12. CLASE.

12.1. Tipuri abstracte de date

La programarea problemelor complexe intervin concepte noi care nu pot fi exprimate simplu prin tipuri predefinite de date. De obicei, orice limbaj de programare pune la dispoziția programatorului un număr de tipuri predefinite, care însa, în mod frecvent, nu corespund tuturor conceptelor necesare programului. Astfel de concepte se implementează în limbajul C++ prin intermediul claselor. O clasă defineşte un tip abstract de date.

Prin **tip abstract de date** înțelegem o mulțime de date care au o aceeaşi reprezentare şi pentru care este definit setul de operații care se pot executa asupra elementelor mulțimii respective. Ca exemple de concepte care nu corespund unor tipuri predefinite din limhajul C++ amintim: şir de caractere; număr complex; listă; arbore; etc. Pentru fiecare din aceste concepte se poate defini un tip abstract de date prin intermediul claselor.

Din definiția tipului abstract de date rezultă că acesta are 2 părți, o parte care defineşte reprezentarea datelor tipului respectiv şi o parte care defineşte operațiile asupra datelor respective. Partea care defineşte reprezentarea datelor este formată din componente care sunt date de diferite tipuri. Aceste componente se numesc **date membru**. Partea care defineşte operațiile asupra datelor tipului respectiv conține funcții numite **funcții membru.**

Prima încercare de a face legătura dintre reprezentarea datelor unui tip şi operațiile asupra datelor respective a condus la extinderea construcției *struct*, aşa cum s-a vazut în tema precedentă. Astfel, construcția *struct* permite definirea reprezentării datelor tipului care se introduce de utilizator (date membru), precum şi enumerarea funcțiilor care definesc operații cu datele respective (funcții membru). Acest fapt nu este încă suficient pentru implementarea tipurilor abstracte de date deoarece nu se asigură protejarea datelor membru. Ele pot fi accesate direct şi de către alte funcții decît funcțiile membru.

Lipsa unei protecții a datelor, face ca tipurile de date introduse prin construcția *struct* să nu poată fi supuse unor controale cu privire la operațiile care se execută asupra lor. De aceea, pasul următor în implementarea tipurilor abstracte de date este acela de a introduce protejarea datelor şi funcțiilor membru. Acest lucru a condus la noțiunea de *clasă.*

Protejarea datelor şi funcțiilor membru se realizează utilizînd modificatorii de protecție: *private*; *protected*; *public*. Aceşti modificatori sunt urmați de ':'. Modificatorii *private* şi *protected* protejează elementele (date şi funcții membru) aflate în domeniul lor de acțiune. Domeniul de acțiune al unui modificator de protecție ține din punctul în care este scris modificatorul respectiv şi pînă la sfîrşitul definiției care îl conține sau pînă la un alt modificator de protecție.

Membrii din domeniul de acțiune al modificatorului *public* nu sunt protejați şi ei pot fi folosiți fără restricții în tot programul unde ei sunt "vizibili".

În mod implicit, membrii unei ciase sunt protejați ca şi în cazul în care s-ar afla în domeniul de acțiune a lui *private*. Despre un astfel de membru vom spune că este **privat**. El poate fi utilizat numai de către o funcție membru. Vom spune că accesul la un astfel de membru nu se face direct, ci indirect prin intermediul funcțiilor membru. Nu acelaşi lucru este valabil pentru tipurile definite cu ajutorul construcției *struct*. În acest caz, membrii *dată* sau *funcție* sunt în mod implicit publici, deci la ei putem face acces direct. Aceasta explică de ce în programul din exercițiul 21.2. se pot folosi direct (în afara funcțiilor membru) expresii de forma: *a.rea*l şi *a.imag*, unde: *a* - este o variabilă de tip *complex*. Într-adevăr, construcția *struct* nu protejează datele memhru *real* şi *imag*, deci ele pot fi folosite în mod obişnuit, la fel ca în limbajul C. De aceea, compilaturul nu poate realiza nici un control asupra operațiiior realiyate cu datele de tip *complex*, definite cu ajutorul construcției *struct*, ca în E21.1. Cu totul alta este situația dacă definim tipul *complex* cu ajutorul unei clase.

Pentru a defini o clasă se utilizează acelaşi format ca şi în cazul construcției *struct*. La definirea tipului *complex* cu ajutorul clasei vom proteja datele membru, adică *real* şi *imag*. Formatul clasei pentru tipul *complex* va fi:

```
class complex {
    //date membru private
    double real;      double imag;
    // funcții membru
}
```

În continuare, putem defini date de tip complex în mod obişnuit. De exemplu:

```
complex a;
```

defineşte pe *a* de tip complex. În acest caz, *a* este o dată care are două componente: *real* şi *imag*, ambele de tip *double* dar ele sunt private şi deci nu putem avea acces direct la ele. De aceea, instrucțiunile de mai jos sunt interzise în afara funcțiilor membru:

```
a.real = 1;     a.imag = 2;       printf("a=%g+i*(%g)\n",a.real,a.imag);
```

Mai mult decît atît, datele de acest tip nici nu pot fi inițializate în mod obişnuit. Deci, o declarație de forma:

```
complex a = {1,–1};
```

este interzisă.

Deoarece singurul acces posibil la datele membru *real* şi *imag* este prin intermediul funcțiilor membru, rezultă că este nevoie de funcții membru care să realizeze nu numai operațiile obişnuite cu numere complexe (modul, argument, adunare, scădere, înmulțire şi împărțire), ci şi alte operații cum sunt: inițializare; atribuire; afişare. Este necesar ca toate aceste funcții membru să fie publice. Rezultă că o primă variantă pentru definirea tipului *complex* ar putea fi clasa de mai jos:

```
class complex {
//date membru private
        double real;   double imag;
public:     // funcții membru publice
        void atribuire(double x=0,double y=0) {    //real+i*imag = x+i*y
            real = x;       imag = y;
```

```
    }
    double retreal() { return real; }  / returnează partea reala a numărului complex
    double retimag(){ return imag; } //returnează partea imaginara a numărului complex
    void afiscomplex(char *format) {   //afişează numărul complex conform
        printf(format,real,imag);          //formatului definit de pointerul format
    }
    void adcomplex(complex *x,complex *y) {         // calculează x+y
        real = x–>real + y–>real;          imag = x–>imag + y–>imag;     }
    void negcomplex(complex *x) {     // calculează –x
        real = –x–>real;       imag = –x–>imag;
    }
    void sccomplex(complex *x, complex *y) {    // calculează x–y
        real = x–>real – y–>real;      imag = x–>imag – y–>imag;
    }
    void mulcomplex(complex *x,complex *y) {  // calculează x*y
        real = x–>real * y–>real – x–>imag * y–>imag;
        imag = x–>real * y–>imag + x–>imag * y–>real;
    }
    int divcomplex(complex *x,complex *y);  /* - calculează x/y;
                        - returnează: 0 - la împărţirea cu zero; 1 - altfel. */
    };  //sfîrşitul definiţiei clasei complex
    int complex :: divcomplex(complex *a,complex *b) {     /* - calculează a/b;
                        - returnează: 0 - la împărţirea cu zero; 1 - altfel. */
        double d;
        d = b–>real * b–>real + b–>imag * b–>imag;
        if(d == 0.0)    return 0;
        real = (a–>real * b–>real + a–>imag * b–>imag)/d;
        imag = (a–>imag * b–>real – a–>real * b–>imag)/d;
        return 1;
    }
```

Funcţiile membru sunt funcţii *inline*, exceptînd funcţia *divcomplex*, care, conţinînd mai multe instrucţiuni decît celelalte, a fost definită în afara definiţiei clasei *complex* şi deci ea nu este o funcţie *inline*. În antetul funcţiei *divcomplex* s-a utilizat numele: *complex :: divcomplex*, construit cu ajutorul operatorului de rezoluţie. Aceasta regulă se utilizează pentru toate funcţiile membru care nu se definesc în interiorul definiţiei clasei pentru care ele sunt funcţii membru.

O clasă defineşte un tip care a fost numit tip abstract. Partea privată a unei clase defineşte modul de implementare al datelor de tipul abstract definit de clasa respectivă. Partea publică defineşte interfaţa acestor date cu restul programului. Această interfaţă conţine, de obicei, funcţii care au fost numite funcţii membru. Funcţiile membru se mai numesc şi **metode**. În general, partea publică poate conţine şi date membru, dar atunci datele respective nu mai sunt protejate, lucru care pe cît posibil trebuie evilat. Lipsa unei protecţii pentru datele rnembru nu permite compilatorului să controleze operaţiile realizate cu datele respective.

În cazul clasei *complex*, programatorul nu are acces direct la componentele *real* şi *imag* ale datelor de tip complex. Toate operaţiile care se pot realiza cu datele de acest tip se fac numai prin intermediul funcţiilor membru care au fost declarate ca publice.

Datele ale cărui tip se definesc printr-o clasă se numesc **obiecte**. Cu alte cuvinte, un obiect în C++ este o dată de un tip abstract. Obiectele, la fel ca şi datele predefinite, se declară printr-o declaraţie. În forma cea mai simplă, o declaraţie de obiect are formatul:

```
    nume_clasă lista_de_nume_de_obiecte;
```

unde prin *lista_de_nume_de_obiecte* înţelegem un nume sau mai multe separate prin virgulă. De asemenea, *nume_clasă* este un nume.

Exemplu: Dacă se consideră definiţia clasei *complex* de mai sus, atunci:

```
    complex z,xl,x2;
```

declară pe *z, xl* şi *x2* ca obiecte, adică date de tip *complex*.

Despre obiectele unei clase se obişnuieşte să se spună că sunt **instanţieri** ale clasei respective. Deci *z, xl* şi *x2*, sunt instanţieri ale clasei complex.

Pentru a apropia tipurile abstracte de cele predefinite, se impune ca obiectele să poată fi iniţializate. Iniţializarea unui obiect este o problemă mult mai complexă decît cea a datelor obişnuite (tipuri predefinite sau definite de utilizator cu ajutorul construcţiilor *struct* sau *union*). De aceea, iniţializarea unui obiect este posibilă folosind în acest scop o funcţie membru specială. Aceasta poartă denumirea de **constructor**. Constructorul unei clase are totdeauna acelaşi nume cu numele clasei. Dacă o clasă are constructor, atunci acesta se apelează automat la instanţierea unui obiect al clasei respective.

Exemplu: Pentru clasa *complex* definită mai sus, putem defini un constructor cu ajutorul funcţiei membru:

```
    complex(double a=0,double b=0) {      //constructorul clasei complex
real = a;    imag = b;
    }
```

Această funcţie membru se presupune că se defineşte în interiorul definiţiei clasei *complex*. Deci, este o funcţie *inline*. Pentru a o defini în afara definiţiei clasei *complex* se schimbă antetul funcţiei astfel:

inline complex :: complex(double a,double b)

Pentru a iniţializa un obiect, în declaraţia obiectului, după numele lui, se includ în paranteze rotunde valorile efective ale parametrilor constructorului. Exemplu:

complex z(1,2);

Se instanţiază obiectul *z*, de tip complex, ale cărui componente au valorile iniţiale:

z.real = 1 şi z.imag = 2

În cazul în care există o singură valoarea de iniţializare, atunci se poate utiliza semnul de atribuire (=). De exemplu, declaraţia:

complex z = 123;

este identică cu declaraţia:

complex z(123);

Ele iniţializează obiectul *z* astfel:

z.real = 123; z.imag = 0 - valoarea implicită.

La o declaraţie de forma:

complex z;

adică fără iniţializare, constructorul este apelat în mod automat şi se instanţiază obiectul *z* cu ambele componente egale cu zero (valorile implicite ale parametrilor constructorului).

Obiectele sunt distruse de către programator sau în mod automat. Distrugerea obiectelor se poate realiza folosind o funcţie membru speciala numită **destructor**. Un destructor are un nume aparte şi anume:

~nume

unde: *nume* - este numele clasei. De asemenea, destructorul nu are parametri. El se apelează de programator cînd nu mai este nevoie de un obiect. Acest lucru este important atunci cînd obiectul a fost creat în memoria heap.

Destructorul se apelează automat cînd obiectul îşi încetează existenţa.

Funcţiile membru se apelează calificînd numele lor cu numele obiectului pentru care se realizează apelul:

nume_obiect.nume_funcţie_membru(lista parametrilor efectivi)

De exemplu, dacă *z* este o instanţiere a clasei *complex* (complex z), atunci pentru a afişa partea reală şi cea imaginară a obiectului *z*, se va apela funcţia membru *afiscomplex*:

z.afiscomplex("Partea reală = %g\t Partea imaginară = %g\n");

În mod analog, dacă obiectele *z, zl* şi *z2* sunt instanţieri ale clasei *complex,* atunci pentru a realiza atribuirea:

z = zl+z2;

se va apela funcţia membru *adcomplex* astfel:

z.adcomplex(&zl, &z2);

În cazul în care se utilizează un pointer spre un obiect:

nume_clasa *po;

funcţiile membru se apelează printr-o construcţie de forma:

po–>nume_funcţie_membru(lista parametrilor efectivi)

În corpul unei funcţii membru, ne putem referi la obiectul pentru care a fost apelată funcţia membru folosind pointerul implicit *this*. Acest pointer se utilizează în corpul funcţiilor membru în mod implicit la datele membru ale obiectului pentru care este apelată funcţia membru respectivă. Cu toate acestea, programatorul poate utiliza explicit pointerul this pentru a se referi, în corpul unei funcţii membru, la obiectul pentru care s-a apelat funcţia respectivă.

În continuare prin *obiect curent* vom înţelege obiectul pentru care s-a apelat o funcţie membru, adică obiectul spre care pointează pointerul *this*. Apelul unei funcţii membru referitor la un obiect al clasei pentru care funcţia respectiv este funcţie membru, simplifică modul de exprimare al operaţiilor care se realizează asupra obiectelor. De asernenea, acest mod de apel poate fi controlat de compilator. Orice apel neautorizat va fi semnalat şi eliminat de compilator. În feiul acesta, lucrui cu obiectele unei clase devine similar cu cel existent pentru datele de tipuri predefinite: obiectele se pot iniţializa; asupra lor se pot face numai operaţii în prealabil definite prin funcţii membru şi care sunt controlabile prin compilator.

Protecţia datelor şi funcţiilor membru ale unei clase care nu sunt publice constituie un mijloc de "ascundere" a elementelor membru respective. Această "ascundere", numită şi **incapsulare** se realizează la un nivel calitativ superior faţă de ascunderea realizată cu ajutorul modulelor.

De exemplu, prin intermediul clasei *complex* se pot instanţia oricîte obiecte (numere) de tip *complex* care pot fi şi iniţializate, iar operaţiile cu ele se realizează simplu cu ajutorul funcţiilor membru şi aplicarea lor este controlată de compilator.

Mai jos, introducem tipul abstract *stiva* pentru a pune mai bine în evidenţă diferenţa dintre facilităţile obţinute pe bază utilizarii claselor faţă de cele oferite de programarea modulară. Vom implementa tipul *stiva* tot cu ajutorul unui tablou de tip *int* care are cel mult 100 de elemente. Modulul definit în tema 7 implementează o singură stivă. În cazul de faţă se pot instanţia oricîte obiecte de tip stivă.

```
enum Boolean {false,true};
class stiva {
int stack[100];        int istack;
public:
stiva ( ) {      // coastructor
 istack=0; // la început stiva este vidă
}
void push(int n);      //pune pe n pe stivă
int pop();  //scoate elementul din vîrful stivei. se returnează elementul din vîrful stivei
```

```
int top();   //returnează elementul din vîrful stivei. elementul rămîne pe stivă
void clear() { // videază stiva
    istack=0;
}
    Boolean empty( ) {   //returnează true dacă stiva este vidă şi false altfel
return istack == 0 ? true : false;
    }
    Boolean full ( ) {      // returnează true dacă stiva este plină şi false altfel
return istack == 100 ? true : false;
    }
    };
```

În continuare, se pot instanția obiecte de tip *stiva*:

```
stiva stiva1, stiva2;
```

La instanțierea obiectelor *stiva1* şi *stiva2* se apelează în mod automat constructorul clasei. Acesta realizează atribuirea:

```
istack = 0;
```

În felul acesta, stivele *stiva1* şi *stiva2* sunt vide la instanțiere.

Datele membru *stack* şi *istack* sunt "incapsulate" (ascunse) în clasa *stiva*, fiind private. Ele nu pot fi accesate direct ca şi în cazul modulului definit în tema 7. Stivele se pot gestiona simplu cu ajutorul funcțiilor membru. Astfel, pentru a pune un element pe stiva *stiva1*, utilizăm apelul funcției *push*:

```
stiva1.push(expresie)
```

În mod analog, punem un element pe stiva *stiva2*, apelînd aceeaşi funcție *push* ca mai jos:

```
stiva2.push(expresie)
```

În felul acesta, funcțiile membru gestionează simplu oricîte stive de tip *stiva*. Acest lucru nu este posibil prin intermediul modulului descris în tema 7. Evident, se poate construi un modul care să gestioneze mai multe stive, dar acest lucru nu se realizează atît de simplu şi elegant ca mai sus, prin utilizarea claselor. Acest fapt explica afirmația lui B. Stroustrup: dacă într-un program este suficient un *singur* exemplar al unei date de un anumit tip, atunci este suficient un modul pentru lucrul cu data respectiva. Altfel, se defineşte o clasă pentru tipul datei respective şi astfel se vor putea instanția oricîte exemplare.

În aceasta constă esența saltului realizat prin trecerea de la programarea modulară la stilul programării prin abstractizarea datelor.

În rezumat, programarea prin abstractizarea datelor are la bază utilizarea tipurilor abstracte de date. În acest scop, se porneşte cu stabilirea conceptelor necesare la realizarea unui program. O parte din aceste concepte se realizează cu ajutorul tipurilor predefinite, existente în limbajul C++. Celelalte concepte se realizează sub formă de tipuri abstracte de date care se definesc în limbajul C++ cu ajutorul claselor.

Un tip abstract de dată prezinti 2 aspecte: unul legat de implementarea tipului şi celălalt legat de utilizarea lui, care defineşte interfața tipului respectiv cu restul programului. Implementarea tipului este partea lui protejată, iar interfața constituie partea publică a tipului, utilizabilă în tot programul. Despre partea protejată se spune că, conține informația incapsulată în clasa care defineşte tipul respectiv.

La definirea unei clase se precizeaza: - reprezentarea datelor tipului abstract; - funcțiile care descriu operații cu datele tipului abstract.

Reprezentarea datelor se defineşte prin componente date numite **date membru**.

Funcțiile care definesc operații cu datele tipului abstract,se numesc **funcții membru**.

De obicei, partea incapsulată în clasă conține date membru, dar ea poate conține şi funcții membru.

Elementele publice sunt funcții membru. Ele pot fi şi date membru, dar atunci acestea nu mai sunt protejate, lucru care este bine să fie evitat.

Un tip abstract de date trebuie astfel definit, încît la utilizarea lui să se facă abstracție de implementare. Mai mult decît atît, utilizarea datelor de un tip abstract nu trebuie să fie intluențată de implementare. Aceasta înseamnă că irnplementarea tipului abstract poate fi oricînd schimbată, fără a schimba şi utilizarea, în program, a datelor tipului respectiv.

De exemplu, la implementarea tipului *stiva* s-a folosit un tabiou de tip *int* de 100 de elemente şi vaziabiia *istack* de tip *int*. Schimbînd implementarea prin înlocuirea tabloului cn o listă simplu înlănțuită (tema 11), interfața formată din funcțiile membru: *push, pop, top, clear, empty* şi *ful1* se utilizează în acelaşi mod. Evident, funcțiile se modifică, dar ele sunt apelate la fel şi au acelaşi efect. Dacă acest lucru nu este realizabil, înseamnă că tipul abstract nu a fost bine definit, deoarece la utilizarea lui nu se poate face abstracție de implementare.

Datele de un tip abstract sunt numite **obiecte** ale tipului respectiv. Obiectele se declară printr-o declarație asemănătoare cu declarația datelor pentru tipuri predefinite. Ele pot fi inițializate la declararea lor. Obiectele se creează, de obicei, printr-un **constructor**, care este o funcție membru specială, al cărui nume coincide cu numele clasei. Constructorul unei clase se apelează automat la construirea unui obiect al clasei respective. Despre un obiect al unei clase se spune că este o instanțiere a clasei respective. Un obiect poate fi distrus printr-o funcție specială numită **destructor.** Numele destructorului este numele clasei precedat de caracterul "~".

Funcțiile membru pot fi definite în interiorul definiției clasei dacă sunt foarte simple. Aceasta deoarece în acest caz, ele se definesc în mod implicit ca funcții *inline*. O funcție membru definită în afara definiției clasei are numele din antet construit cu ajutorul operatorului de rezoluție;

```
nurne_clasa :: nume_funcție_membru
```

La apelul unei funcții membru se califică numele funcției cu numele obiectului pentru care se apelează funcția.

Facilitaţile obţinute cu ajutorul claselor apropie modul de tratare şi utilizare al obiectelor de cel al datelor de tipuri predefinite.

O diferenţă între utilizarea obiectefor şi a datelor de tipuri predefinite apare la scrierea expresiilor. Aşa de exemplu, dacă se consideră datele declarate ca mai jos:

```
int i, j, k;      double a,b,c;
```

Atunci se pot utiliza expresii de forma:

```
k = i+j;          c = a+b;
```

care sunt sugestive faţă de exprimarea adunării datelor de tip *complex* prin apelul funcţiei membru *adcomplex*:

```
complex z,u,v;    …      v.adcomp1ex(&z,&u);
```

De accea, ar fi bine ca expresii de forma:

```
v = z + u;
```

să fie acceptate de compilator şi pentru obiecte. Această idee ne conduce la noţiunea de **supraîncărcare a operatorilor**. Într-adevăr, se poate spune că operatorul + este supraîncărcat pentru tipuri predefinite. El se poate aplica la operanzi de tipuri numerice predefinite (*int, long, float, double, unsigned* şi *long double*). De aceea, pentru a putea scrie adunarea obiectelor complexe *z* şi *u* sub forma expresiei: *z + u* ca în cazul tipurilor numerice predefinite, este suficient ca operatorul + să poată fi supraîncărcat cu operanzi de tip complex.

Supraîncărcarea operatorilor este o facilitate care simplifică mult exprimarea operaţiiior asupra obiectelor. Pe baza ei se pot scrie expresii cu obiecte la fel ca şi cu date de tipuri predefinite. Supraîncărcarea operatoriior se realizează prin construcţii asemănătoare funcţiilor dar care au un antet special în care este prezent cuvîntul *operator*.

Supraîncărcarea operatorilor pentru obiecte este completată şi cu diferite conversii care se aplică la operaţii cu obiectele respective. În felul acesta se ajunge să utilizăm obiecte1e la fel de simplu ca şi datele de tipuri predefinite.

Putem afirma că, clasele sunt un instrument pentru *a crea* tipuri noi care poi fi utiţizate tot aşa de bine ca şi tipurile predefinite. Ideal, tipurile noi (tipurile abstracte) nu trebuie să difere, în utilizare, de tipurile predefinite, ci numai în modul în care sunt create.

12.2. Definiţia claselor

O clasă defineşte un tip abstract de date. Ea are o definiţie a cărei format simplificat:

class nume (lista elementelor membru);

Ulterior o să vedem şi un alt format mai complex, pentru clase.

În limbajul C++, formatul de mai sus poate fi folosit şi pentru a defini tipuri noi cu ajutorul cuvintelor cheie *struct* şi *union*. Deci, în formatul de mai sus, se poate înlocui cuvîntul *class* prin *struct* sau *union*. De fapt, în cele ce urmează, vom considera că şi formatele în care cuvîntul cheie *class* se înlocuieşte prin *struct* sau *union* definesc tot clase. Diferenţa constă în aceea că, în cazul utilizării cuvîntului *class*, elementele membru în mod implicit sunt protejate prin protecţia oferită de modificatorul de protecţie *private*, iar în cazul lui *struct* şi *union*, în mod implicit elementele membru sunt neprotejate (publice). În cazul utilizării cuvîntului *struct* se poate modifica protecţia cu ajutorul modificatorilor de protecţie. În schimb, în cazul utilizării cuvîntului **union**, elementele membru pot fi numai publice. De aceea, se obişnuieşte să se spună că *struct* şi *union* definesc clase cu elemente membru publice. În cele ce urmează vom înţelege prin clasă de tip *struct* o clasă definită cu ajutorul lui *struct*, iar prin clasă de tip *union*, o clasă definită cu ajutorul lui *union*.

Numele aflat după *class, struct* sau *union* este numele tipului introdus prin definiţia respectivă. El se numeşte şi **numele clasei** şi în continuare poate fi utilizat pentru a declara (instanţia) date de tipul respectiv, date numite şi obiecte de tipul respectiv. Lista elemente1or membru poate conţine: - declaraţii de date; - definiţii de funcţii; - prototipuri de funcţii; - modificatori de protecţie.

Datele declarate într-o definiţie de clasă se numesc **date membru**. Funcţiile definite sau pentru care este prezent numai prototipul,în definiţia clasei,se numesc **funcţii membru**.

Funcţiile definite în definiţia unei clase trebuie să fie simple, deoarece ele se definesc în mod implicit ca funcţii *inline*. Pentru funcţiile mai complexe se indică în definiţia clasei numai prototipurile funcţiilor respective şi ele se definesc ulterior. În acest caz, antetul funcţiei conţine un nume format cu ajutorul operatorului de rezoluţie:

```
nume_clasă :: nume_funcţie_membru
```

De obicei, o clasă are unul sau mai mulţi constructori precum şi un destructor. Aceştia sunt funcţii membru de nume speciale, şi anume, numele constructorilor coincide cu numele clasei (dacă sunt mai mulţi constructori, atunci aceştia sunt funcţii supraîncărcate), iar numele destructorului este numele clasei precedat de caracterul ~.

Un alt caz particular de funcţii membru sunt cele care definesc operatori şi conversii pentru obiectele clasei respective. Ca modificatori de protecţie am văzut că se pot folosi: *private:, protected:* şi *public:*. Primii doi asigură o protecţie a datelor sau funcţiilor membru din domeniul de acţiune al lor, iar ultimul se utilizează pentru elemente membru care dorim să nu fie protejate. Domeniul unui modificator de protecţie începe din punctul în care este scris şi pînă la sfîrşitul definiţiei clasei care îl conţine sau pînă la întîlnirea unui alt modificator de protecţie. De obicei, datele membru sunt protejate, dar aceasta nu înseamnă că ele nu pot fi publice. De asemenea, funcţiile membru, numite şi metode, de obicei sunt publice, dar ele pot fi şi protejate.

Clasele pot fi definite incomplet în cazul în care este nevoie sa ne referim la ele. O astfel de definiţie incompleta are formatul: *class nume; struct nume;* sau *union nume;*

Datele membru se declară în mod obişnuit. Dacă este prezentă o clasă de memorie, atunci aceasta poate fi numai clasa de memorie *static*. O dată membru a unei clase nu poate fi de tipul definit prin clasa respectivă. Ea poate fi numai un pointer spre tipul respectiv sau o referinţă la tipul respectiv. Fie definiţia:

```
class nume {  …
    nume *ptr; //corect   …
```

```
    nume& ref; // corect …
    nume obiect; // incorect
};
```

Data membru *obiect* de tip *nume* nu este admisă. În schimb, se pot declara date de tipuri introduse prin a1te clase.

Domeniul de existență ai numelui unei clase este din punctul definirii ei pînă la sfîrșitul blocului (instrucțiunii compuse care o conține). De obicei, definifia unei clase se scrie la începutul fișierului sursă în care este utilizată și în afara oricărui bloc. Se obișnuiește adesea să se construiască un fișier de tip *h* care să conțină definiții de clase. Fișierul respectiv se include la începutul fiecărui fișier care utilizează definițiile claselor respective.

12.3. Obiecte

Un obiect este o dată de un tip definit printr-o clasă. Se obișnuiește să se spună că este o instanțiere a clasei respective. Declarația de obiecte este asemănătoare cu cea pentru datele de tipuri predefinite. În cea mai simplă formă, un obiect se declară folosind formatul:

```
nume_clasă nume_obiect;
```

De obicei, clasele au constructori care se apelează automat la întîlnirea declarației de instanțiere a unui obiect al clasei respective.

Datele membru se alocă distinct la fiecare instanțiere a clasei. Deci, datele membru există în atîtea exemplare cîte obiecte au fost instanțiate. O excepție o constituie datele membru care au clasa de memorie *static*. O dată nembru de clasa de memorie *static* (numită *dată membru statică*) este o parte comună pentru toate instanțierile clasei şi există într-un singur exemplar.

Funcțiile membru sunt într-un singur exemplar oricîte instanțieri ar exista. O funcție membru se apelează totdeauna în strînsă dependență cu un obiect care este o instanțiere a clasei pentru care funcția respectivă este funcție membru. Legătura dintre funcții membru şi obiectul pentru care se face apelul se realizează folosind operatorul *punct* sau *săgeata*.

Exemplu: Considerăm tipul complex definit la începutul temei. Fie declarațiile:

```
complex z;        complex *pz;      char *format = "%g\t%g\n";
```

Atunci:

```
z.afiscomplex(format);
```

afişează componentele numărului complex *z* (în cazul de față 0 0 deoarece constructorul clasei are parametri impliciți zero). Acelaşi efect se obține folosind secvența:

```
pz = &z;        pz –> afiscomplex(format);
```

Programatorul poate utiliza în mod explicit pointerul *this*, în corpul unei funcții membru. Acesta are ca valoare adresa obiectului curent, adică a obiectului pentru care s-a făcut apelul funcției membru. O excepție de la aceste reguli o reprezintă funcțiile membru care au clasa de memorie *static*. Acestea se numesc **funcții membru statice**. O astfel de funcție poate fi apelată în 2 moduri:

- Ca orice funcție membru, folosind operatorul punct sau săgeată.
- Independent de un obiect al clasei pentru care este funcție membru. În acest caz, apelul se realizează folosind operatorul de rezoluție, adică sub forma:

```
nurne_clasă:: nume_funcție_membru_statică
```

În acesl caz, pointerul *this* nu mai poate fi utilizat. Exemplu:

Fie definiția de clasă:

```
class dc {
int zi,luna,an;
static int zz,ll,aa;
    public:
dc(int z=l,int l=l,int a=1995) {
     zi = z; luna = l; an = a;
    }
static Boolean v_calend(dc *d);
    .   .   .
    };
```

Datele membru statice: *zz, ll, aa* sunt comune pentru toate instanțierile, spre deosebire de celelalte date care se alocă distinct la fiecare instanțiere. Acestea din urmă, se inițializează cu ajutorul constructorului.

Funcția *v_calend* verifică dată calendaristică definită de datele membru *zi, luna* şi *an*, care sunt componente ale obiectului spre care pointează *d*, precum şi cea definită de datele membru statice *zz, 11* şi *aa*. Ea poate fi definită ca mai jos:

```
Boolean dc :: v_calend(dc *d) {      /* verifică data calendaristică zi, luna, an, ale obiectului
                  spre care pointează d;
                   -zz, 11, aa; - returnează True daca datele sunt corecte şi False altfel. */
static tnrz[ ] = (0,31,28,31,30,31,30,31,31,30,31,30,31);
int z,l,a;
    if(d –> an < 1600 || d –> an > 4900) return False;
if(d –> luna < 1 || d > luna > 12)     return False;
if(d–>zi < 1 || d–>zi > tnrz[d–>luna]+ (d–>luna==2 &&
    (d–>an%4==0 && d–>an%100 || d–>an%400 == 0)))    return False;
z = dc :: zz;    1 = dc :: ll;     a = dc :: aa;
if(a < 1600 || a > 4900)   return False;
```

```
if(l < 1 || l > 12)   return False;
if(z<1 || z>tnrz[1]+(l==2 && (a%4==0 && a%100 || a%400==0)))      return False;
return True;
    }
    Fie instanțierea:
    dc data_calend(29,2);
```

Pentru a valida instanțierea *data_calend* se va apela funcția *v_calend* astfel:

```
    if(dc::v_calend(&data_calend) == False) {
.   .   .   //dată calendaristică eronată
    else {
.   .   .    // dată calendaristică corectă
    }
```

În acest caz, funcția *v_calend* a fost apelată folosind operatorul de rezoluție.

Se verifică data calendaristică definită de datele membru *zi, luna* şi *an* a instanțierii *data_calend*. Constructorul acestei clase a inițializat obiectul *data_calend* cu valorile:

zi =29 luna = 2 an = 1995

Datele membru statice (zz,ll,aa) există în afara instanțierilor clasei *dc*. Deşi aceste date sunt private, referirea la ele se poate face în funcția *v_calend* care este o funcție membru a clasei *dc*. Totuşi, referirea la ele nu se poate face direct, deoarece în acest caz pointerul *this* nu este definit, funcția *v_calend* nefiind apelată pentru un obiect. De aceea, referirea la datele statice *zz, ll*, şi *aa* s-a facut folosind numele clasei şi operatorul de rezoluție:

dc :: zz, dc :: ll şi dc :: aa

12.4. Domeniul unui nume

Unui nume îi corespunde un **domeniu**. Acesta se defineşte prin declarația lui sau este corpul unei funcții dacă el este numele unei etichete. Prin bloc înțelegem o instrucțiune compusă. În limbajul C o declarație poate fi în interiorul unui bloc sau în afara blocurilor. În plus, în limbajul C++ o declarație poate fi şi în interiorul unei ciasc (în interiorul declarației de clasă).

O declarație, care este în afara blocurilor sau a claselor şi care nu este o declarație de date externe, se spune că este o **defniție**.

Pentru un nume vom distinge 3 tipuri de domenii, în funcție de poziția declarației (definiției) sale. Aceste tipuri sunt: **local**, **fişier** şi **clasă**. Un nume declarat într-un bloc are *un domeniu de tip local*. Acesta începe în punctul în care este declarat şi ține pînă la sfîrşitul blocului respectiv. Un astfe1 de nume poate fi utilizat în domeniul lui, inclusiv în blocurile incluse în domeniul respectiv, dacă el nu este redeclarat în aceste blocuri.

Exemplu: Fie instrucțiunile compuse imbricate de mai jos:

```
    {
.   .   .
int i;      // începe domeniul de tip local a lui i
.   .   .   // nu începe alt bloc
    i=10;       // atribuire corectă
    .   .   .
    {
    int j;
    .   .   .   // nu există declarație pentru i
    j= i+2;     //expresie corectă deoarece i se află în domeniul lui şi nu a fost redeclarat
    .   .   .
}
    .   .   .
{
    long n;     long i; // redeclararea lui i în acest bloc nu se poate utiliza variabila i
    .   .   .           //   declarată la inceput de tip int
    n = i+20;           //se utilizează i de tip long
    .   .   .
}       // în acest punct se termină domeniul lui i declarat de tip long i nu
.   .   .   //există redeclararea lui i. Este valabila prima declarație a lui i
i=100; //se atribuie 100 variabilei i de tip int
    .   .   .
    }       // în acest punct se termină domeniul lui i declarat int i
```

Un nume declarat în afara oricărui bloc sau declarație (definiție) de clasă are *un domeniu de tip fişier*. Acest domeniu începe în punctul în care numele este definit şi ține pînă la sfîrşitul fişierului care conține definiția respectivă. El poate fi utilizat în domeniul respectiv fără nici o restricție dacă nu este redefinit în blocurile incluse în domeniul său.

Dacă un nume care are un domeniu de tip fişier este redefinit într-un bloc inclus în domeniul său, atunci el poate fi folosit, în acel bloc, dacă este precedat de operatorul de rezoluție. Exemplu:

```
    .   .   .
    int i = 100;    //definiția lui i nu este inclusă în nici un bloc sau clasă;
            //are domeniu de tip fişier care începe în acest punct
    {
```

```
int j;
.  .  .   // i nu este redeclarat pînă în acest punct
j=i+123;  //se utilizează i definit mai sus
   .  .  .
long i;    // redeclarare a lui i. Începe un domeniu de tip local pentru i
j=i–123;  //se utilizează i declarat prin long i
   .  .  .
j=j+ :: i   //se utilizează i declarat prin int
   .  .  .
   }  .  .  .
   //sfîrşit fişier: se termină domeniul lui i declarat prin int i
```

O clasă are o declarație (definiție) care defineşte un tip abstract de date. Numele acestui tip este considerat totodată ca fiind numele unei clase. Numele unei clase are un domeniu care se stabileşte la fel ca şi domeniul oricărei variabile. Un nume de clasă poate fi redeclarat ca orice nume. În acest caz, putem folosi numele respectiv într-un domeniu inclus în care este redefinit folosind construcția:

class :: nume clasa

în locul numelui. Exemplu:

```
class a {
.  .  .
};
.  .  .
{      // instrucțiune compusă aflată în domeniul numelui a
double a;  //redeclararea lui a
a = 3. 1415;   //atribuire corectă
class :: a x;   //x este o instanțiere a lui a
   .  .  .
}
```

Această regulă se utilizează şi în cazul construcțiilor *struct*, *union* şi *enum*. De fapt, construcțiile *struct* şi *union* se consideră că definesc clase ale căror elemente membru sunt toate publice.

Un nume al unui element membru al unei clase care nu este public are *un domeniu de tip clasă*. Aceasta înseamnă că el poate fi folosit numai în corpul funcțiilor membru ale clasei respective. Elementele membru statice vor fi prefixate de numele clasei urmat de operatorul de rezoluție (§ precedent). O clasă poate conține ca şi date membru obiecte care sunt instanțieri ale altei clase, dar nu ale clasei respective. Exemplu:

```
class clasa {
.  .  .
clasa1 obiect1;   // obiect1 este instanțiere a clasei clasa1
clasa obiect;  // eroare; o clasă nu poate conține ca obiecte membru instanțieri ale ei
   .  .  .
}
```

În schimb, se pot defini date membru ale unei clase care să fie pointeri sau referințe spre obiectele clasei respective. Exemplu:

```
class clasa {
.  .  .
clasa *pobiect;   //pointer spre un obiect de tip clasa
clasa& robiect;   //referință la un obiect de tip clasa
   .  .  .
};
```

Clasele pot fi declarate incomplet. O astfel de declarație are formatul:

class nume;

Astfel de declarații se pot folosi şi în cazul construcțiilor *struct* şi *union*. Exemplu:

```
class a;    struct. s;   union u;
class b {
.  .  .  a *pa;     s *ps;     u *pu;     .  .  .
};
```

12.5. Vizibilitatea şi durata de viață a datelor

Un nume este vizibil în domeniul său dacă nu este redefinit în blocuri incluse în domeniul respectiv. Un nume redefinit în blocuri din domeniul său, devine temporar ascuns. Un nume cu domeniul de tip fişier poate fi făcut vizibil în domeniul în care este redefinit, folosind operatorul de rezoluție iar dacă numele respectiv este numele unei clase, atunci el va fi precedat de cuvîntul cheie corespunzător: *class*, *struct* sau *union*.

Domeniul de vizibilitate al unui nume este acea parte a domeniu1ui său în care el poate fi utilizat. De obicei, domeniul de vizibilitate al unui nume coincide cu domeniul său.

Durata de existență a datelor este legată de clasa de memorie a acestora. Prin **durata de viață** a datelor se înțelege perioada în care ele sunt alocate în memorie.

Există 3 feluri de durată: statică, locală şi dinamică.

Durata statică - înseamnă că data respectivă este alocată în memorie pe perioada execuţiei programului: de la lansare, şi pînă la terminarea execuţiei programului.

Datele care au un domeniu de tip fişier sunt date cu durată statică. Datele care au un domeniu de tip fişier sunt date globale sau locale fişierului dacă ele sunt declarate cu ajutorul cuvîntului cheie *static*. În ambele cazuri, ele au o durată statică. De asemenea, datele care au un domeniu de tip local au o durată statică, dacă sunt declarate cu ajutorul cuvîntului cheie *static*. În conciuzie, datele globale, precum şi cele declarate cu ajutorul cuvîntului *static* au o durată statică.

Durată locală - este durata datelor automatice. Acestea sunt date cu domeniu de tip local şi care sunt alocate, la execuţie, pe stivă sau în regiştri. Ele nu conţin, în declaraţia lor, cuvîntul *static*. Alocarea pe stivă se face cînd controlul programului ajunge la blocul în care sunt declarate. Cînd contrulul programului iese din blocul respectiv, datele se elimină de pe stivă.

Durată dinamică - este durata datelor alocate în memoria heap. Acestea se alocă şi se eliberează la execuţie prin funcţii sau operatori corespunzători. Ea se realizează de către programator. În acest scop, în limbajele C şi C++, se pot utiliza funcţiile *malloc* şi *free*. De obicei, în limbajul C++ se utilizează operatorii *new* şi *delete*.

12.6. Alocarea şi dezalocarea obiectelor

Alocarea obiectelor se face în funcţie de durata de viaţă a obiectelor. Obiectele de durată statică şi locală se alocă automat. Obiectele dinamice se alocă de către programator. În acest scop, de obicei, se utilizează operatorul *new*.

În mod analog, dezalocarea obiectelor se realizează automat dacă ele au durată statică sau locală şi de către programator dacă sunt dinamice. Obiectele alocate cu ajutorul operatorului *new* se dezalocă cu ajutorul operatorului *delete*.

Alocarea obiectelor se mai numeşte şi **crearea** sau **construirea obiectelor**.

Dezalocarea obiectelor se mai numeşte şi **distrugerea obiectelor**.

Alocarea obiectelor statice care sunt globale se realizează înainte de execuţia funcţiei *main* a programului. Ele se dezalocă la terminarea programului ca parte a procedurii de ieşire din funcţia *main*.

Obiectele locale se alocă în momentul în care domeniul lor devine activ, adică atunci cînd controlul programului ajunge la instanţierea lor. Ele se dezalocă în momentul în care controlul programului iese din domeniul lor.

Alocarea obiectelor este o operaţie mai complexă decît alocarea datelor de tip predefinit sau definit de utilizator. Alocarea datelor de tip predefinit sau definit de utilizator se poate face împreună cu iniţializarea datelor respective. În cazul obiectelor, iniţializarea datelor membru este o problemă mai complexă din cauza protecţiei datelor respective. De aceea, se pune problema ca iniţializarea obiectelor să fie o operaţie care se realizează la alocarea lor, prin funcţii membru speciale. Aceste funcţii membru, care realizează alocarea şi iniţializarea obiectelor, se consideră că ele construiesc obiectul care se instanţiază. De aceea, ele se numesc constructori.

Un constructor este o funcţie membru al unei clase care se apelează la fiecare instanţiere. El are acelaşi nume ca şi numele clasei.

Dezalocarea unui obiect este şi el un proces complex care se realizează cu o funcţie membru specială numită **destructor**. Destructorul unui obiect se apelează la distrugerea obiectului. El se apelează automat sau uneori explicit de către programator. Apelul explicit al destructorului se face pentru a distruge obiectele dinamice. Pentru celelalte obiecte, destructorul se apelează automat la încetarea existenţei lor: la ieşirea prin funcţia *exit* pentru obiectele globale; la ieşirea din domeniul unui obiect de durată locală.

12.7. Iniţializare

Datele pot fi iniţializate prin deciaraţiile (definiţiiie) lor. În general, datele de durată statică neiniţializate, au valoarea iniţială egală cu zero. Celelalte categorii de date dacă nu sunt iniţializate, au o valoare iniţială nedefinită.

Datele de tipuri predefinite şi cele definite de utilizator se iniţializează folosind formatele din limbajul C. În limbajul C++ se pot iniţializa şi datele de tip ufilizator definite cu ajutorul construcţiei *union*. Aceasta este posibil numai pentru prima componentă a reuniunii.

Elementele unui tablou, unei structuri, unei reuniuni sau un enumerator, se pot iniţializa prin expresii constante, adică expresii care conţîn operanzi ce pot fi evaluaţi la compilare, la întîlnirea lor. În aceste expresii se poate folosi operatorul *sizeof*.

Datele care nu sunt tablouri, structuri, reuniuni sau de tip enumerare, se pot iniţializa prin expresii care nu este necesar să fie expresii constante. În acest caz, operanzii expresiilor care se utilizează la iniţializare, trebuie să poată fi evaluaţi în momentul în care controlul programului ajunge la ei. Exemplu:

```
    int f(int n) {
  int i = n+10;  /* parametrul n are precizată valoarea cînd controlul programului ajunge la evaluarea expresiei n+10 şi
    anume valoarea lui este egală cu a parametrului efectiv de la apelul lui f */
  int j = k–2;     //eroare; k nu este definit la întîlnirea expresiei k–2
  int k=3;    .    .    .
    }
```

Un loc aparte îl ocupă iniţializarea obiectelor. Aşa cum s-a spus mai sus, obiectele se iniţializează cu ajutorul constructorilor care se apelează în mod automat la instanţierea lor.

Datele membru ale obiectelor statice care nu sunt iniţializate au valoarea iniţială zero.

Datele membru ale celorlalte obiecte (obiecte de durată locală sau dinamică) care nu sunt iniţializate, au o valoarea iniţiala nedefinita.

Datele membru statice se iniţializează în afara constructorilor. Aceasta, deoarece o dată membru statică este o zonă comună care nu se multiplică la fiecare instanţiere a clasei respective. Iniţializarea unei astfel de date se realizează la fel ca o dată globală obişnuită, adică printr-o definiţie a datei respective în care este prezentă şi valoarea de iniţializare, definiţie care se scrie în afara corpului oricărei funcţii. Exemplu:

```
class dc {
int zi,luna,an;      static int zz,ll,aa;
public:     …
};
```

Datele membru statice *zz*, *ll*, *aa* se vor iniţializa astfel:

```
int dc::zz = 1;      int dc::ll = 1;      int dc::aa = 1600;
```

Se observă prezenţa numelui clasei şi a operatorului de rezoluţie pentru a specifica faptul că *zz*, *ll* şi *aa* sunt date membru ale clasei *dc*. În absenţa numelui clasei şi a operatorului de rezoluţie, datele *zz*, *ll*, *aa* devin date globale iniţializate cu valorile respective. Atribuiri de forma:

```
dc :: zz = 1;    dc :: ll = 1;     dc :: a = 1600;
```

sunt posibile, dar numai dacă sunt scrise în corpul unei funcţii membru (datele respective sunt protejate). Astfel de atribuiri, de obicei, nu sunt considerate a fi iniţializări. Ele se realizează numai dacă funcţia membru care le conţine este apelată.

Dacă datele membru statice nu se iniţializează, ele nu trebuie definite în modul indicat mai sus. Ele sunt alocate în mod automat şi au valoarea iniţială egală cu zero.

12.8. Constructori

Datele de tip predefinit sau definit de utilizator se alocă în mod automat, în conformitate cu declaraţia sau definiţia acestora. Odată cu alocarea datelor se pot face şi iniţializări. În cazul obiectelor, acestea se alocă la instanţierea lor. De asemenea, obiectele pot fi iniţializate la instanţiere. În acest scop, utilizatorul poate defini constructori, care sunt funcţii membru de acelaşi nume cu numele clasei. Se pot defini mai mulţi constructori pentru o clasă. În acest caz, ei sunt funcţii supraîncărcate şi deci ei diferă prin numărul şi/sau tipurile parametrilor.

Valorile de iniţializare se transferă constructorului şi ele joacă acelaşi rol ca parametrii efectivi de la apelurile funcţiilor obişnuite. Ele formează o listă care se include între paranteze rotunde şi sunt prezente în declaraţia (definiţia) obiectelor. În felul acesta, o declaraţie sau definiţie de obiect poate avea formatul:

```
nume_clasa nume_obiect(lista);
```

unde: *lista* - este formată dintr-o expresie sau mai multe, separate prin virgule.

Lista, împreună cu parantezele care o includ, sunt absente dacă obiectul nu se iniţializează sau dacă există un constructor cu toţi parametri impliciţi.

Exemplu: Se consideră clasa *complex* definită ca mai jos:

```
class complex {
double real;   double imag;
public:
complex(double x=0,double y=0) {     / * constructor pentru numere complexe; implicit se instantiază numărul complex cu
ambele părţi egale cu zero */
real = x;   imag = y;
}   …
};
```

Exemple de instanţieri ale clasei *complex*:

```
complex z;     //se intanţiază numărul complex iniţializat implicit: z = 0+0*i
complex r(3); //se instantiază numărul complex r = 3+0*i
complex i(0,1);    //se instanţiază numărul complex i = 0+1*i
complex c(1.5, –1.5);     // se instanţiază numărul complex c = 1.5–1.5*i
```

La iniţializare, se utilizează regulile de la apelurile funcţiilor supraîncărcate dacă există mai mulţi constructori. Dacă există un singur constructor, atunci se aplică regula de la apelurile funcţiilor din limbajul C, adică parametrii efectivi (în cazul de faţă expresiile prin care se face iniţializarea) se convertesc spre tipurile parametrilor forrriali corespunzători ai constructorului. Această regulă se aplică şi în cazul clasei *complex*, care are un singur constructor. Mai jos, dăm un exemplu de clasă cu mai mulţi constructori:

```
class dc {
int zi,luna,an;
public:
dc ( ) {     // constructor fără parametri
zi = 1; luna = 1;   an = 1600;
}
dc(int z,int l,int a=1995) {   // constructor cu trei paramaetri de tip int
zi = z; luna = l;    an = a;
}
dc(int z,char *denl, int a);    //constructor pentru iniţializare cu denumirea lunii
…                               // calendaristice
}
```

Exemple de instanţieri a1e clasei *dc*:

```
dc d1; //se apelează constructorul fără parametri d1: zi=1, luna=1, an=l600
dc d2 (15,9,1995);    // se apelează constructorul cu toţi parametri de tip int
                      //d2: zi=15, luna=9, an=1995
dc d3 (15,9);  // se apelează acelaşi constructor ca la d2 şi se obţine acelaşi rezultat
```

dc d4(15,"septernbrie",1997); // se apelează constructorul cu parametrul char *denl

dc *pd = new dc(15,9); /* se apelează constructorul ca şi în cazul obiectelor d2 şi d3; obiectul se aloca în memoria heap, datele membru se inițializează ca la obiectele d2 şi d3; pd are ca valoare adresa de început a obiectului alocat în memoria heap */

În acest exemplu s-a definit un constructor fără parametri. Un astfel de constructor se numeşte **constructor implicit.**

În cazul în care există un constructor implicit nu se mai poate defini, pentru clasa respectivă, un constructor cu toți parametrii impliciți. Într-adevar, un astfel de constructor conduce la ambiguitate la instanțierea obiectelor. De exemplu, dacă alături de constructorul *dc()* al clasei *dc*, am defini constructorul:

dc(int z=1,int l=1,int a=1600)

atunci instanțierea: *dc d;* este ambiguă, deoarece se pot apela ambii constructori.

Prezența constructorilor nu este obligatorie. Se pot defini clase şi fără constructori. În acest caz, compilatorul C++ generează în mod automat un constructor fără parametri, adică un constructor implicit. Acesta are rol numai de alocare a obiectelor clasei respective. Constructorii definiți de programator sunt necesari numai în cazul în care se doreşte inițializarea obiectelor la instanțierea lor.

În cazul în care o clasă are cel puțin un constructor şi nici unul nu este constructorul implicit, atunci nu se pot instanția obiecte neinițializate. Aceasta, deoarece compilatorul nu creează constructorul implicit pentru clasele care au cel puțin un constructor. Exemplu:

```
class complex {
double real;   double imag;
    public:
complex(double x,double y) {
     real = x;   imag = y;
} …
    }
```

În acest caz, se pot instanția numai obiecte cu ambele părți inițializate:

```
complex z(1,2);
complex z 1;  //eroare: nu există constructor implicit
```

Declarația (definiția) obiectelor pentru care lista de inițializare se reduce la un singur parametru, poate fi scrisă într-un format care să nu difere de cel utilizat la inițializarea variabilelor simple. Astfel:

nume_clasa nume_obiect = expresie;

realizează instanțierea obiectului *nume_obiect* al clasei *nume_clasa*, instanțiere la care se apelează un constructor pentru care primul parametru are ca valoare, valoarea lui *expresie*. Alți parametri, sau nu există la constructorul apelat sau sunt parametri impliciți. Exemplu:

```
class complex {
double real;   double imag;
    public:
complex(double x=0,double y=0) {
    real = x;   imag = y;
} …
    };
```

Exemple de instanțieri:

```
complex z;          //z = 0+0*i          complex z1(1);    //z1 = 1+0*i
complex z2 = 1;  //z2 = 1+0*i          complex z3(1,2); //z3 = 1+2*i
```

În cazul în care dorim să instanțiem obiecte atît inițializate, cît şi neinițializate, putem folosi un constructnr implicit vid, care se va apela la instanțierea obiectelor neinițializate. Exemplu:

```
class dc {
int zi, luna, an;
    public:
dc(){};//constructor implicit vid. Se utilizează pentru instanțierea obiectelor neinițializate
    dc(int z,int l,int a) {  // constructor utilizat pentru inițializarea obiectelor
    zi = z;      luna = l;        an = a;
    } …
    }
```

Se pot utiliza instanțieri de forma:

```
dc d;   // se apelează constructorul implicit
dc d1(1,2,1997); // se instantiază ohiectul d1 inițializat astfel: zi=1, luna=2, an=1997
```

Parametrii unui constructor pot fi de orice tip, cu excepția tipului definit de clasa pentru care este funcție membru. Deci, dacă clasa este numele unei clase, atunci nu se poate defini un constructor de antet:

clasa(clasa p)

În schimb, constructorul unei clase poate avea ca parametri pointeri sau referințe la obiectele clasei respective. Deci:

clasa(clasa *p) şi clasa(clasa& p)

sunt antete corecte de constructori. Constructorul:

clasa(const clasa& p)

este un constructor special care permite copierea obiectelor. El se numeşte **constructor de copiere**. Constructorul de copiere poate avea şi alți parametri care însă trebuie să fie impliciți. Constructorul de copiere se apelează într-o instanțiere de felul celei de mai jos:

clasa c(...); //instanțiere cu inițializare obişnuită
clasa c1=c; //se apelează constructorul de copiere; c1 este o copie a lui c

Dacă programatorul nu defineşte un constructor de copiere, atunci compilatorul generează un constructor de copiere implicit, dacă este nevoie de el.

Exemplu: Mai jos, definim un constructor de copiere pentru numerele complexe.

```
class complex {
double real;  double imag;
public:
complex(double x=0,double y=0) {
real = x;  imag = y;
}
complex(const complex& c) {  // constructor de copiere
real = c.real;  imag = c.imag;
} …
}
```

Exemple de instanțieri:

```
complex z(1,2);  //z = 1+2*i
complex z1=z;  // se apelează constructorul de copiere z1 = 1+2*i
complex z2(z);  // se apelează constructorul de copiere z2 = 1+2*i
```

Datele membru ale unei clase pot fi date arbitrare dar nu obiecte ale clasei respective. În particular, datele membru pot fi obiecte ale unei alte clase. Un exemplu simplu este cel oferit de primitivele care se utilizează la definirea figurilor pe ecranul setat în mod grafic.

Considerăm clasa *Punct* care defineşte un punct pe ecran. Acesta are o poziție definită prin coordonatele sale (coloana şi linia în care se afişează punctul respectiv). Definim clasa *Punct* ca mai jos:

```
class Punct {
int x;  //coloana  int y; // linia
public:
Punct(int col=0,int linia = 0) {  //constructor: punctul implicit este cel de coordonate (0,0)
x = col;  y = linia;
} …
}
```

Un dreptunghi se poate trasa dacă se definesc 2 vîrfuri diagonal opuse. De obicei, se consideră vîrful din stînga sus al dreptunghiului şi cel din dreapta jos. Un vîrf al dreptunghiului este un obiect al clasei *Punct*. De aceea, putem defini clasa *Dreptunghi* cu ajutorul a 2 obiecte ale clasei *Punct*:

```
class Dreptunghi {
Punct st_sus;  Punct dr_jos;
}
```

Constructorul clasei *Dreptunghi* trebuie să transfere valori pentru parametrii constructorului clasei *Punct*. Acest lucru se realizează modificînd antetul constructorului, ca mai jos. Fie clasa cl definita astfel:

```
class cl {
cl1 c1;  cl2 c2;  …  cln cn;
};
```

unde: *cl1,cl2,...,cln* - sunt nume de clase, în prealabil definite, care nu neapărat sunt toate distincte. Ele, au fiecare, constructori pentru inițializarea obiectelor.

Constructorul *cl* transferă valorile de inițializare pentru obiectele membru *c1,c2,...,cn* prin antetul său de format:

cl(...): c1(...),c2(...),...,cn(...)

În acest antet vor lipsi obiectele pentru care nu se transferă date de inițializare.

Reluind exemplul de mai sus, completăm definiția clasei *Dreptunghi* cu 2 constructori:

```
class Dreptunghi {
Punct st_sus;  Punct dr_jos;
public:
Dreptunghi(int dr,int j): dr_jos(dr,j) {  }  //se instanțiază dreptunghiul cu vîrfurile:
                    // stînga sus: (0,0) - implicit;  dreapta jos: (dr,j)
Dreptunghi(int st, int sus, int dr, int jos): st_sus(st,sus),dz jos(dr,jos) {  }
// se instanțiază dreptunghiul cu vîrfurile: stînga sus: (st,sus) ; dreapta jos: (dr,jos)
… };
```

Metoda de inițializare a obiectelor, care sunt date membru ale unei clase, poate fi utilizată şi pentru date membru care nu sunt obiecte. Astfel, un constructor de forma:

```
complex(double x,double y) {
real = x;  imag = y;
}
```

poate fi scris şi sub forma:

```
complex(double x,double y):real(x),imag(y) {      }
```

Aşadar, constructorii nu returnează nici o valoare la revenirea din ei. Mai mult decît atît, antetul lor constituie o excepție, deoarece nu este admis cuvîntul cheie *void*, cuvînt care trebuie să fie prezent în antetul funcțiilor care nu returnează o valoare. În cazul în care o clasă are obiecte membru care au constructori, aceştia se apelează înainte de a se apela constructorul clasei respective. Obiectele claselor care au cel puțin un constructor nu pot fi componente ale unei reuniuni. Spre deosebire de funcțiile obişnuite, adresa constructorului nu se poate determina. De obicei, constructorii sunt funcții membru publice, dar nu se pot apela explicit la fel ca celelalte funcții membru. Un constructor poate fi apelat explicit pentru situații de felul celui de mai jos:

```
complex z = complex(1,2);
```

În acest caz a fost apelat constructorul clasei compler pentru a crea un obiect anonim care este inițializat cu valorile 1 pentru partea reală şi 2 pentru partea imaginară. Apoi, obiectul respectiv este copiat în obiectul *z*.

12.9. Destructor

Destructorii sunt funcții care pot fi considerați că acționează în sens invers față de constructori. Ei au multe caracteristici în comun cu constructorii, dar între ei există şi diferențe. Nurnele unui destructor este numele clasei precedat de caracterul "~". Un destructor nu are parametri şi el este unic pentru o clasă. Dacă programatorul nu a definit un destructor, atunci compilatorul generează un destructor pentru clasa respectivă. Antetul destructorilor nu conține cuvîntul *void*, deşi ei nu returnează valori. De aceea, destructorii au antetul:

```
~nume_clasa()
```

în interiorul definiției clasei *nume_clasa*.

În afara definiției clasei, antetul destructorului va fi:

```
nume_clasa:: ~nume_clasa()
```

Ca şi în cazul constructorilor, adresa destructorului nu poate fi determinată.

Obiectele unei clase care au un destructor şi/sau cel puțin un constructor nu se pot utiliza ca şi componente ale unei reuniuni. Dacă la instanțierea unui obiect s-au apelat mai mulți constructori, atunci la distrugerea lui, destructorii se vor apela în ordine inversă.

Pentru un obiect global, destructorul se apelează ca parte a procedurii *exit* de la terminarea execuției funcției *main*. De aceea, un astfel de destructor nu trebuie să apeleze funcția *exit* deoarece se intră într-un ciclu infinit.

Pentru un obiect local, destructorul se apelează cînd controlul programului iese din domeniul lui (se iese din blocul în care este declarat).

Obiectele dinamice nu pot fi distruse automat. Distrugerea se realizează de către programator deoarece numai el ştie cînd un astfel de obiect nu mai este necesar.

Destructorul poate fi apelat de programator atît direct, cît şi indirect prin intermediul operatorului *delete*. Noi utilizăm operatorul *delete* pentru a distruge obiecte create dinamic cu ajutorul operatorului *new* (obiecte dinamice). Exemple:

1. Fie tipul *complex* definit în exemplele precedente.

```
…complex *pz;            //în memoria heap se construieşte un obiect de tip
…                        // complex cu partea reală egala cu 1 şi partea imaginară
pz = new complex(1,2); // egală cu 2; pz pointează spre obiectul respectiv
delete pz;               //se distruge obiectul creat mai sus prin new
```

2. Se defineşte clasa *string* ca mai jos:

```
class string {
char *sir;  int lung;
public:
string(char *);     ~string();  …
};
string :: string (char *s ) {    // definiția constructorului
lung = strlen(s) + 1;  sir = new char[lung];      strcpy(air,s);
}
string :: ~string() {         // definiția destructorului
delete sir;
}
```

Exemple de instanțieri:

```
string sir_de_caractere("Acesta este un sir de caractere");
//se creează obiectul sir_de_caractere în memoria heap şi se inițializează cu textul:
// "Acesta este un şir de caractere"
…
delete sir_de_caractere; //se distruge obiectul apelîndu-se indirect destructorul
```

Apelul direct al destructorului se poate face numai daca numele lui este precedat de numele clasei care este urmat de operatorul de rezoluție:

```
…
string s("exemplu");
…
s.string:: string ( );    // apel direct al destructorului
```

Exerciţii: 22.1 Să se definească tipul abstract de date *complex* care să aibă funcţii membru pentru modul, argument, pentru accesul la partea reală şi imaginară a obiectelor de tip *complex*, pentru afişarea obiectelor complexe, precum şi constructori pentru iniţializare şi copiere.

```
// FIŞIERUL BXXII1.H
class complex {  // date membru protejate(private)
double real;  double imag;
public:  // funcţii membru neprotejate
complex(douhle x=0, double y=0); // constructor folosit la initializare
complex(const complex&);  // constructor de copiere
double modul ( );  // modulul numărului complex
double arg ( );  //argumentul numărului complex
double retreal ();  // returnează partea reală
double retimag ( );  //returnează partea imaginară
void afiscomplex();  //afişează numărul complex
}; //sfîrşit definitie clasa complex
// FIŞIERUL BXXII1
#ifndef __BXXII1_H  #include "BXXIIl.H"  #define __BXXII1  #endif
#ifndef __MATH_H  #include <math.h>  #define __MATH.H #endif
#ifndef __PI  #define PI 3.14159265358979  #define __PI  #endif
inline complex :: complex(double x,double y) {// constructor pentru initializarea obiectelor
real = x;  imag = y;
}
inline complex :: complex(const complex& z) {  //constructor de copiere
real = z.real;  imag = z.imag;
}
inline double complex :: modul() {  //calculează modulul numărului complex
return sqrt(real*real+imag*imag);
}
double complex :: arg() {  //calculează argumentul numărului complex
double a;
if(rea1==0 && imag==0)  return 0.0;
if(imag==0)
if(real > 0)  return 0.0;
else  return PI;
if(rea1==0)
if(imag > 0)  return PI/2;
else  return (3*PI)/2;
a = atan(imag/real);  // real != 0 şi imag != 0
if (real < 0)  return a+PI;  //rea1 < O şi y!=0
else  //real > 0
if (imag < 0)  return 2*PI+a;  //real >0 şi imag < 0
return a;  //real >0 şi imag > 0
}  //sfîrşit arg
inline double complex :: retreal() { // returnează partea reală a numărului complex
return real;
}
inline double complex :: retimag() {  //returnează partea imaginară a numărului complex
return imag;
}
inline woid complex :: afiscomplex() {// afişează numărul complex
printf("%g+i*(%g)ln",real,imag);
}
```

22.2 Să se scrie un program care realizează următoarele: Citeşte perechi de numere care reprezintă, fiecare, partea reală şi respectiv partea imaginară a unui număr complex; afişează: - numărul complex citit; - rădăcina pătrată din fiecare număr complex citit; - suma numerelor complexe citite.

```
// PROGRAMUL BXXII2
#include <stdio.h>  #include <conio.h>  #include "BXXII1.CPP"
main() {  /* - citeşte perechi de numere care reprezintă partea reală şi respectiv imaginară a unui număr complex; Afişează: -
numărul complex citit; - rădăcina pătrată din numărul complex citit; - suma numerelor complexe citite. */
double x;  double y;  double sx = 0;  double sy = 0;
do {
printf("partea reala =");  if(scanf("%lf",&x) != 1) break; // nu mai sunt numere
printf("partea imaginară =");
if (scanf("%lf",&y) != 1) {  //eroare
printf("partea imaginară eronata\n");  printf("se reia citirea numărului\n");
```

```
            fflush(stdin);  //videază zona tampon de la intrarea standard
            continue;       //se reia ciclul de citire
        }
        // se însumează partea reală şi cea imaginară a numerelor complexe citite
        sx += x;    sy += y;
         complex z (x, y);//se construieşte numărul complex avînd părţile citite; se apelează
                          //constructorul care realizează iniţializarea obiectului z: z=x+i*y
        z.afiscomplex( ); // afişează numărul citit
        double m = z.modul ();  // calculează modulul lui z
        double a=z.arg(); //calculează argumentul numărului complex
        double m1 = sqrt(m);     //rădăcina pătrată din modulul z
        double a1 = a/2;  //semiargumentul numărului z
        printf("modulul = %g\t argumentul = %g\n",m,a);
        double preal = m1+cos(a1);      double pimag = m1*sin(a1);
        complex rz1 = complex(preal,pimag);       // rz1 = sqrt(z)
        complex rz2 = complex(–preal,–pimag);   // rz2 = sqrt(z);
        printf("radacina patrata\n");      printf("sqrt(z)1 = ");
        rz1.afiscomplex();                 printf("sqrt(z)2 = ");
        rz2.afiscomplex();
    } while (1);
    complex sz = complex(sx, sy);  //sz = suma
    printf("suma numerelor complexe citite = ");
    sz. afiscomplex();     //afişează suma numerelor complexe citite
        }
```

22.3 Să se extindă tipul abstract *complex*, definit în exerciţiul 22.1, aşa încît să se poată realiza următoarele operaţii asupra obiectelor complexe: adunare; scădere; negativare; înmulţire; împărţire; citirea de la intrarea standard a componentelor unui obiect complex.

```
        //   FIŞIERUL BXXII3.H
        enum Boolean (false,true);
        class complex {
        double real;    double imag;  //date membru protejate(private)
        public:     // functii membru neprotejate
    complex(double x=0,double y=0);  //constfttctor
    complex(const complex&);          //constructor de copiere
    double modul( );                  //modulul numărului complex
    double arg( );                    //argumentul numărului complex
    double retreal ( );               //returnează partea reala
    double retimag ();                // returnează partea imaginara
    void afiscomplex( );              //afisează numărul complex
            // functii membru noi
        Boolean citcomplex();//citeşte componentele numărului complex; Returnează false la EOF
    void adcomplex(complex *z1,complex *z2);  //calculează z = z1+z2
    void sccomplex(complex *z1,compIex *z2);  // calculează z = z1–z2
    void negcomplex(complex *z1);             // calculează z = –z1
    void mulcomplex(complex *z1,complex *z2);     //calculează z = z1*z2
    Boolean divcomplex(complex *zl,complex *z2); //calculeam z = z1/z2; returnează false
                    //   la împărţirea cu zero
        };
```

Fisierul de mai jos, conţine definiţiile funcţiilor membru noi. Definiţiile funcţiilor membru vechi se preiau din fişierul BXXII1.CPP.

```
        //   FlŞIERUL BXXII3
        #ifndef __BXXII3_H     #include "BXXII3.H"     #define __BXXII3_H     #endif
        #define __BXXII1_H
        #ifndef __BX18_CPP     #include "BXl8.CPP"     #define __BX18_CPP     #endif
        #ifndef __STDIO_H #include <stdio.h>         #define __STDIO_H      #endif
        //se defineşte BXXII1_H pentru a nu mai include
        //fişierul BXXII1.H prin includerea fişierului BXXII1.CPP
        #include "BXXII1.CPP" //se preiau definiţiile funcţiilor membru vechi
            //definitiile functiilor membru noi
        Boolean complex :: citcomplex() { /* - citeşte componentele numărului complex.
                                    - returneaza: false la sfirsitul fisierului; true altfel. */
    double preal;  double pimag;
    if(pcit_double("Partea reala:",&preal) == 0)    return false;    //EOF
    if(pcit_double("Partea imaginara: ",&pimag) == 0) return false; // EOF
    real = preal;    imag = pimag;     return true;
```

```
    }
    inline void complex :: adcomplex(complex *z1, complex *z2) {     // calculează z = z1+z2
real = z1–>real + z2–>real;  imag = z1–>imag + z2–>imag;
    }
    inline void complex :: sccomplex(complex *z1, complex +z2) {     //calculează z = z1–z2
real = z1–>real – z2–>real;  imag = z1–>imaq – z2–>imag;
    }
    inline void complex negcomplex(complex *z1) {      // calculează z = –z1
real = –z1–>real;      imag = –z1–>imag;
    }
    inline void complex :: mvlccmplex(complex *z1, complex *z2) {   // calculează z = z1*z2
    real = z1–>real * z2–>real – z1–>imag * z2–> imaq;
imag = z1–>real * z2–>imag + z1–>imag * z2–>real;
    }
    Boolean compiex :: divcomplex(complex *z1, complex *z2) {       // calculează z = z1/z2
double d = z2–>real * z2–>real + z2–>imag * z2–>imag;
if (d == 0) return false;       //divizor nul
    real = (z1–>real * z2–>real + z1–>imag * z2–>imag)/d;
imag = (z1–>imag * z2–>real – z1–>real * z2–>imag)/d;
return true;
    }
```

22.4 Să se scrie un program care citeşte numerele complexe *a*, *b*, *c*, rezolvă şi afişează rădăcinile ecuaţiei de gradul 2: a*x*x+b*x+c=0.

```
    //   PROGRAMUL BXXII4
    #include <stdlib.h>
    #ifndef __STDIO_H     #include <stdio.h>    #define __STDIO_H     #endif
    #include "BXXII3.CPP"
    main() {    /* - citeşte: a,b,c. Rezolvă ecuaţia: a*x*x+b*x+c=0. Afişează soluţiile ecuaţiei
complex a,b,c;     char er[] = "s-a tastat EOF\n";
    if(a.citcomplex() == false) {     printf(er);      exit(1);     }
    if(b.citcomplex() == false) {     printf(er);      exit(1);     }
    if(c.citcomplex() == false) {     printf(er);      exit(1);     }
if(a.retreal() == 0 && a.retimag() == 0 && b.retreal() == 0 && b.retimag() == 0 &&
    c.retreal() == 0 && c.retimag() == 0) {
        printf("ecuaţie nedeterminată\n");   exit(0);
    }
if(a.retreal() == 0 && a.retimag() == 0 && b.retreal() == 0 && b.retimag() == 0) {
        printf("ecuaţia nu are soluţie\n");    exit(1);
}
if(a.retreal() = 0 && a.retimag() == 0) {
    printf ("ecuaţie de gradul 1\n");
    conplex x,z;
    z.divcomplex(&c,&b);        x.negcomplex(&z);   //x = –z
    printf("x=");  x.afiscomplex();  exit(0);
} // obiectele x şi z se distrug în acest punct
    //ecuaţie de gradul 2
complex bp;   bp.mulcomplex(hb,ab);  //bp = b*b
complex patru(4,0);       complex patrua;  complex patruac; complex deltap;
    patrua.mulcomplex(&a,&patru);
patruac.mulcomplex(apatrua,hc);
deltap.sccomplex(&bp,&patruac);  //b*b–4*a*c
double r = deltap.modul();                r = sqrt(r);
double argument = deltap.arg();           argument = argument/2;
complex delta(r*cos(argument),r*sin(argument));     //delta = sqrt(b*b–4*a*c)
complex mb;       mb.negcomplex(&b);          //mb = –b
complex doia;     doia,adcomplex(&a,&a);      //doia = a+a
complex x;        x.adcomplex(&mb,&delta); //x = –b+delta
complex x1;       xl.divcomplex(&x,&doia);   //x1 = (–b+delta)/(2*a)
printf("x1 = ");    x1.afiscomplex();     //afişează rădăcina x1
complex x2;       x.sccomplex(&mb,&delta); //x = –b–delta
x2.divcomplex(&x,&doia); //x2 = (–b–delta)/(2*a)
printf("x2 = ");    x2.afiscomplex();     //afişează rădăcina x2
    }
```

22.5 Să se definească tipul abstract *dc* pentru implementarea datei calendaristice. În acest scop se defineşte clasa *dc* care are următoarele componente de tip int: - *zi*, *luna*, *an*; - *minzz*, *minll*, *minaa* - pentru data calendaristică considerată ca dată

validă minimă; - *maxzz, maxll, maxaa* - pentru data calendaristică considerată ca dată validă maximă; - *tnrz* - este un tablou ale cărui elemente definesc numărul de zile ale lunilor calendaristice (luna februarie se consideră că are 28 de zile). Componentele *minzz, minll, minaa, maxzz, marll, maraa* şi *tnrz* nu se multiplică la fiecare instanţiere a clasei *dc*, ele fiind utilizate în comun de către funcţiile membru. De aceea, ele se definesc ca date membru statice. În mod implicit, data calendaristică minimă se consideră 1 ianuarie 1600, iar cea maximă se consideră 31 decembrie 4900.

Clasa *dc* are 3 funcţii pentru verificarea corectitudinii datelor calendaristice. Una este o funcţie membru obişnuită care verifică data calendaristică a obiectului curent, iar celelalte 2 verifică corectitudinea datelor calendaristice definite de datele membru statice *minzz, minll, minaa, maxzz, marll*, şi *maxaa*. Aceste 2 funcţii sunt funcţii membru statice. Aceste 3 funcţii apelează o funcţie membru statică *v_calend* care validează o dată calendaristică definită prin parametrii ei. Faptul că o dată calendaristică este dintr-un an bisect sau nu, se determină folosind trei funcţii, una pentru obiectele de tip *dc* şi 2 pentru cele două date membru statice. Aceste funcţii folosesc în comun o funcţie *bisect* care stabileşte dacă data calendaristică definită de parametri ei este o dată dintr-un an bisect sau nu. Funcţia *bisect* este o funcţie obişnuită. Ea se defineşte în acelaşi fişier cu funcţiile membru ale clasei.

Clasa are un constructor implicit care se utilizează la instanţierea obiectelor fără iniţializarea datei calendaristice nestatice (zi, luna, şi an).

Datele statice se iniţializează independent, în afara constructorilor. Constructorii clasei *dc* vor verifica corectitudinea datelor calendaristice statice. În caz de eroare, se afişează un mesaj corespunzător şi se forţează datele implicite.

Pentru instanţierea obiectelor iniţializate se utilizează un constructor cu trei parametri pentru cele 3 date membru care nu sunt statice (zi, luna şi an). Constructorul utilizat în acest scop, verifică atît datele statice, cît şi cele care se iniţializează prin constructorul respectiv. În caz de eroare, se afişează un mesaj corespunzător şi se instanţiază un obiect cu data calendaristică minimă. Alte funcţii membru:

- constructor de copiere;
- *retzi*: returnează ziua din data obiectului curent;
- *retluna*: returnează luna din data obiectului curent;
- *retan*: returnează anul din data obiectului curent;
- *afisdata*: afişează data calendaristică a obiectului curent;
- *datamin*: returnează data minimă;
- *datamax*: returnează data maximă;
- *modifmin*: modifică data minimă;
- *modifmax:* modifică data maximă;
- *citdata:* citeşte o dată calendaristică;
- *adzi*: adună un număr de zile la data obiectului curent;
- *difdata*: determină diferenţa, în zile, dintre 2 date calendaristice;
- *zi_din_an*: returnează ziua din an pentru data obiectului curent;
- *ziua_si_luna*: determină luna şi ziua din lună din parametrii *an* şi *ziua* din *an*;
- *nr_zile_luna*: returnează numărul de zile din luna calendaristică a obiectului curent;
- *verif_min_max*: verifică datele minimă şi maximă; dacă o dată este eronată, se dă un mesaj de eroare şi se forţează data implicită: 1 ianuarie 1600 pentru dată minimă şi 31 decembrie 4900 pentru data maximă.

```
// FIŞIERUL BXXII5.H
#ifndef __Boolean  #define __Boolean  enum Boolean(false,true);  #endif
class dc {
int zi,luna,an;     static int minzz,minll,ninaa;          //date membru protejate
static int maxzz,maxll,maxaa;  static int tnrz[13];
static char *tdenluna[13];     static char *ermin;
static char *ermax;            static char *erdc;
static void afiser(char *sir);     // afisează textul spre care pointează sir
static void verif_min_max();       // verifică datele minimă şi maximă; la eroare se
                                   //dă mesaj şi se fortează dată implicita corespunzatoare
public:
Boolean valid_dc();  / * verifică corectitudinea datei calendaristice a obiectului curent; la eroare returnează false */
static Boolean valid_dc_min(); /* verifică corectitudinea datei minime;
                                  la eroare returnează false */
static Boolean valid_dc_max(); /* verifică corectitudinea datei maxime;
                                  la eroare returnează false */
Boolean bisect dc();  /* returneaza: true - dacă data obiectului curent este dintr-un
                         an bisect; false - altfel. */
static Boolean bisect_dc_min();    /* returneaza: true - dacă data minimă este dintr-un
                         an bisect; false - altfel */
static Boolean bisect_dc_max();    /* returneaza: true - dacă data maximă este dintr-un
                         an bisect; false - altfel. */
static Boolean v_calend(int z,int l,int a); /* returneaza: true - dacă data definită de
                         parametrii z, l şi a este validă; false - altfel. */
dc ( );  //constructor implicit pentru obiecte neiniţializate
dc(int z,int l,int a);   //constructor pentru iniţializarea obiectelor
dc (const dc& );         //constructor pentru copiere
```

```
    int retzi( );              //returnează ziua din data obiectului curent
    int retluna( );            // returnează luna din data obiectului curent
        int retan( );              //returnează anul din data obiectului curent
    void afisdata( );          //afişează data obiectului curent
        dc *datamin( );    // returnează un pointer spre data minimă
        dc *datamax( );    // returnează un pointer spre data maximă
        void modifmin( );      // schimbă data calendaristică minimă cu data obiectului curent
    void modifmax( );      // schimbă data calendaristică maximă cu data obiectului curent
    int citdată ( ); // citeşte o dată calendaristică; returnează 0 la sfîrşit de fisier şi 1 altfel
        Boolean adzi(long z);      // adună, la data obiectului curent, numărul de zile dat de z
    long difdata(dc *d);  /* returnează diferența, în număr de zile, dintre data calendaristică a obiectului curent şi cea spre care
                             pointează d */
        char *denluna( ); /* returnează un pointer spre denumirea lunii datei obiectului curent
    int zi_din_an();    // returnează ziua din an pentru data obiectului curent
    Boolean ziua_si_luna(int z,int a);   /* determină data calendaristică a obiectului curent
                                            din z - ziua din an şi din a - anul calendaristic */
    int nr_zile_luna();     /*returnează numărul de zile din luna calendaristică a
                              datei obiectului curent */
        }   //sfîrşit definiția clasei dc
        Definițiile funcțiilor membru şi inițializarea datelor statice se dau într-un fişier de extensie .CPP.
        //   FIŞIERUL BXXII5
        #ifndef __STDIO_H      #include <stdio.h>         #define __STDIO_H      #endif
        #ifndef __Boolean         enum Boolean(false,true);    #define __Boolean    #endif
        #ifndef __DC_H       #include "BXXII5.H"          #define __DC_H          #endif
         //inițializarea datelor statice
        int dc::minzz=1;  int dc::minll=1; //implicit, data calendaristică minimă este 1 ianuarie 1600
        int dc::minaa=1600; int dc::maxzz=31;      // implicit, data calendaristică maximă este
        int dc::maxll=12; int dc::maxaa=4900;      //31 decembrie 4900
         //inițializarea tabloului cu numărul zilelor din lunile calendaristice
        int dc::tnrz[13] = {0,31,28,31,30,31,30,31,31,30,31,30,31};
         //initializarea textelor de eroare
        char *dc::erdc = "dată calendaristică eronată\n";
        char *dc::ermin = "dată minimă eronată\n";    char *dc::ermax = "dată maximă eronată\n";
        // inițializarea tabloului cu denumirile lunilor calendaristice
        char *dc::tdenluna[13] = {"lună ilegală", "ianuarie", "februarie", "martie", "aprilie", "mai",
        "iunie", "iulie", "august", "septembrie", "octombrie", "noiembrie", "decembrie"};
        //funcție obişnuită folosită de funcțiile membru
        inline int bisect(int a) {   // returnează: 1 dacă a defineşte un an bisect; 0 altfel.
        return a%4==0 && a%100 || a%400 == 0;
        }
        // funcții membru
        Boolean dc::v_calend(int z,int l,int a) {      /* returneaza: true dacă data calendaristică
            definită de parametrii z, l şi a este validă; false în caz contrar. */
    if(a < 1600 || a > 4900) return false;
    if(a < dc::minaa || a > dc::maxaa) return false;
    if(l < 1 || l >12)    return false;
    if(a==dc::minaa && l<dc::minll || a==dc::maxaa && l>dc::maxll) return false;
    if(z <1 || z>tnrz[1]+(l==2 && bisect(a))) return false;
            if(a==dc::minaa && l==dc::minll && z<dc::minzz || a==dc::maxaa && l==dc::maxll && z > dc::maxzz) return
    false;
    return true;
        }
        inline void dc::afiser(char *sir) {    //afisează textul spre care pointează sir
    printf("%s\n",sir);
        }
inline Boolean dc::valid_dc() { /* - verifică corectitudinea datei caiendaristice a obiectului curent; - la eroare returnează false.
        */
    return v_calend(zi,luna,an);
        }
        inline Boolean dc::valid dc_min() {// - verifică corectitudinea datei minime; la eroare false.
    return v_calend(dc::minzz,dc::minll,dc::minaa);
        }
        inline Boolean dc::valid dc_max() {// - verifică corectitudinea datei maxime; la eroare false.
    return v_calend(dc::maxzz,dc::maxll,dc::maxaa);
        }
```

```
        Boolean dc::bisect_dc ( ) {   /* returnează true dacă anul calendaristic al obieetului curent
        este bisect */
    if(bisect(an)) return true;
        else     return false;
        }
        Boolean dc::bisect_dc_min() { // returnează true dacă anul minim este bisect
        if(bisect(dc::minaa)) return true;
        else return false;
        }
        Boolean dc::bisect_dc_max() { // returnează true dacx anul maxim este bisect
    if(bisect(dc::maxaa)) return true;
        else return false;
        }
        void dc::verif_min_max() { /* - verifică datele minimă şi maximă; - dacă o dată este eronată se dă un mesaj de eroare şi
se forţează data implicită corespunzătoare. */
        if (dc::valid_dc_min() == false) {
            dc::afiser(dc::ermin);      // se forţează data minimă implicită
        dc::minzz = 1;     dc::minll = 1;      dc::minaa = 1600;
        }
        if (dc::valid_dc_max() == false) {
            dc::afiser(dc::ermax);          // se forţează data maximă implicită
        dc::maxzz = 31;   dc::maxll = 12;    dc::maxaa = 4900;
        }
        }
dc :: dc (int z, int l, int a) {  /* constructor pentru iniţializarea datelor. Se verifică data minimă, maximă şi cea care se
        instanţiază */
        dc :: verif_min_max();
        zi = z; luna = l;    an = a;      // iniţializarea obiectului care se instanţiază
        if (valid_dc() == false) {
            afiser(dc::erdc)    zi = dc::minzz;     luna = dc::minll;  an = dc::minaa;
        }
inline dc :: dc() { // constructor pentru obiecte neiniţializate
        dc :: verif_min_max();
}
        dc :: dc(const dc &d) {   // constructor de copiere
        zi = d.zi;   luna = d.luna;      an = d.an;
        if (valid_dc() == false) {
            afiser(dc::erdc)    zi = dc::minzz;     luna = dc::minll;  an = dc::minaa;
        }
        inline int dc :: retzi() {    // returnează ziua obiectului curent
        return zi;
        }
        inline int dc :: retluna() {      // returnează luna obiectului curent
        return luna;
        }
        inline int dc :: retan () {  // returnează anul obiectului curent
        return an;
        }
        inline void dc :: afisdata() { // afişează data obiectului curent
        printf("zi: %d\t luna: %d\t an: %d\n",zi,luna,an);
        }
        dc *dc :: datamin() { // returnează un pointer spre data minimă
        dc *d_min = new dc(dc::minzz, dc:: minll, dc::minaa);    return d_min;
        }
        dc *dc :: datamax() {// returnează un pointer spre data maximă
        dc *d_max = new dc(dc::maxzz, dc:: maxll, dc::maxaa); return d_max;
        }
        void dc::modifmin() {    // schimbă data calendaristică minimă cu data obiectului curent
    dc::minzz = zi;     dc::minll = luna;       dc::minaa = an;    dc::verif_min_max();
        }
        void dc::modifmax() {    //schimbă data calendaristică maximă cu data obiectului curent
        dc::maxzz = zi;    dc::maxll = luna;       dc::maxaa = an;   dc::verif_min_max();
        }
        irit dc::citdata() {// citeşte o dată calendaristică; returnează: 0 la sfirsit de fisier; 1 altfel.
        int i,c;
```

```
    for( ; ; ) {
    for( ; ; ) {
        printf("ziua: ");
        if((c=scanf("%d",&i)) == 1 && i > 0 && i <= 31)    break;
        if(c == EOF) return 0;
        printf("ziua eronată\n");       fflush(stdin);
    }   zi = i;
    for(;;) {
        printf("luna: ");
        if((c=scanf("%d",&i)) == 1 && i > 0 && i <= 12) break;
        if(c == EOF) return 0;
        printf("luna eronată\n");       fflush(stdin);
    }   luna=i;
    for(;;) {
        printf("anul: ");
        if((c=scanf("%d",&i)) == 1 6& i >= 1600 && i <= 4900) break;
        if(c == EOF) return 0;
        printf("anul eronat\n");        fflush(stdin);
    }   an=i;
    if(valid_dc() == true) break;
    printf("dată calendaristică: %d /%d/%d eronată\n", zi,luna,an);
    }   return 1;
    }
    int dc::zi_din_an() {  // returnează ziua din an pentru data obiectului curent
    int b = bisect(an);     int z=zi;
    for(int i=1;i < luna; i++)      z += dc::tnrz[i]+(i==2&&b);
return z;
    }
    Boolean dc::ziua_si_luna(int z, int a) {//determina data calendaristică din: z - ziua din an
    //şi din a - anul
int b = bisect(a);
    if (z > 365+b) {   //ziua din an eronată
    printf ("ziua = %d eronată\n",z);     return false;
    }
if(a < 1600 || a > 4900) {      //anul eronat
    printf("anul = %d eronat\n",a);       return false;
    }
int i = 1;
do {
    int j = dc::tnrz[i]+(i==2 && b);
    if(z <= j)  break;
    z –= j;      i++;
} while(1);
zi = z;      luna = i;    an = a;      return valid_dc();
    }
    inline char *dc::denluna() { //returnează un pointer spre denumirea lunii obiectului curent
    return luna<1 || luna>12 ? dc::tdenluna[0] : dc::tdenluna[luna];
    }
    Boolean dc:: adzi(long z) {  //adună la data obiectului curent numărul de zile dat de z
long totzile = zi_din_an();
totzile += z;   /* numărul total de zile care se consideră după 31 decembrie din anul precedent celui curent */
int ziledinan = 365 + bisect(an);
if(totzile >= 0)
while(totzile >= ziledinan) { // se socotesc anii
    totzile –= ziledinan;  an++;
    if( a <= 4900) ziledinan = 365+bisect(an);
    }
else    {
    an —;       ziledinan = 365 +bisect(an);
    while(–totzile >= ziledinan) {    // se socotesc anii
        totzile += ziledinan;       an —;
        if(an >= 1600)     ziledinan = 365+bisect(an);
    }
    totzile += ziledinan;  // determină numărul de zile din anul determinat
    }
```

```
    if(totzile == 0) {
        zi = 31;    luna = 12;      an—;        return valid_dc();
        }
    return ziua_si_luna(totzile,an);
        }
long dc::difdata(dc *d) {/* returnează diferența, în număr de zile, dintre data calendaristică curentă şi cea spre care pointează d
        */
    long zidinan1 = zi_din_an();     // numărul de zile din anul curent
    long zidinan2 = d–>zi_din_an();    //numărul de zile din anul definit de pointerul d
        long difzi = zidinan1 – zidinan2;    int an1,an2;
    if(an < d–>an) {
        an1 = an;  an2 = d–>an;
        }   else {
        an1 = d–>an;  an2 = an;
        }
    long difzian = 0;
    while(an1 < an2) {
        int b = bisect(an1);    difzian += 365+b;     an1++;
        }
    if(an < d–>an)     difzian = –difzian;
    return difzi + difzian;
        }
```

Observații:

1. Funcțiile membru statice, de obicei, se apelează calificînd numele lor cu numele clasei urmat de operatorul de rezoluție. Mai sus a fost apelată funcția membru statică *v_calend* fără a respecta acest lucru. Aceasta este posibil dacă nu există definită în acelaşi program încă o altă funcție cu acelaşi nume şi listă a parametrilor formali. Pentru a evita astfel de situații se recomandă să se respecte regua cu privire la calificarea numelor funcțiilor membru statice prin numele clasei urmat de operatorul de rezoluție.
2. Funcțiile membru care nu sunt statice se apelează numai în legătură cu un obiect al clasei pentru care ele sunt funcții membru. Apelurile unei astfel de funcții au forma:

```
        obiect.nume_funcție( ... )
    sau         pobiect–>nume_funcție( ... )
```

unde: *pobiect* - este un pointer spre tipul implementat prin clasa pentru care *nume_funcție* este funcție membru.

Mai sus, s-au apelat funcții membru nestatice ale clasei *dc* în corpul altor funcții nembru nestatice ale aceleeaşi clase fără a respecta regula de mai sus. De exemplu, în corpul funcției membru *citdata* se apelează funcția membru *valid_dc* astfel:

```
        if(valid_dc() == true) break;
```

Acesta este corect deoarece în mod implicit apelul de mai sus este realizat folosind pointerul *this*:

```
        if(this–>valid_dc() == true)     break;
```

22.6 Să se scrie un program care citeşte, de la intrarea standard, o succesiune de date calendaristice şi afişează fiecare dată calendaristică împreună cu cea a zilei următoare.

```
    //  PROGRAMUL BXXII6
    #ifndef __STDIO_H    #include <stdio.h>   #define __STDIO_H    #endif
    #ifndef __CONIO_H    #include <conio.h>   #define __CONIO_H    #endif
    #include "BXXII5.CPP"
main() {   /* citeşte o succesiune de date calendaristice şi afişează datele citite împreună cu data zilei urmatoare */
    for(;;) {
        dc data_citita;
        if(data_citita.citdata() == 0) break; //s-a întîlnit EOF
        data_citita.afisdata( );              //afisează dată citita
        dc data_urm = data_citita;            //copiere
        data.urm.adzi(1L);                    //determină data zilei următoare
        data_urm.afisdata( );                 //afişează data zilei următoare
        }   }
```

22.7 Să se scrie un program care realizează urmatoarele: citeşte date calendaristice care sunt situate între 2 date limită; afişează fiecare dată citită împreună cu diferența, în număr de zile, dintre data citită şi datele limită. Cele 2 date limită se definesc prin argumente în linia de comandă a programului. Data minimă este prima dată calendaristică din linia de comandă.

```
    //  ROGRAMUL BXXII7
    #ifndef __STDIO_H    #include <stdio.h>   #define __STDIO_H    #endif
    #ifndef __CONIO_H    #include <conio.h>   #define __CONIO_H    #endif
    #ifndef __STDLIB_H   #include <stdlib.h>  #define __STDLIB_H   #endif
    #include "bxxii5.cpp"
    main(int argc,char *argv[]) {/*-citeşte date calendaristice situate intre 2 date limita; afişează datele citite împreună cu
diferența, în număr de zile, dintre datele limită şi data citita */
    if(argc != 7) {
    printf("număr argumente = %d eronat\n",argc);    exit(1);
```

```
    }
int tabarg[6];
    for(int i=l;i < 7;i++) {
    char *p = argv[i];
    for(int j=0;*p;j++,p++)
        if(*p<'0' || *p>'9'){
            printf("argument eronat: %c\t%d\n",*p,*p);    exit(1);
    }
    if(j > 4) {
        printf("argument eronat: %s\n",argv[i]);    exit(1);
    }
    tabarg[i–1] = atoi(argv[i] );  //conversia argumentului
    }
dc liminf(tabarg[0],tabarg[1],tabarg[2]);    dc limsup(tabarg[3],tabarg[4],tabarg[5]);
liminf.afisdata();       //afişează limita inferioară
limsup.afisdata();       //afişează limita superioară
liminf.modifmin();    //defineşte data statică minimă
limsup.modifmax();  //defineşte data statică superioară
for(;;) {     //citirea datelor calendaristice
    dc data_crt;
    if(data_crt.citdata() == 0)     break; //s-a intilnit EOF
    data_crt.afisdata( );   //afişează data citită
    long difi,difs;
    difi = liminf.difdata(&data)_crt);    difs = limsup.difdata(&data_crt);
    printf("diferența în număr de zile = %ld\t%ld\n",difi,difs);
    dc *pinf;        pinf = pinf–>datamin();
    dc *psup;       psup = psup–>datamax();
    printf("limita inferioară\n");      pinf–>afisdata();
    printf("limita superioară\n");     psup–>afisdata();
    delete pinf;    delete psup;
    }   }
```

2.8 Să se definească tipul abstract *punct* care să se utilizeze la instanțierea punctelor din plan prin coordonate rectangulare.

```
    //   FIŞIERUL BXXII8.H
    class punct {       double x;       double y;       }
    public:
punct(double abs=0,double ord=0);
punct(const punct&);
    int citpunct( ); // citeşte coordonatele punctului; returnează: O la EOF; 1 a1tfe1.
void afispunct( ); // afişează coordonatele punctului
void xtrans(double dx);  // translație pe direcția abscisei
void ytrans (double dy); // translație pe direcția ordonatei
    double retx(); double rety();            // returnează abscisa şi ordonată
    };
    Funcțiile membru se definesc separat într-un fişier de tip CPP.
    //   FIŞIERUL BXXII8
    #ifndef __PUNCT_H    #include "BXXII8.H"    #define __PUNCT_H    #endif
    inline punct::punct(double abs,double ord) { / * - constructor pentru instantierea obiectelor
    de tip punct; - implicit se instantiază originea axelor. */
    x = abs;    y = ord;
    }
    inline punct::punct(const punct& p) {  // constructor de copiere
x = p.x;    y = p.y;
    }
    int punct::citpunct() {    // - citeşte coordonatele punctului; - returneaza: 0-la EOF; 1-altfel.
int c;
for(;;){
    printf("Abscisa=");   if((c=scanf("%lf",&x))==1) break;
    if(c==EOF) return 0;
    printf("nu s-a tastat un număr\n");   fflush(stdin);
    }
for(;;){
    printf("Ordonata="); if((c=scanf("%lf",&y))==1) return 1;
    if(c==EOF) return 0;
    printf("nu s-a tastat un număr\n");   fflush(stdin);
    }   }
```

```
inline void punct::afispunct() {       //afisează coordonatele punctului
printf("x=%g\ty=%g\n",x,y);
}
inline void punct::xtrans(double dx) {  x += dx; }      // translație în direcția axei x
inline void punct::ytrans(double dy) { y += dy }        // translație în direcția axei y
inline double punct::retx() { return x; }      //returnează abscisa
inline double punct::rety() { return y; }      //returnează ordonata
```

22.9 Să se scrie un program care generează obiecte de tip *punct* ale căror coordonate se generează aleator. Programul afişează punctele care aparţin ecranului, iar în final se indică: numărul *m* al punctelor afişate; numărul *n* al punctelor generate; raportul *m/n*.

```
//   PROGRAMUL BXXII9
#ifndef __STDIO_H     #include <stdio.h>   #define __STDIO_H     #endif
#ifndef __CONIO_H     #include <conio.h>   #define __CONIO_H     #endif
#ifndef __STDLIB_H    #include <stdlib.h>  #define __STDLIB_H    #endif
#include "BXXII8.CPP"
main() {   /* - generează puncte cu coordonate aleatoare; - afişează punctele care aparţin ecranului (80 coloane a 25 linii); - afişează numărul m al punctelor afişate; - afişează numărul n al tuturor punctelor generate; - raportul m/n. */
  int m=0,n=0;
for(;;) {
double xaleat == rand()%100;        double yaleat = rand()%100;
punct pct(xaleat,yaleat);                n++;
if(xaleat > 0 && xaleat <= 80 && yaleat 0 && yaleat <= 25) { //afişează punctul
    m++;       pct.afispunct();
    if(m%22 == 0) {
        printf("Acționați o tastă pentru a continua\n");
        printf("Acționați zero pentru a termina\n");
        if(getch() == '0') break;
}   }   }
printf("numărul punctelor afişate = %d\n",m);
printf("numărul punctelor generate = %d\n",n);
printf("raportul m/n = %g\n", (double)m/n);
}
```

22.10 Să se definească tipul abstract *vector*. Un vector este definit de 2 puncte. Unul este originea vectorului, iar celalalt este virful lui.

```
//   FIŞIERUL BXXII10.H
#ifndef __PUNCT_H     #include "BXXII8.CPP"      #define __PUNCT_H     #endif
class vector {
  punct origine;      punct virf;
public: // constructori
  vector(double xo,double yo,double xv, double yv);    vector(double xv,douhle yv);
  vector(const vector&);        double modul ( ); // modulul vectorului
  double arg ( ); // argumentul vectorului (unghiul făcut cu axa Ox)
public: // constructori
  void x_trans(double dx);      // translație în direcția axei Ox
  void y_trans(double dy);      // translație în direcția axei Oy
public:
  punct retorig();                  // returnează originea vectorului
  punct retvirf();                  // returnează vîrful vectorului
public:
  double prodscalar(vect,or *v);   // returnează produsul scalar dintre vectorul curent şi
                                    // cel spre care pointează v
};
```

Observație: Modulul şi argumentul vectorilor se calculează asemănător cu modulul şi argumentul numerelor complexe. În acest caz se face diferența dintre coordonatele extremităților vectorului. Funcțiile membru ale clasei vector se definesc în fişierul de mai jos, de extensie *CPP*.

```
//   FIŞIERUL BXXII10
#ifndef __VECTOR_H       #include "BXXII10.H"   #define __VECTOR_H       #endif
#ifndef __MATH_H      #include <math.h>         #define __MATH_H      #endif
#ifndef __PI          #define PI 3.14159265358979       #define __PI  #endif
inline vector::vector(double xv, double yv): virf(xv,yv) {
}               //originea are coordonatele egale cu zero (valori implicite)
inline vector::vector(double xo,double yo,double xv,double yv):origine(xo,yo),virf(xv,yv) {
}               // constructor pentru inițializarea originii şi virfului unui vector
vector::vector(const vector& v): origine(v.origine),virf(v.virf) {
}       //constructor de copiere
double vector::modul() {     // returnează modulul unui vector
  double a = virf.retx() – origine.retx();   double b = virf.rety() – origine.rety();
  return sqrt(a*a+b*b);
```

```
    }
    double vector::arg() {    // returnează argumentul unui vector
    double a = virf..retx() – origine.retx();      double b = virf.rety() – origine.rety();
if(a == 0 && b== 0) return 0.0;
if(b == 0)
    if(a > 0)   return 0.0;
    else        return PI;
if(a ==0)
    if(b > 0)   return PI/2;
    else        return(3*PI)/2;
        // a!=0 şi b!= 0
double c = atan(b/a);
if(a < 0)   return PI+c;   //a < 0 şi b != 0
if(b < 0)   return 2*PI+c;    //a >0 şi b < 0
    return c;   //a >0 şi b > 0
    }
    inline void vector::x_trans(doub1e dx) {   // translație în direcția axei Ox
    origine.xtrans(dx);    virf.xtrans(dx);
    }
    inline void vector::y_trans(double dy) {    // translație în direcția axei Oy
origine.ytrans(dy);    virf.ytrans(dy);
    }
    inline punct vector::retorig() {  return origine; }  // returnează originea vectorului
    inline punct vector::retvirf() {  return virf; }      // returnează vîrful vectorului
    double vector::prodscalar(vector *v) { /* returnează produsul scalar dintre vectorul curent
        şi cel spre care pointează v */
    double ax = v–>virf.retx()–v–>origine.retx(); double ay=v–>virf.rety()–v–>origine.rety();
double bx = virf.retx()–origine.retx();      double by = virf.rety() – origine.rety();
return ax*bx+ay*by;
    }
```

22.11 Să se scrie un program care citeşte coordonatele a 2 vectori din plan şi afişează modulul şi argumentul fiecărui vector, precum şi produsul lor scalar.

```
    //   PROGRAMUL BXXII11
    #ifndef __STDIO_H    #include <stdio.h>   #define __STDIO_H    #endif
    #ifndef __STDLIB_H   #include <stdlib.h>  #define __STDLIB_H   #endif
    #include "BXXII10.CPP"
    main() {  / * - citeşte coordonatele a 2 vectori; - calculează şi afişează: modulul şi argumentul fiecărui vector; - produsul scalar al ceior doi vectori. */
punct ov1, ov2; /* origine */    punct vv1,vv2;   // vîrf
ov1.citpunct();    vv1.citpunct();    //citeşte primul vector
ov2.citpunct();    vv2.citpunct();    //citeşte al doilea vector
vector v1(ov1.retx(),ov1.rety(),vv1.retx(),vv1.rety());    //construieşte vectorii
    vector v2(ov2.ret;x(),ov2.rety(),vv2.retx(),vv2.rety());
    // afişează modulul şi argumentul lui v1 şi v2
printf( "Vectorul v1\n"); printf("modul =%g\t argument=%g\n",v1.modul(),v1.arg());
printf("Vectorul v2\n"); printf("modul = %g\t argument = %g\n",v2.modul(),v2.arg());
printf("(v1,v2) = %g\n",v1.prodscalar(&v2)); // afişează produsul scalar
    }
```

22.12 Să se definească tipul abstract *sir* pentru instanțierea şirurilor de caractere. Interesul manifeslat pentru acest tip rezultă din faptul că şirnrile de caractere sunt utilizate frecvent în aproape toate programele. Prezentăm o implemetare a tipului *sir*. Ca date membru alegem un pointer spre zona de memorie în care se păstrează caracterele şirului şi lungimea acestora în număr de caractere (fără caracterul nul de la sfîrşitul şirului).

```
    //   FIŞIERUL BXXII12.H
    class sir {
char *psir;    int lung;
    public:
sir(char *s);   /* constructor pentru inițializarea obiectului cu pointerul spre şirul de
    caractere; acesta se păstrează în memoria heap */
sir(int nrcar=70); /* constructor care rezervă zonă de memorie în memoria heap şi
    păstrează în ea şirul vid */
    sir(const sir&);   //constructor de copiere
~sir();             // destructor
int retlung();      // returnează lungimea şirului
void afsir();       // afişează şirul de caractere
int citsir();       // citeşte un şir de caractere
```

```
        Boolean atribsir(sir *a); /* - transferă şirul spre care pointează s în zona rezervată pentru
        şirul obiectului curent; - dacă nu există zonă suficientă, se trunchează şirul spre care
         pointează s şi se returnează valoarea false; - altfel se returnează true. */
        };
        Mai jos, se definesc funcţiile membru ale clasei sir.
        //   FIŞIERUL BXXII12
        #ifndef Boolean       #define __Boolean        enum Boolean (false,true);  #endif
        #ifndef __STDIO_H     #include <stdio.h>   #define __STDIO_H     #endif
        #ifndef __STRING_H    #include <string.h>  #define __STRING_H    #endif
        #ifndef __SIR_H       #include"BXXII12.H"   #define __SIR_H       #endif
        sir::sir(char *s) { /* constructor utilizat la iniţializarea obiectului cu pointerul spre copia
                            în memoria heap a şirului spre care pointează s */
    lung = strlen(s);  //lungimea şirului
    psir = new char[lung+1];     //rezervă zonă în memoria heap
    strcpy (psir,s);             // transferă şirul în memoria heap
        }
        sir::sir(int dim) { /* - constructor care rezervă zonă pentru şiruri de dimensiunea dim;
            - păstrează şirul vid în zona respectivă. */
    lung = dim;    //determină lungimea
    psir = new char[lung+1];     // rezervă zona
    *psir = '\n';    // păstrează şirul vid
        }
        sir::sir(const sir& s) {    //constructor de copiere
    lung = s.lung;          // lungimea şirului
    psir = new char[lung+1];     // rezervă zona
    strcpy(psir,s.psir);     // copiază şirul
        }
        inline sir::~sir() {// destructor: eliberează zona din memoria heap ocupată de sir
        delete psir;
        }
        inline int sir::retlung() { // returnează lungimea şirului: numărul caracterelor şirului fără
    return lung; // caracterul nul
        }
        inline void sir::afsir() {  // afişează şirul spre care pointează psir
        printf(psir);    printf("\n");
        }
        int sir::citsir() {   /* - citeşte un şir de la intrarea standard şi-l păstrează în zona heap
            rezervată pentru obiectul curent; - returneaza: 0 - la sfîrşit de fişier;
        –1 - la trunchierea sirului citit;  1 - altfel. */
    char temp[255];
    if(gets(temp)==0)     return 0;   //s-a întîlnit sfîrşitul de fişier
    strncpy(psir,temp,lung);      *(psir+lung)="\0";
    if(strlen(temp) > lung)    return -1; // trunchiere
        else return 1;
        }
        Boolean sir::atribsir(sir *s) {     /* - transfera sirul spre care pointează s în zona rezervată
        pentru şirul obiectului curent; - dacă nu există zonă suficientă, se trunchează şirul
        spre care pointează s şi se returnează valoarea false; altfel se returnează true */
    strncpy(psir,s–>psir,lung);
    if(strlen(s–>psir) > lung)     return false;
    return true;
        }
```

22.13 Să se scrie un program care realizează următoarele operaţii asupra obiectelor de tip *sir*: iniţializare; citire de şiruri de caractere de la intrarea standard; copiere de obiecte de tip *sir*, atribuiri de obiecte de tip *sir*, afişări de şiruri de caractere din compunerea obiectelor de tip *sir*.

```
        //   PROGRAMUL BXXII13
        #ifndef __STDIO_H     #include <stdio.h>   #define __STDIO_H     #endif
        #ifndef __STDLIB_H    #include <stdlib.h>  #define __STDLIB_H    #endif
        #include "BXXII12.CPP"
        main ( ) { // operaţii cu obiecte de tip sir
         //instantieri de obiecte de tip sir
        sir sir1("Limbajul C++ este un C mai bun");
    sir sir2("Limbajul C++ suportă stilul de programare:\n - prin abstractizarea datelor;\n\
        - orientată spre obiecte");
    sir sir3; //instanţiere fără iniţializare; sir3 conţine şirul vid
```

```
sir sir4 = sir1; //instanţiere folosind constructorul de copiere
    //afişarea obiectelor instanţiate mai sus
printf("sir1\n");      sir1.afsir();      printf("sir2\n");      sir2.afsir();
printf("sir3\n");      sir3.afsir();      printf("sir4\n");      sir4.afsir();
    //citiri de şiruri de la intrarea standard
while(sir3.citsir( )) sir3.afsir();  //citirea şi afişarea şirurilor citite
    //atribuiri de şiruri
if(sir3.atribsir(&sir1) == false)  printf("trunchiere la atribuire\n");   sir3.afsir();
if(sir3.atribsir(&sir2) == false)  printf("trunchiere la atribuire\n");   sir3.afsir();
    }
```

12.10. Funcţii prietene (Friend function)

La începutul temei, s-a afirmat că o proprietate de bază a tipurilor abstracte este protecţia datelor membru ale tipului respectiv. Elementele protejate constituie aşa numita implementare a tipului abstract. Tipurile abstracte se definesc cu ajutorul claselor. Se obişnuieşte să se spună că datele protejate ale unui tip abstract sunt incapsulate în clasa care defineşte tipul respectiv.

Protecţia datelor se realizează prin aceea că la ele au acces numai funcţiile membru ale tipului (clasei). De asemenea, dacă o funcţie membru este protejată, atunci ea poate fi apelată numai prin intermediul unei funcţii membru a tipului (clasei). Acest mod de lucru, deşi asigură o protecţie bună a elementelor membru protejate ale clasei (elemente protejate prin private sau protected), uneori este considerat prea rigid. Astfel, deşi există funcţii descrise în C care pot fi utilizate pentru a prelucra instanţieri ale unei clase, ele nu se pot utiliza simplu deoarece nu sunt funcţii membru ale clasei respective şi deci nu au acces la datele membru. Mai mult decît atît, o funcţie membru se apelează totdeauna în dependenţă cu un obiect care este numit **obiectul curent** al apeiului. În acest fel, o funcţie membru se apelează prin una din urmatoarele formate:

```
    nume_obiect.nume_funcţie_membru(...)
sau     pointer_nume_clasă –> nume_funcţie_membru()
```

În cazul funcţiilor obişnuite nu se admit astfel de apeluri, toate datele prelucrate de funcţie se transferă prin parametri sau sunt globale. De aceea, o funcţie obişnuită poate fi utilizată pentru a prelucra obiectele unei clase dacă ea se modifică în aşa fel ca să devină funcţie membru.

S-a făcut un compromis pentru a admite accesul la elementele protejate şi pentru anumite funcţii care nu sunt membru. Aceste funcţii au fost numite **funcţii prieten** pentru clasa respectivă. Ele trebuie să fie precizate ca atare în definiţia clasei. În acest scop, prototipurile lor sunt prezente în definiţia clasei şi sunt precedate de cuvîntul cheie **friend**.

Exemplu: Fie tipul definit de utilizator:

```
struct complex {
double real;   double imag;
};
```

Funcţia *modul*, pentru calculul modulului unei date de tip complex poate fi definită, ca mai jos, ca o funcţie obişnuită:

```
double modul(complex *z){ // returnează modulul numărului complex spre care pointează z
return sqrt(z–>real * z–>real + z–>imag * z–>imag);
}
```

Dacă se consideră declaraţiile:

```
complex z1 = {1,1};      double d;
```

atunci instrucţiunea de atribuire: *d = modul(&z1);* atribuie lui d modulul lui z1 = 1+ i.

Acelaşi lucru se poate realiza implementînd tipul abstract *complex*:

```
class complex {
double real;   double imag;
public:
complex(double x=0,double y=0) {
 real = x;  imag = y;
}
double modul() {
 return sqrt(real*real+imag*imag);
}  …
};
```

Declaraţia pentru z1 se scrie:

```
    complex z1(1,1);
```

iar instrucţiunea de atribuire devine:

```
    d = z1.modul();
```

În acest caz, funcţia *modul* nu are parametru deoarece ea se apelează pentru obiectul *z1*. În corpul funcţiei *modul* este definit pointerul implicit *this* şi acesta are ca valoare chiar adresa lui z1. Expresia: *real*real+imag*imag* din corpui funcţiei membru *modul*, trebuie considerată ca şi cînd ar fi scrisă sub forma:

```
    this–>real * this–>real + this–>imag * this–>irnag
```

Funcţia *modul*, definită pentru tipul utilizator *complex* poate înlocui funcţia membru *modul* a clasei *complex* dacă ea se declară ca funcţie prieten a clasei *complex*, ca mai jos:

```
class complex {
double real;   double imag;
```

```
    public:
complex(double x=0,double y=0) {
    real = x;   imag = y;
    }
friend double modul(complex *z);  …
    };
```

Funcția *modul*, prieten a clasei *complex*, are aceeaşi definiție ca şi funcția *modul* definită pentru tipul *complex* introdus prin construcția *struct*. De asemenea, ea se apelează ca orice funcție obişnuită. În exemplul de față, dacă se consideră instanțierea:

```
    complex z(1,1);
```

atunci funcția prieten *modul* se apelează în mod obişnuit, adică prin:

```
    d = modul(&z);
```

Spre deosebire de funcțiile membru, în cazul funcțiilor prieten nu mai este definit pointerul implicit *this*. Acest lucru conduce la faptul că o funcție prieten are un parametru în plus față de o funcție membru care are acelaşi efect. O funcție prieten poate fi o funcție obişnuită (ca în cazul funcției *modul*) sau o funcție membru a unei alte clase. Exemplu:

```
    class clasa1 {
… tip functie_membru(...); …
    };
    class clasa2 {
… friend tip clasa1::functie_membru(...); …
    };
```

Funcția *functie_membru* este o funcție membru a clasei *clasa1*. Ea este o funcție prieten a clasei *clasa2*.

În cazul în care se doreşte ca toate funcțiile membru ale clasei *clasa1* să fie funcții prieten ale clasei *clasa2*, se poate proceda ca mai jos, în loc de a indica individual fiecare funcție din clasa *clasa1* că este funcție prieten:

```
    class clasa1;       //definiția prescurtată de clasă
    class clasa2 {
… friend clasa1;      …
    };
```

În acest caz, se spune că *clasa1* este o clasă prieten a clasei *clasa2*.

Proprietatea de clasă prieten nu este tranzitivă. Astfel, dacă *clasa1* este o clasă prieten pentru *clasa2* şi *clasa2* este o clasă prieten pentru *clasa3*, aceasta nu implică faptul că *clasa1* este clasă prieten pentru *clasa3*.

Modificatorii de protecție nu au nici o influență asupra unei funcții prieten. De aceea, specificarea faptului ca o funcție este prieten pentru o clasa, poate fi scrisă în orice punct din interiorul definiției clasei respective. Funcția prieten nu este protejată şi deci poate fi utilizată fără nici o restricție ca orice funcție obişnuită. Exerciții:

22.14 Să se scrie o funcție care ridică un număr complex la o putere întreagă pozitivă. Un număr complex se poate ridica la o putere întreagă pozitivă folosind *formula lui Moivre*. Aceasta se exprimă prin relația:

$$(r\cdot(\cos\alpha+i\cdot\sin\alpha)^n = r^n\cdot(\cos n\alpha + i\cdot\sin n\alpha$$

unde: r - este modulul numărului complex; α - este argumentul acestuia. Tipul complex se implementează folosind clasa definita în exercițiul 22.3.

```
    //  FUNCȚIA BXXII14
    void cputere(cornplex& z,int n){ /*ridică la puterea n num.complex la care z este referință
double r;  double a;
    r = z.modul( ); /* - se calculează modulul numărului complex la care z este referință;
                - se apelează funcția membru modul a clasei complex */
a = z.arg( );   /* - se calculează argumentul numărului complex la care z este referință;
                - se apelează funcția membru arg a clasei complex */
double rlan = pow(r,(double)n);     double na = n*a;
z.real = rlan*cos(na);              z.imag = rlan*sin(na);
    }
```

Observație: Funcția *cputere* nu este o funcție membru a clasei *complex*. Ea are acces la componentele *real* şi *imag* ale obiectului de tip complex referit de z, numai dacă este o funcție prieten a clasei *complex*.

22.15 Să se modifice definiția clasei *complex* din fişierul BXXII3.H inserînd funcția *cputere* ca funcție prieten.

```
    //  FIŞIERUL BXXII15.H
    enum Boolean {false,true};
    class complex {  //date membru protejate (private)
double real;   double imag;
    public:             // funcții membru neprotejate
complex(double x =0, double y = 0);       complex(const complex&);
double modul();      double arg();          double retreal();
double retimag();    void afiscomplex();    Boolean citcomplex();
void adcomplex(complex *z1,complex *z2);void sccomplex(complex*z1,complex *z2);
void negcomplex(complex *z1);             void mulcomplex(complex*z1,complex *z2);
Boolean divcomplex(complex *zl,complex *z2);
friend void cputere(complex& z,int n);// funcție prieten calculează z**n,pentru n natural
    };
```

22.16 Să se scrie un program care ridică numărul complex 1+i la puterea n. Numărul n este un întreg citit de la intrarea standard.

```
// PROGRAMUL BXXII16
#ifndef __PI #define PI 3.141592653589793238 #define __PI #endif
#ifndef __MATH_H #include <math.h> #define __MATH_H #endif
#ifndef __STDIO_H #include <stdio.h> #define __STDIO_H #endif
#define __BXXII3_H //pentru a nu include fişierul BXXII3.H se defineşte __BXXII3_H
#include "BXXIIl5.H" //se include definiția clasei complex din fişierul BXXII15.H
#include "BXXIZ3.CPP" //se includ definițiile funcțiilor membru
#include "BXXII14.CPP" //se include definiția funcției prieten cputere
#include <stdlib.h>
main() { // citeşte intregul n, calculează şi afişează (1+i)**n
int n,c; complex z(1,1); //z = 1+i
complex zlan; //zlan = 0+0*i
for(;;) { // citeşte pe n
printf("exponent=");
if((c = scanf("%d",&n)) == 1) break;
printf("nu s-a tastat un întreg\n");
if(c == EOF) {
printf("s-a tastat EOF\n"); exit (1);
}
fflush(stdin);
}
complex complex_unu(1,0); int m;
m = n < 0 ? –n: n; //calculează abs(n)
if (m == 0) { //(1+i)**0 = 1+0.i
complex_unu.afiscomplex(); exit(0);
}
if(n == 1) { //(1+i)**1 = 1+i
z.afiscomplex(); exit(0);
}
if( n == –1) { // (1+i)**(–1) = 1/(1+i)
zlan.divcomplex(&complex unu,&z);
zlan.afiscomplex(); exit(0);
}
//m >1
cputere(z,m); //z = z**m
if(n > 1) {
z.afiscomplex() exit(0);
}
// putere negativă: z**n = 1/z**abs(n)
zlan.divcomplex (&complex_unu, &z);
zlan.afiscomplex();
}
```