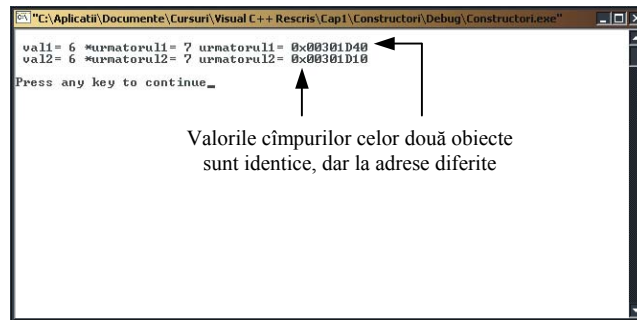


Fig. 1. 27. Constructorul de copiere creează dinamic o nouă adresă

O situație specială apare în cazul obiectelor care au ca membri instanțieri ale unor obiecte de alt tip, rezultând astfel o încuibărire a claselor. În acest caz, regulile sunt:

1. constructorii obiectelor “încuibărite” se apelează înaintea constructorului obiectului “cuib;
2. dacă nu sunt apelați explicit constructorii obiectelor încuibărite, se încearcă apelarea unui constructor cu parametrii luând valori implicite;
3. destructorul obiectului cuib este apelat înaintea destructorilor obiectelor încuibărite.

De exemplu, într-un nou proiect **Win32 Console Application**, numit *Patrat* putem scrie:

```
// Patrat.h
class Punct
{
    int coordx, coordy;
public:
    Punct(int x=0, int y=0);
    ~Punct();
};

class Patrat
{
    Punct st_sus, st_jos, dr_sus, dr_jos;
public:
    Patrat();
    ~Patrat();
};

// Patrat.cpp
#include <iostream.h>
#include "Patrat.h"

Punct::Punct(int x, int y)
{
    coordx=x;
    coordy=y;
    cout << "\n Constructor clasa Punct cu x= "
        << coordx << " y= " << coordy;
}

Punct::~~Punct()
{
    cout << "\n Destructor clasa Punct cu x= "
        << coordx << " y= " << coordy << " ";
}
```

```

}

Patrat::Patrat():st_jos(0, 1), dr_jos(1, 1), dr_sus(1, 0)
{
    cout << "\n Constructor clasa Patrat";
}
Patrat::~Patrat()
{
    cout << "\n Destructor clasa Patrat";
}

void main()
{
    Patrat p;
    cout << "\n";
}

```

Ce putem observa. Înainte de construirea obiectului `Patrat`, se construiesc cele 4 obiecte `Punct` care-l compun conform constructorului. Destructorii sunt apelați în ordine inversă apelului constructorilor.

De observat forma aparte a constructorului clasei `Patrat`. Construcția cu `:` urmat de o listă de apeluri de constructor ale obiectelor membru se numește *listă de inițializare*. **Atenție, instrucțiunile dintr-o listă de inițializare au o ordine de execuție pur aleatoare!** Clasa `Patrat` apelează explicit constructorii pentru obiectele `st_jos`, `dr_jos` și `dr_sus`. Constructorul obiectului `st_sus` este apelat implicit, cu parametri implicați de valoare 0.

### 1.2.7. Redefinirea operatorilor.

O facilitare extrem de puternică a limbajului C++ este dreptul programatorului de a adăuga operatorilor limbajului și alte sensuri decât cele predefinite. Cu alte cuvinte, este vorba despre o redefinire a operatorilor limbajului C++, fără a se pierde vechile sensuri. Termenul consacrat în literatura de specialitate pentru această operațiune este cel de *overloading operators*.

Redefinirea operatorilor (sau supraîncărcarea operatorilor) se poate face în două moduri: fie sub formă de funcții membre, fie ca și funcții `friend`. Redefinirea unui operator presupune declararea unei funcții al cărei nume este format din **cuvântul cheie operator, un spațiu, și simbolul operatorului redefinit**.

La ce e de fapt bună redefinirea operatorilor? Haideți să luăm iar un exemplu: să declarăm o structură care să implementeze un număr complex, de forma **Real+i\*Imaginar**.

```

struct Complex
{
    double Re;
    double Im;
};
Complex nr1, nr2;

```

Să presupunem că am atribuit valori părților reale și imaginare pentru variabilele `nr1` și `nr2` și dorim să efectuăm operația `nr3=nr1+nr2`. Pentru aceasta, fie că declarăm o funcție, fie că facem operația de adunare în programul principal, va trebui să facem adunările separat pentru partea reală și respectiv pentru partea imaginară:

```

Complex Adun(Complex n1, Complex n2)
{
    Complex n3;
    n3.Re=n1.Re+n2.Re;
    n3.Im=n1.Im+n2.Im;
    return n3;
}

...
void main()
{
    Complex nr1, nr2, nr3;
    ...
    nr3=Adun(nr1,nr2);
}

```

O modalitate de implementare a problemei mai apropiată de spiritul C++, este de a redefini operatorii aritmetici, astfel încât, dacă operanzii sunt numere complexe, să se poată scrie direct instrucțiuni de forma  $nr3=nr1+nr2$ ,  $nr3=nr1*nr2$ , etc. Pentru început, să declarăm clasa `Complex` (într-un nou proiect **Win32 Console Application**, numit *Complex*):

```

class Complex
{
    double Re, Im;
public:
    Complex(double Real=0, double Imag=0){Re=Real;Im=Imag;};
    void afisez();
    Complex& operator + (Complex&);
    friend Complex& operator - (Complex&, Complex&);
};

```

Ce conține clasa? Am declarat pentru structura de date doi membri privați, care vor implementa partea reală și partea imaginară a numărului complex. Constructorul primește două argumente de tip `double`, care au valorile implicite 0. De asemenea, clasa declară o funcție de afișare. Ca noutate, apare declararea operatorului `+` ca și funcție membră a clasei și respectiv a operatorului `-`, ca funcție prietenă. Deoarece rezultatul unei operații între două numere complexe este tot un număr complex, cei doi operatori redefiniți vor returna o referință la clasa `Complex`. De fapt, operatorul este interpretat ca o funcție, dar cu un rol special. Operatorul `+` în cazul nostru va fi interpretat pentru secvența  $c=a+b$  ca și  $c=a.functia+(b)$ , iar în cazul `-`,  $c=fuctia-(a,b)$ .

Să implementăm acum fișierul sursă:

```

#include <iostream.h>
#include "Complex.h"

Complex& Complex::operator +(Complex& operand)
{
    return *new Complex(this->Re+operand.Re, this->Im+operand.Im);
}

void Complex::afisez()
{
    if (Im>0)    cout << "\n" << Re << "+"<< Im <<"i";
}

```

```

        else cout <<"\n" << Re << Im <<"i";
    }

Complex& operator - (Complex& desc, Complex& scaz)
{
    return *new Complex(desc.Re-scaz.Re,desc.Im-scaz.Im) ;
}

void main()
{
    Complex nr1(4,5) , nr2(7,8) , nr3;
    nr3=nr1+nr2;
    nr3.afisez() ;
    nr1=nr1-nr3-nr2;
    nr1.afisez() ;
    cout <<"\n\n";
}

```

Ce face operatorul +? Creează un nou complex, a cărui parte reală, respectiv imaginară este suma dintre partea reală (imaginară) a obiectului curent, adică a primului operand și partea reală (imaginară) a obiectului primit ca argument, adică cel de-al doilea operand. Returnează apoi valoarea noului complex.

În mod absolut similar este implementată și funcția prietenă pentru operatorul -.

Unii dintre operatori (operatorul `new`, operatorul `delete`, operatorul `[]`, operatorul `()` și operatorul `=`) necesită precauții suplimentare în cazul redefinirii lor. Dintre aceștia, cel mai des întâlnit este operatorul `=`.

Dacă o clasă nu redefineste operatorul `=`, compilatorul atașează clasei respective un operator `=` implicit, care, să ne reamintim, efectuează o copie bit cu bit a operandului din dreapta în operandul din stânga. Situația este identică cu cea în care se generează un constructor de copiere implicit. Dacă clasa conține membri de tip pointer, se ajunge la situația în care doi pointeri se încarcă cu adresa aceleași zone de memorie, care poate avea efecte dezastruoase dacă se eliberează unul din pointeri și apoi se accesează memoria prin al doilea pointer. De aceea se recomandă redefinirea operatorului `=` doar în cazul claselor care nu au membri de tip pointer.

Spre exemplu, în cazul clasei `elem_lista`, vom putea redefini operatorul `=` ca mai jos:

Va trebui întâi declarat operatorul în zona de declarații publice ale clasei:

```
elem_lista& operator = (elem_lista);
```

apoi va trebui definit:

```

elem_lista& elem_lista::operator =(elem_lista operand)
{
    if (this!=&operand)
    {
        val=operand.val;
        *urmatorul=*operand.urmatorul;
    }
    return *this;
}

```

Ce facem de fapt? Întâi ne uităm dacă `this` nu conține tocmai adresa operandului transmis ca argument, adică dacă nu suntem în cazul `element2=element1`. În acest caz, nu trebuie să facem nimic, decât să returnăm valoarea de la acea adresă. Dacă

operandii sunt diferiți, vom încărca la adresa `this` conținutul de la adresa pointată de pointerul operandului din dreapta. Astfel, vom avea din nou, adrese diferite, dar încărcate cu aceeași valoare.

### 1.3 Moștenirea

Să mergem mai departe și să înțelegem un concept nou, care dă adevărata “putere” programării obiectuale. Pentru acesata, să deschidem un proiect nou, pe care să-l numim sper exemplu *Mostenire*. În fișierul header, să declarăm din nou clasa `punct_plan`, ca mai jos, adăugând și atreia dimensiune (ce înseamnă un membru `protected` vom vedea imediat):

```
class punct_plan
{
public:
    int coordx;
private:
    int coordy;
protected:
    int coordz;
public:
    void setcoordy(int cy);
    int getcoordy();
    void setcoordz(int cz);
    int getcoordz();
};
```

Fișierul sursă va fi completat ca mai jos:

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

void punct_plan::setcoordy(int cy)
{
    coordy=cy;
}
int punct_plan::getcoordy()
{
    return coordy;
}
void punct_plan::setcoordz(int cz)
{
    coordz=cz;
}
int punct_plan::getcoordz()
{
    return coordz;
}

void main()
{
    punct_plan punct1;

    punct1.coordx=5;
    punct1.setcoordy(10);
    punct1.setcoordz(3);
}
```

```

    cout << "\n punct1= (" << punct1.coordx << " , "
        << punct1.getcoordy() << " , " << punct1.getcoordz() << ")";
    cout << "\n\n";
}

```

Dacă ne uităm în **ClassView**, vom observa că în dreptul variabilei `coordz` este figurată o cheie. Deci, variabila protejată nu are indicația nici a variabilei publice, nici a celei private. Totuși, în program, nu am fi putut să o accesăm direct și a fost nevoie de implementarea metodelor de interfață `setcoordz()` și `getcoordz()`. **Deci, față de programul principal, sau mai bine zis, față de mediul din afara clasei, variabila protejată are statut de variabilă privată.**

Să declarăm acum o nouă clasă, numită `punct_colorat`, care pe lângă cele trei coordonate ale clasei `punct_plan`, va mai avea un atribut și anume, un cod de culoare. Am putea declara noua clasă ca mai jos (tot în fișierul *Mostenire.h*):

```

class punct_colorat
{
public:
    int coordx;
private:
    int coordy;
protected:
    int coordz;
    int culoare;
public:
    void setcoordy(int cy);
    int getcoordy();
    void setcoordz(int cz);
    int getcoordz();
    void setculoare(int cul);
    int getculoare();
};

```

Este totuși neperformant să declarăm astfel cea de a doua clasă, deoarece ea repetă identic informațiile din prima clasă, adăugând ceva în plus. În C++, se poate stabili o ierarhie de clase, astfel încât acestea să se afle într-o relație de moștenire. O clasă poate fi **derivată** din altă clasă, moștenindu-i atributele și metodele, putând adăuga în plus altele noi. Clasa de la care se moștenește se numește **clasă de bază**, sau **superclasă**, iar clasa care se derivează este **clasă derivată** sau **subclasă**.

O subclasă poate fi derivată din clasa de bază în mod **public** sau **privat** (în funcție de *specificatorul de acces* folosit: `public` sau `private`). În baza celor arătate aici, rezultă că o sintaxă mai rafinată pentru declararea unei clase este următoarea:

```

class Nume_Clasa [:public/private Nume_Clasa_De_Baza]
{
[private: lista membri privati]
[public: lista membri publici]
};

```

Modul normal de declarare a clasei `punct_colorat` este:

```

class punct_colorat: public punct_plan
{
    int culoare;
public:

```



```

    void setculoare(int cul);
    int getculoare();
};

```

Clasa `punct_colorat` este **derivată public** din clasa `punct_plan`. Va amosteni toate atributele (cele 3 coordonate) și toate funcțiile de interfață pe care le are și superclasa, dar va adăuga un nou atribut (culoare) și două noi metode.

Vom completa acum fișierul sursă cu definițiile celor două metode noi.

```

#include <iostream.h>
#include "Mostenire.h"
...
int punct_plan::getcoordz()
{
    return coordz;
}
void punct_colorat::setculoare(int cul)
{
    coordz=coordx;
    culoare=cul;
}
int punct_colorat::getculoare()
{
    return culoare;
}

void main()
{
    punct_plan punct1;
    punct_colorat punctc1;
    ...

    punctc1.coordx=5;
    punctc1.setcoordy(10);
    punctc1.setculoare(4);
    cout <<"\n punctc1= (" << punctc1.coordx << " , "
        << punctc1.getcoordy() << " , " << punctc1.getcoordz() <<")";
    cout << "    Culoare= " << punctc1.getculoare();
    cout << "\n\n";
}

```

Ce observăm? Că funcția `setculoare()`, în afară de faptul că atribuie o valoare membrului privat `culoare`, atribuie membrului protejat `coordz` moștenit din clasa de bază valoarea membrului public `coordx` moștenit de asemenea. Deci, ***un membru protejat al clasei de bază este accesibil ca și un membru public dintr-o clasă derivată.***

Am fi putut declara clasa `punct_colorat` ca mai jos:

```

class punct_colorat: private punct_plan
{
    int culoare;
public:
    void setculoare(int cul);
    int getculoare();
};

```

În acest caz, clasa `punct_colorat` este derivată privat din clasa `punct_plan`. Dacă vom compila în acest caz programul, vom observa o mulțime de erori, datorate faptului că toți membrii publici moșteniți din clasa de bază, accesați prin intermediul obiectului de clasă derivată privat se comportă ca și cum ar fi fost privați în clasa de bază. Aceasta este deosebirea dintre moștenirea publică și cea privată: ***un obiect al unei clase derivate public păstrează tipul de acces al membrilor moșteniți din clasa de bază în mod identic. Un obiect al unei clase derivate privat, transformă toți membrii moșteniți din clasa de bază în membrii privați.*** Acest fapt este sintetizat în tabelul 1.1, care prezintă posibilitatea accesului direct al unui membru al clasei derivate:

Tabelul 1.1

Drept de acces în clasa de bază	Specificator de acces (tip moștenire)	Acces în clasa derivată	Acces în afara claselor de bază și derivată
public	public	accesibil	accesibil
protected		accesibil	inaccesibil
private		inaccesibil	inaccesibil
public	private	accesibil	inaccesibil
protected		accesibil	inaccesibil
private		inaccesibil	inaccesibil

### 1.3.1 Constructorii și destructorii claselor aflate în relația de moștenire

Este interesent de văzut carei este ordinea și modalitățile de apel a constructorilor și destructorilor în cazul moștenirii. Regula este următoarea:

- în cazul constructorilor, se apelează mai întâi constructorul clasei de bază, și apoi constructorul clasei derivate. Apelarea constructorului clasei de bază se face implicit, dacă este posibil și dacă constructorul clasei de bază nu este apelat explicit în clasa derivată;
- în cazul destructorilor, se apelează mai întâi destructorul clasei derivate, și apoi destructorul clasei de bază;

Pentru a lămuri această problemă, să modificăm programul astfel încât să definim explicit constructorii și destructorii, iar în programul principal să declarăm două obiecte:

```
class punct_plan
{
...
public:
    punct_plan(){ cout << "\n Constructor punct_plan ";};
    ~punct_plan(){ cout << "\n Destructor punct_plan ";};
    void setcoordy(int cy);
    ...
};

class punct_colorat: public punct_plan
{
    int culoare;
public:
    punct_colorat(){ cout << "\n Constructor punct_colorat ";};
    ~punct_colorat(){ cout << "\n Destructor punct_colorat ";};
    ...
};
```

respectiv

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

...
void main()
{
    punct_plan punct1;
    punct_colorat punctc1;
}
```

Rezultatul programului este prezentat în fig. 1.28.

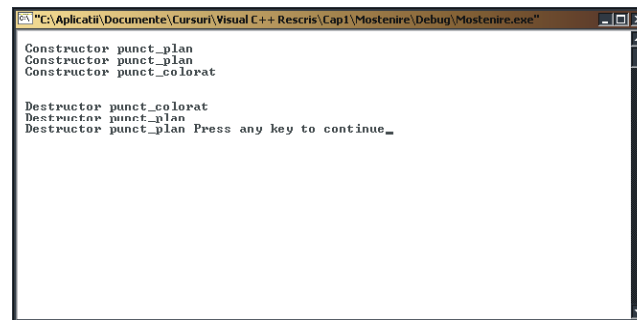


Fig. 1.28. Așa se apelează constructorii și destructorii

Ce observăm? La declararea obiectului `punct1`, se apelează constructorul clasei `punct_plan`. Apoi, la declararea obiectului `punctc1`, se apelează întâi constructorul clasei de bază și apoi constructorul clasei derivate. La distrugere, destructorii se apelează învers.

### 1.3.2 Pointeri. Când facem conversii explicite de tip?

Să declarăm acum 2 pointeri (în programul sursă):

```
void main()
{
    punct_plan punct1, *ppunct1;
    punct_colorat punctc1, *ppunctc1;
    ...
}
```

Vom putea face direct conversii între clasa de bază și subclasă, sau va trebui să facem o conversie explicită de tip?

Dacă `punct_colorat` este derivată public, putem scrie:

```
ppunct1=&punct1; // evident, sunt de același tip
ppunctc1=&punctc1;
ppunct1=&punctc1;
ppunctc1=(punct_colorat*) &punct1;
```

Dacă `punct_colorat` este derivată privat, putem scrie:

```
ppunct1=&punct1; // evident, sunt de același tip
ppunctc1=&punctc1;
```

```
ppunct1=(punct_plan*)&punctc1;
ppunctc1=(punct_colorat*)&punct1;
```

În concluzie, orice conversie de la superclasă la subclasă trebuie făcută în mod explicit. În cazul conversiei de la subclasă la subclasă superclasă, avem cazurile:

- implicit dacă moștenirea este publică;
- explicit, dacă moștenirea este privată;

Faptul că printr-un pointer la clasa de bază putem accesa direct un obiect al unei clase derivate public, duce la niște consecințe extrem de interesante.

### 1.3.3 Tablouri eterogene. Funcții virtuale

Până în acest moment, am fost obișnuiți ca tablourile să conțină elemente de același tip. Aceste tablouri se numesc *tablouri omogene*.

O observație importantă care se impune este aceea că un pointer la o clasă de bază, poate păstra adresa oricărei instanțieri a unei clase derivate public. Așadar, având un șir de pointeri la obiecte de clasă de bază, înseamnă că unii dintre acești pointeri pot referi de fapt obiecte de clase derivate public din aceasta, adică tabloul de pointeri este neomogen. Un astfel de tablou neomogen se numește *eterogen*.

Limbajul C++ aduce un mecanism extrem de puternic de tratare de o manieră uniformă a tablourilor eterogene: funcțiile virtuale.

O *funcție virtuală* este o funcție care este prefixată de cuvântul cheie *virtual*, atunci când este declarată în clasa de bază. Această funcție este redeclarată în clasa derivată (cu aceeași semnătură, adică aceeași listă de parametri formali, același nume și același tip returnat), și prefixată de cuvântul cheie *virtual*.

Să presupunem că un obiect instanțiat dintr-o clasă D (care este derivată public din superclasa B) este accesat folosind un pointer la un obiect de tip B. Să mai facem presupunerea că această clasă B declară o metodă virtuală M, care este apoi redeclarată în clasa D. Atunci când se încearcă apelarea metodei M, folosind pointerul la un obiect de tip B, compilatorul va lua decizia corectă și va apela metoda virtuală M redeclarată în clasa D. Dacă metoda nu este declarată virtuală, la apelul metodei prin pointerul la clasa B, va fi apelată metoda clasei de bază.

Comportamentul diferit al unei funcții cu același nume pentru obiecte din superclasă, respectiv din clasele derivate, se numește *polimorfism*. În concluzie, în C++ polimorfismul este implementat prin funcții virtuale.

Să verificăm această problemă într-un nou proiect, pe care să-l numim *Virtual*.

```
class ObGrafic
{
public:
    ObGrafic();
    ~ObGrafic();
    virtual void Desenez();
};

class Cerc: public ObGrafic
{
public:
    Cerc();
    ~Cerc();
    virtual void Desenez();
};
```

```

class Patrat: public ObGrafic
{
public:
    Patrat();
    ~Patrat();
    virtual void Desenez();
};

```

Am declarat superclasa `ObiectGrafic`, din care am derivat public clasele `Patrat` și `Cerc`. Fiecare clasă implementează un constructor și un destructor, pentru a putea vizualiza modul de construcție și distrugere a obiectelor și o funcție `Desenez()`, declarată ca și virtuală. Implementarea fișierului sursă este:

```

#include <iostream.h>
#include "Virtual.h"

ObGrafic::ObGrafic(){cout << "\n Constructor ObiectGrafic";}

ObGrafic::~~ObGrafic(){cout << "\n Destructor ObiectGrafic";}

void ObGrafic::Desenez(){cout << "\n Desenez un ObiectGrafic";}

Cerc::Cerc(){cout << "\n Constructor Cerc";}

Cerc::~~Cerc(){cout << "\n Destructor Cerc";}

void Cerc::Desenez(){cout << "\n Desenez un Cerc";}

Patrat::Patrat(){cout << "\n Constructor Patrat";}

Patrat::~~Patrat(){cout << "\n Destructor Patrat";}

void Patrat::Desenez(){cout << "\n Desenez un Patrat";}

void main()
{
    ObGrafic* ptab[3];
    ptab[0] = new ObGrafic();
    ptab[1] = new Cerc(); // conversie implicita!
    ptab[2] = new Patrat; // din nou conversie implicita!

    // acum ptab este un tablou neomogen...
    for(int i=0; i<3; i++)
        ptab[i]->Desenez();
    // care este tratat într-o maniera uniforma, datorita mecanismului
    // functiilor virtuale
    for(i=0; i<3; i++)
        delete ptab[i]; // eliberare memorie
}

```

Am creat tabloul `ptab[3]` de pointeri la clasa `ObGrafic`. Deoarece primul element pointează spre un `ObGrafic`, al doilea spre un `Cerc` și al treilea spre un `Patrat`, este un tablou neomogen.

Există totuși o observație legată de programul de mai sus: dacă este rulat, se poate observa că memoria nu se eliberează corect! Este apelat de trei ori destructorul clasei `ObGrafic`, dar nu se apelează nicăieri destructorii claselor derivate! Această problemă

apare datorită faptului că destructorii nu au fost declarați virtuali. Un pointer la clasa de bază, va apela doar destructorul clasei de bază. Problema se rezolvă folosind tot mecanismul funcțiilor virtuale și declarând destructorii claselor ca fiind virtuali.

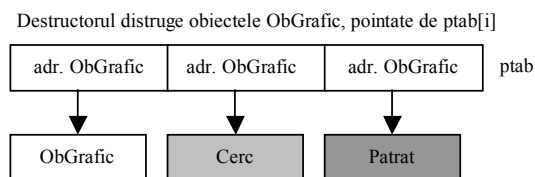


Figura 1.28. Eliberarea incorectă a memoriei

Așadar, codul corectat este:

```
class ObGrafic
{
public:
    ObGrafic();
    virtual ~ObGrafic();
    virtual void Desenez();
};

class Cerc: public ObGrafic
{
public:
    Cerc();
    virtual ~Cerc();
    virtual void Desenez();
};

class Patrat: public ObGrafic
{
public:
    Patrat();
    virtual ~Patrat();
    virtual void Desenez();
};
```

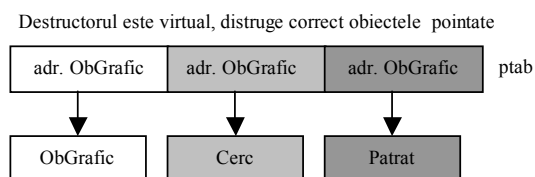


Figura 1.29. Eliberarea corectă a memoriei

### 1.3.4 Clase abstracte. Funcții virtuale pure

Am învățat până acum că o funcție o dată declarată va trebui să fie și definită, în caz contrar compilatorul generează o eroare. Există uneori situații în crearea unei ierahii de clase, în care este util să declarăm ca și superclase clase generice, care nu implementează anumite operațiuni, ci le doar declară (descriu), urmând a fi implementate în clasele derivate. Aceasta se realizează folosind **funcțiile virtuale pure**, care este un alt concept specific limbajului C++. **Pentru o funcție virtuală pură,**

*declarația este urmată de =0;* Aceasta nu înseamnă o inițializare, ci specifică caracterul virtual pur al funcției.

***O clasă care conține cel puțin o funcție virtuală pură se numește clasă abstractă.*** Ea nu poate fi instanțiată, deci nu se pot declara obiecte de această clasă, datorită faptului că ea nu furnizează implementarea metodei virtuale pure!

Ca exemplu (în proiectul *Virtual\_Pur*), putem presupune că avem o clasă de bază `Animal` care declară, fără a implementa, metoda `Hraneste()`. Această metodă este apoi implementată în clasele derivate. Clasa `Animal` este o clasă generică, adică nu putem crea obiecte de acest tip. Putem în schimb crea obiecte de tipul claselor derivate. Fie următoarele declarații de clase:

```
class Animal
{
public:
    virtual ~Animal(){};
    virtual void Hraneste()=0;
};
class Ghepard: public Animal
{
public:
    virtual ~Ghepard(){};
    virtual void Hraneste() {cout <<"\n Omoara o gazela si maninc-o";}
};
class Casalot: public Animal
{
    virtual ~Casalot(){};
    virtual void Hraneste() {cout <<"\n Prinde pesti si maninca-i";}
};
class Pisica: public Animal
{
    virtual ~Pisica(){};
    virtual void Hraneste() {cout <<"\n Bea niste lapte";}
};
```

Fișierul sursă va fi:

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

void main()
{
    Animal * ptr[3];
    ptr[0] = new Ghepard();
    ptr[1] = new Casalot();
    ptr[2] = new Pisica();
    ptr[0]->Hraneste();
    ptr[1]->Hraneste();
    ptr[2]->Hraneste();
    delete ptr[0];
    delete ptr[1];
    delete ptr[2];
}
```

Se poate observa că programul se execută corect, cu toate că funcția `Hraneste()` este doar declarată în clasa de bază, nu și implementată. ***În schimb, în clasele derivate este obligatorie definirea funcției virtuale pure!***

## 1.4 Să facem un exemplu complet

Haideți să implementăm o stivă (LIFO) și coadă de numere întregi (FIFO), pornind de la o clasă de bază abstractă, numită `Base`. Această clasă declară două metode virtuale pure, `Push()` și `Pop()`. Acestea sunt implementate în clasele derivate, specific fiecărei structuri de date. În cazul stivei, metoda `Pop()` returnează ultimul element din stivă. În cazul cozii, metoda `Pop()` returnează primul element din coadă. Metoda `Push()` introduce un întreg în capul listei interne folosite pentru stocare, în cazul stivei, sau la coada listei, în cazul cozii. Derivarea din aceeași clasă de bază și folosirea metodelor virtuale permite tratarea într-o manieră omogenă a tablourilor eterogene de obiecte de tip stivă sau coadă. Să deschidem un nou proiect **Win32 Console Application**, pe care să-l numim *Liste*.

Să implementăm fișierul header ca mai jos:

```
class Baza;
class Stiva;
class Coadă;

class ElementLista
{
    int valoare;
    ElementLista* urmatorul;
public:
    ElementLista(int i=0);
    friend class Baza;
    friend class Stiva;
    friend class Coadă;
};

class Baza
{
protected:
    ElementLista* CapLista;
public:
    // Baza(): CapLista(NULL){}
    Baza() {CapLista=NULL;}
    ~Baza();
    virtual void Afisez();
    virtual void Push(int)=0;
    virtual int Pop() =0;
};

class Stiva: public Baza
{
public:
    virtual void Afisez();
    virtual void Push(int);
    virtual int Pop();
};

class Coadă: public Baza
{
public:
    virtual void Afisez();
    virtual void Push(int);
    virtual int Pop();
};
```



Ce am declarat de fapt? O clasă `ElementLista`, care implementează o structură de date caracteristică listei simplu înlănțuite. Cum datele sunt private, dar vor fi folosite în alte clase, le-am declarat pe acestea prietene. Deoarece aceste clase nu au fost încă declarate, ele trebuie *anunțate* la începutul fișierului. Constructorul clasei construiește implicit un obiect cu câmpul `valoare=0`.

Am declarat apoi o clasă abstractă `Baza`, care declară funcțiile virtuale pure `Push()` și `Pop()` și funcția virtuală `Afisez()`. Clasa conține un pointer la clasa `ElementLista`, care va fi de fapt capul listei simplu înlănțuite. Constructorul clasei construiește acest pointer implicit `NULL` ( să ne reamintim că câmpul `valoare` era implicit 0 din construcția obiectului `ElementLista`). În exemplu sunt date două modalități de implementare a constructorului : cu listă de inițializare și respectiv „clasic”. În continuare sun declarate clasele derivate din clasa de bază, care vor implementa stiva și respectiv coada.

Fișierul sursă pentru programul exemplu va fi:

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

ElementLista::ElementLista(int i)
{
    valoare=i;
    urmatorul=NULL;
}

Baza::~Baza()
{
    ElementLista* ptr=CapLista;
    while (CapLista!=NULL)
    {
        CapLista=CapLista->urmatorul;
        delete ptr;
        ptr=CapLista;
    }
}

void Baza::Afisez()
{
    ElementLista* ptr=CapLista;
    if (ptr==NULL)
        cout << "\n Structura de date este vida! ";
    else
        while (ptr!=NULL)
        {
            cout << "\n " << ptr->valoare;
            ptr=ptr->urmatorul;
        }
}

void Stiva::Push(int i)
{
    ElementLista* ptr=new ElementLista(i);
    ptr->urmatorul=CapLista;
    CapLista=ptr;
}

int Stiva::Pop()
{
    int valret;

```

```
ElementLista* ptr;
if (CapLista==NULL)
{
    cout << "\n Stiva este vida! ";
    return 0;
}
valret=CapLista->valoare;
ptr=CapLista;
CapLista=CapLista->urmatorul;
delete ptr;
return valret;
}

void Stiva::Afisez()
{
    cout << "\n Stiva contine: ";
    Baza::Afisez();
}

void Coadă::Push(int i)
{
    ElementLista* ptr, *ElementNou= new ElementLista(i);

    if (CapLista==NULL)
        CapLista=ElementNou;
    else
    {
        ptr=CapLista;
        while(ptr->urmatorul!=NULL)
            ptr=ptr->urmatorul;
        ptr->urmatorul=ElementNou;
    }
}

int Coadă::Pop()
{
    int valret;
    ElementLista* ptr;

    if (CapLista==NULL)
    {
        cout << "\n Coadă este vida! ";
        return 0;
    }
    valret=CapLista->valoare;
    ptr=CapLista;
    CapLista=CapLista->urmatorul;
    delete ptr;
    return valret;
}

void Coadă::Afisez()
{
    cout << "\n Coadă contine: ";
    Baza::Afisez();
}

void main()
{
    Baza* ptab[2];
    ptab[0]=new Stiva;
```

```

ptab[1]=new Coadă;
ptab[0]->Push(1); ptab[0]->Push(2);
ptab[0]->Afisez();
ptab[0]->Pop();
ptab[0]->Afisez();
ptab[1]->Push(1); ptab[1]->Push(2);
ptab[1]->Afisez();
ptab[1]->Pop();
ptab[1]->Afisez();
}

```

Funcțiile `Push()` și `Pop()` sunt astfel implementate încât pentru stivă inserează și scot un element din capul listei, iar pentru coadă, inserează un element la sfârșit și respectiv scot un element din capul listei.

### Întrebări și probleme propuse

1. Implementați și executați toate exemplele propuse în capitolul 1;
2. Este întotdeauna utilă declararea unei funcții `inline`? În ce situații este utilă?
3. În ce condiții poate fi apelat din afara clasei un membru `private`?
4. Când trebuie utilizat numele calificat pentru definirea unei funcții membre a unei clase?
5. Ce deosebire este între o variabilă statică și una nestatică în declararea unei clase?
6. Când și de ce trebuie declarat explicit constructorul de copiere? În același caz este obligatorie supraînscriserea operatorului `=`?
7. Avem următoarele linii de cod:

```

class A
{
public:
    int v1;
protected:
    int v2;
private:
    int v3;
};

class B: public A
{
public:
    void afisez()
    {
        cout << v1; cout << v2;
    };
};

class C: private A
{
public:
    void SiEuAfisez()
    {
        cout << v1; cout << v2;    cout << v3;
    };
};

```

```
void main()
{
    A vara; B varb; C varc;
    vara.v1=5;
    vara.v2=7;
    varb.v1=3;
    varb.v2=5;
    varc.v1=7;
    varc.v3=8;
}
```

Care din liniile sursă vor genera erori de compilare și de ce?

8. În ce situații funcțiile trebuie declarate virtuale? Când o funcție virtuală poate fi doar declarată și nu și implementată?
9. Concepeți și implementați obiectual un program care să creeze și să execute operații cu o listă dublu înlănțuită (creare, adăugare la început și sfârșit, ștergere la început și sfârșit, parcurgere înainte și înapoi, etc);