5. ELEMENTE SPECIALE DESPRE CLASE ȘI FUNCȚII

În acest capitol se vor aborda elemente mai specifice ale limbajului C++ și anume: tehnici de creare și inițializare a obiectelor, câteva elemente specifice despre constructor și destructor, funcții și clase de tip **friend**, elemente preliminare despre clase derivate și funcții virtuale, tipuri de funcții în limbajul C++, prototipul unei funcții etc.

5.1. Tehnici de creare și inițializare a obiectelor

Exemplele întâlnite până aici s-au referit la cazuri de creare a obiectelor simple și a masivelor de obiecte. În practică, situația normală este aceea în care trebuie create mai multe obiecte, pentru care se rezervă spațiu de memorie. Se pune astfel problema utilizării raționale a spațiului de memorie dinamică (de tipul *heap*). Aceasta înseamnă că vor trebui eliminate acele obiecte a căror existență nu mai este oportună. Se spune că în această situație, obiectele vor fi create în mod dinamic.
Din limbajul C se știe că există cel puțin patru funcții, trei pentru alocare (malloc(), calloc(), realloc()) și una pentru eliberare (free()) a spațiului de memorie. Ele au prototipurile în fișierul antet <**alloc.h**>. Cele mai folosite sunt funcțiile malloc() și free().
Considerăm următoarea structură:

```
struct  SALESMAN {
        char nume[20];
        int volum_vinz_efective;
        int nr_ore_lucrate_sapt;
        int volum_de _vindut;
        long salariu_sapt;
};
```

Utilizând funcția malloc() se obține adresa în memoria dinamică unde va fi alocată o astfel de structură de date la momentul definirii unei variabile de tipul (cu șablonul) SALESMAN. Deoarece funcția malloc() întoarce o adresă, vom declara o variabilă p de forma: **char** *p; Dacă vom defini o structură denumită "salesman" cu șablonul SALESMAN, atunci spațiul în memoria dinamică corespunzător acestei structuri se va aloca prin: p = malloc(sizeof (salesman));
În principiu, acest stil "pur" C este acceptat și în C++. Dar, în C++ există operatorul **new**, care ne permite să alocăm spațiu de memorie pentru obiect (obiecte) la momentul execuției.
Modul de utilizare al operatorului **new** este exemplificat în programul următor:

*Exemplul 5.1.***.**

```
// Program P5_1.CPP  Urmărirea vânzărilor pe oameni
# include <iostream.h>
# include <string.h>
struct SALESMAN {
        char nume[20];
        int volum_vinz_efective;
        int nr_ore_lucrate_sapt;
        int volum_de_vindut;
        long salariu_sapt;
        SALESMAN (char *cine = " ", int vef = 0, int no = 0, int vv = 0, long s = 0l);
public:
        void IDatePrelim (void);
};

SALESMAN :: SALESMAN (char *cine, int vef, int no, int vv, long s)
```

```
{       strcpy (nume, cine);
        volum_vinz_efective = vef;
        nr_ore_lucrate_sapt = no;
        volum_de_vindut = vv;
        salariu_sapt = s;
}
void SALESMAN :: IDatePrelim (void) {
        cout << "Numele persoanei :";
        cin >> nume;
}
void main ()
{       SALESMAN  *p;
        p = new SALESMAN;
        strcpy (p->nume, "Albu Marin");
        p->salar_sapt = 5000;
        p->volum_vinz_efective = 12000;
        p->volum_de_vindut = 10000;
        p->nr_ore_lucrate_sapt = 50;
        . . . . . .
        p->IDatePrelim ();
}
```

Există o diferență netă între operatorul **new** și funcția de alocare malloc(). Opratorul **new** determină în mod automat spațiul de memorie necesar prin analizarea "numelui de tip" ce îi urmează. În acest exemplu, numele de tip a fost o structură cu șablonul SALESMAN. Adresa întoarsă în p (pointer la o structură) ne permite să avem acces la membrii structurii. Dacă rezervarea de memorie pentru această structură s-ar face cu funcția malloc() trebuie să se scrie: p = malloc (**sizeof** (**struct** SALESMAN)).

Alte exemple de utilizare a operatorului **new**

Pentru a se aloca un spațiu de o sută de cuvinte calculator, în C putem scrie, de exemplu:

```
        int *p;
        p = malloc(100*sizeof(int));
```

iar în C++ se scrie astfel:

```
        int *p;
        p = new int[100];
```

După operaturul **new** urmează un nume de tip, în exemplul anterior **int**, iar între paranteze drepte, dimensiunea, adică numărul elementelor de acel tip, care vor fi alocate în zona de memorie. În urma executării instrucțiunii de mai sus se alocă o zonă de memorie de o sută de elemente de tip **int,** iar adresa de început este atribuită variabilei p.

Pentru alocarea unui spațiu pentru un vector de cinci pointeri și reținerea adresei sale în variabila pp (de tipul pointer la pointer la întreg), vom scrie instrucțiunile următoare:

```
        int **pp;
        pp = new int  *[5];
```

În C, eliberarea spațiului alocat dinamic cu malloc(), calloc() sau realloc() se efectuează prin intermediul funcției free(). Deci, pentru cazul de mai sus, în care alocarea memoriei s-a facut cu malloc(), eliberarea acesteia se comunică compilatorului prin: free(p). În C++, opusul lui **new** este operatorul **delete**. Pentru contextele exemplelor de mai sus vom scrie: **delete** p; și respectiv **delete** pp.

Observație: Nu se pot folosi ambele mecanisme de gestionare dinamică a spațiului de memorie, respectiv malloc()/free() și **new**/**delete** în același program.
Aparent, mecanismul **new**/**delete** pare mai îngrădit decât celelalte mecanisme dacă se are în vedere și funcția realloc() care permite realocarea unui spațiu mai mic sau egal de memorie. Legat de acest aspect, în C++, se poate întâlni și forma următoare:

```
double *p = new double [10];
delete  [5]p;
```

Ultima instrucțiune arată că se va elibera numai jumătate din spațiul de zece cuvinte duble alocat anterior. Evident, cealaltă jumătate va fi automat realocată.
Observație: De regulă, operatorii **delete** se întâlnesc în cadrul destructorilor.
Exemplul următor conține o aplicație a operatorului **new**. Mai exact este vorba de preluarea a două șiruri de caractere, concatenarea lor și apoi inversarea șirului rezultat.

```
//Program P5_2.CPP Concatenarea a două șiruri și inversarea șirului rezultat
char *caten (const char *s1, const char *s2);        // Prototipul funcției de concatenare
void reverse (char *s);                              // Prototipul funcției de inversare
void swap (char &, char &);
# include <iostream.h>
# include <string.h>
char s1[] = "1234567890";                            // Primul șir
char s2[] = "abcdefghijklmnopqrstuvzy";              // Cel de-al doilea șir
void main (int, char**)
{       cout << "Sirul s1: " << s1 << endl;
        cout << "Sirul s2: " << s2 << endl;
        char *p;
        p = caten (s1,s2);
        cout << "Sirul concatenat rezultat: " << p << endl;
        reverse (p);
        cout << "Sirul inversat rezultat: " << p << endl;
        return 0;
}
char *caten (const char *s1, const char *s2)         // Definirea funcției caten()
 {      int lng_sir1 = strlen (s1);
        char *ad = new char[ lng_sir1 + strlen(s2) + 1 ];
        if (!s1 || !s2)
        return 0;
        strcpy (ad, s1);
        strcpy (ad + lng_sir1, s2);
        return ad;
}
void reverse (char *s)
{       if (!s)
                return;
        char *invers = s + strlen(s)  - 1;
        for ( ; s < invers; s++, invers--)
                swap (*s, *invers);
}
void swap (char &a, char &b)         // Interschimbare în stil C++ cu parametri tip referință
```

```
{       char   t = a;
        a = b;
        b = t;
}
```
Funcționarea programului este clară și nu mai necesită alte comentarii. Trebuie făcută numai observația că operatorul **new** admite din punct de vedere sintactic în locul unei constante și o expresie care se evaluează într-un mod dinamic la momentul execuției.

5.2. Câteva elemente specifice despre constructor și destructor

Funcțiile constructor și destructor sunt de fapt funcții care diferă puțin față de definițiile funcțiilor. Considerăm definiția următoarei clase denumită CLS, a constructorului aferent și a unui obiect OB:

```
        class   CLS {
                int  i; long  l; double  d;
        public:
                CLS(int i = 0, long l = 0l, double d = 0.);
        };
        CLS :: CLS(int ii, long ll, double dd)
        { i = ii; l = ll; d = dd; }
        . . . . . .
        CLS    OB(10, 20l, 30.);
```

Se observă faptul că în prototipul constructorului s-au specificat valori asumate pentru variabilele i, l și d. Obiectul creat conține valori explicite pentru aceste variabile, astfel i = 10; l = 20l; d = 30.; Dacă s-ar fi creat următorul obiect CLS OB_asumat;, valorile datelor private i, l și d ar fi fost cele asumate, adică nule.

De regulă, operatorul **new** se folosește în funcția constructor. Clasa următoare, numită DIM2 definește un masiv bidimensional m[LIN][COL] cu LIN linii și COL coloane.

Program P5_3.CPP *Utilizarea operatorului new*

```
        class  DIM2  {
                int  *m;
        public:
                DIM2 (int l = 10, int c = 10)
                { m = new (int [l * c]);  }
        };
        void main (void)
        {
                DIM2 m1, m2(30,20);          // Definirea obiectelor
        }
```

În definirea obiectelor m1 și m2(30, 20), m1 va avea ca dată privată un masiv de zece linii și zece coloane, iar m2, 30 linii și 20 coloane. În cuprinsul constructorului lipsesc instrucțiunea **return** și tipul rezultatului "întors" de o astfel de funcție.

Ca și constructorul, destructorul are același nume cu cel al tipului abstract de date. În plus, numele destructorului este precedat de semnul "~". Pentru exemplul de mai sus, destructorul se definește în afara clasei sub forma:

```
        DIM2 :: ~DIM2(void)      { delete m; }
```

Constructorul este automat chemat la momentul definirii obiectului, iar destructorul este de asemenea automat apelat la părăsirea blocului unde este recunoscut acel obiect.

Până acum am definit obiecte în cadrul funcției main(). Deci, scopul numelor obiectelor va fi întregul program. Dar se pot crea obiecte și în interiorul altei funcții secundare existente într-un program. Un exemplu, în acest sens, este dat în programul următor:

```
// Program P5_4.CPP   Crearea dinamică a obiectelor
# include <stdio.h>
# include <string.h>
int f (int nl, int nc);
class DIM2 {
        int *m;
public:
        DIM2(int l = 10, int c = 10)
        {       int lc = l*c;
                m = new int[lc];
        }
};
void  main ()
{       int nl, nc, cod_retur;
        printf ("Precizati valorile pentru nl si nc \n");
        scanf ("%d  %d", &nl, &nc);
        cod_retur = f(nl, nc);
        printf ("Valoarea intoarsa este: %d\n", cod_retur);
}
int f(int nl, int nc)
{       DIM2 masiv(nl,nc);
        return 1;
}
```

La revenirea în funcția main() din funcția f(), destructorul ~DIM2() va elibera în mod automat memoria ocupată de către obiectul denumit "masiv". Se observă că destructorul este o funcție care nu are listă de intrare, nici nu întoarce vreun rezultat, deci nu poate poseda instrucțiunea **return**.

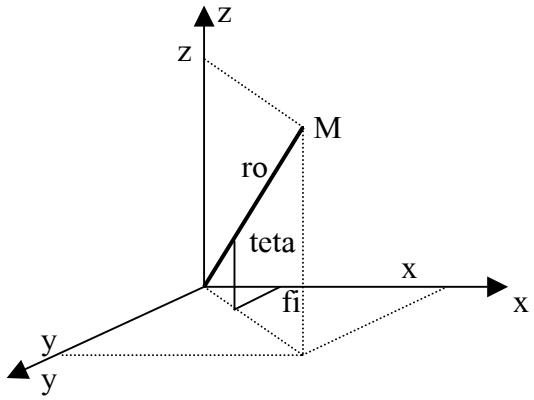
5.3. Funcții și clase de tip friend

Funcțiile și clasele de tip **friend** se caracterizează prin faptul că aceste clase și funcții se bucură de unele privilegii față de funcțiile și clasele comune, normale. Modul de definire și utilizare a unei funcții **friend** este prezentat în Exemplul 5.5. În acest exemplu se realizează transformarea din coordonate polare în coordonate carteziene și calculul distanței dintre două puncte.

Exemplul 5.5. // Program P5_5.CPP   *Geometrie în spațiu*

```
# include <stdio.h>
# include <math.h>
class PUNCT {
        double ro, fi, teta;            // Coordonate polare
        double x, y, z;                 // Coordonate carteziene
public:
        void Setare (double rr, double ff, double tt);
        void Xyz (void);                // Conversia în coordonate carteziene
        friend double distp1p2 (PUNCT *p1, PUNCT *p2);
};
```

```
void PUNCT :: Setare (double rr, double ff, double tt) {
        ro = rr; fi = ff; teta = tt;
}
void PUNCT :: Xyz (void)
{       double t;
        t = ro*cos(teta);
        x = t*cos(fi);
        y = t*sin(fi);
        z = ro*sin(teta);
}
double distp1p2 (PUNCT *p1, PUNCT *p2)
{
   return sqrt(
          (p1->x - p2->x) * (p1->x - p2->x) +
          (p1->y - p2->y) * (p1->y - p2->y) +
          (p1->z - p2->z) * (p1->z - p2->z)
          );
}
void  main (void)
{       double dist;
        PUNCT u, v;                     // Se creează două obiecte (puncte) denumite u și v
        u.Setare (10., .5, .5);
        v.Setare (20., 1., 1.);
        u.Xyz ();               // Se realizează conversia din coordonate polare în coordonate
        v.Xyz ();               // carteziene pentru obiectele u și v
        dist = distp1p2 (&u,&v);        // Se calculează distanța dintre punctele u și v
        printf ("\n\nDistanta este: %f \n", dist);
}
```



Funcția de tip **friend** distp1p2() calculează distanța dintre două obiecte (puncte), p1 și p2, situate în sistemul de referință Oxyz. O funcție de tip **friend** nefiind o funcție membru, la definire nu va mai accepta o formă de genul PUNCT :: distp1p2 (PUNCT *p1, PUNCT *p2) {...}, ci va fi definită așa cum s-a arătat mai sus.

5.4. Elemente preliminare despre clase derivate și funcții virtuale

Se dorește realizarea unei liste înlănțuite formată din mai multe "noduri". Fiecare nod va conține un număr care reprezintă prioritatea acelui nod. Nodurile vor fi sortate în listă în ordinea descrescătoare a priorității lor. Se vor prezenta două variante de realizare a acestei liste: una în manieră C, iar cealaltă în stil C++.

5.4.1. Varianta C

Vom defini trei clase: prima denumită VNod va conține structura unui nod, a doua denumită VSir va conține structura listei în sine, iar cea de-a treia cu numele IntVSir va fi o clasă derivată din clasa VSir pentru variabile de tip întreg. Definirea claselor VNod și VSir se face în fișierul antet "vsir.h".

```
// Fișier  VSIR.H   utilizat în P5_6.CPP
class VNod                                          // Structura unui nod din listă
{       friend class VSir;
        void *data;
        VNod *urmat;
        VNod (void *d, VNod *n) {data = d; urmat = n;}    // Constructorul clasei VNod
};
class VSir                                          // Structura listei de așteptare
{       VNod *virf;
public:
        void Inserez (void *d);
        void *Obtin (void);
        virtual int Gt (void *a, void *b)       {return 0;}
        virtual void Distrug (void *d)          {delete d;}
        void Elimin ();
        VSir () {virf = 0;}                             // Constructorul clasei VSir
        ~VSir () { Elimin ();}                          // Destructorul clasei VSir
};
```

În clasa VNod a fost admisă ca prieten clasa VSir. Rolurile funcțiilor membre ale clasei VSir sunt următoarele:

a) Funcția Inserez (**void** *d) realizează inserarea în listă a unui nod, după prioritatea sa (un număr întreg înscris în variabila data).
b) Funcția *Obtin() servește la obținerea adresei nodului cu prioritatea maximă din listă.
c) Funcția virtuală Gt() este o funcție de comparare a priorităților a două noduri. Ea este însoțită de cuvântul cheie **virtual**, ceea ce înseamnă că structura sa va fi mai precis conturată în funcția cu același nume din clasa derivată IntVSir.
d) Funcția virtuală Distrug() va avea de asemenea un algoritm precizat în clasa derivată IntVSir.
e) Funcția Elimin() este destinată eliminării unui nod din listă.

Listingul programului în care apar implementările claselor și funcția main() este prezentat în cele ce urmează:

```
// Program P5_6.CPP   Lanț de obiecte
# include <iostream.h>
# include "vsir.h"
void VSir :: Inserez (void *d)
{       if (!virf || Gt (d, virf->data))
        {       virf = new VNod (d, virf);
                return;                             // Se trece la instrucțiunea următoare
        }
        VNod *n = virf;

        while (n->urmat && !Gt (d, n->urmat->data))
                n = n->urmat;
```

```
        n->urmat = new VNod (d, n->urmat);
}
void *VSir :: Obtin (void)              // Se preia nodul cu prioritatea maximă
{       if (!virf) return 0;
        void *adresa = virf->data;
        VNod *sterg = virf;
        virf = virf->urmat;
        delete sterg;
        return adresa;
}
void Vsir :: Elimin ()                  // Se elimină întreaga listă de noduri
{       while (virf) {
                VNod *n = virf;
                virf = virf->urmat;
                Distrug (n->data);
                delete n;
        }
}
// Definiția clasei derivate
class IntVSir : public VSir
{public:
        void Inserez (int *d) { VSir :: Inserez (d);}
        int *Obtin () {return (int *)VSir :: Obtin ();}
        int Gt (void *a, void *b)       {return *((int *)a) > *((int *)b);}
        void Distrug (void *d) {delete ((int *)d);}
};
void  main ()
{       IntVSir     nr;                 // Se declară obiectul denumit "nr"
        nr.Inserez (new int (5));
        nr.Inserez (new int (8));
        nr.Inserez (new int (4));
        nr.Inserez (new int (1));
        int *p = nr.Obtin ();
        while (p) {                     // Se preia nodul cu prioritatea maximă
                cout << *p << endl;
                delete p;
                p = nr.Obtin ();
        }
}
```

Vom analiza acest program pe măsura parcurgerii instrucțiunilor din funcția main(). Se crează mai întâi obiectul denumit "nr", folosind "șablonul" clasei IntVSir derivată din clasa da bază VSir. Acest obiect va fi în fapt o listă (un șir) de noduri sortate în mod descrescător după valoarea priorității din variabila data.

Executarea instrucțiunii nr.Inserez (**new int**(5)); conduce la apelarea "în subteran" a constructorului clasei de bază VSir. Se vede aici *o nouă formă sintactică a operatorului* **new**. În locul parantezelor drepte de după **int**, se folosesc paranteze rotunde care înconjoară o constantă. În acest caz, compilatorul va fi informat că valoarea 5 va inițializa variabila "data" a primului nod. Pointerul d la valoarea 5 va fi pasat funcției Inserez() asociată clasei de bază VSir (prin instrucțiunea VSir ::

Inserez(d)) ce apare în definiția funcției Inserez() din clasa IntVSir). În definiția acestei funcții se ajunge la instrucțiunea **if**. Aceasta va inspecta variabila virf (începutul listei nodurilor). Întrucât a fost parcurs constructorul clasei VSir, s-a creat deja un vârf al listei, în care conținutul pointerului este NULL (prin instrucțiunea VSir () {virf = 0;}). Cum virf = 0, atunci, instrucțiunea !virf || Gt (d, virf->data) din **if** va fi evaluată ca adevărată și ca urmare se va executa instrucțiunea virf = **new** VNod (d, virf); Se apelează deci constructorul clasei VNod, pasându-i-se doi parametri și anume: prioritatea (d = 5) și valoarea variabilel virf (virf = 0). Analizând acum constructorul clasei VNod, constatăm că prin el, cei doi pointeri **data** și **urmat** sunt inițializați cu valorile precizate. Este evident faptul că tot prin înstrucțiunea virf = **new** VNod (d, virf); se obține și adresa locului din memorie unde a fost rezervat efectiv spațiul necesar primului nod al șirului "nr". Această adresă este pasată variabilei virf și ea este diferită de NULL. Grafic, se ajunge la situația din Fig. 5.1.

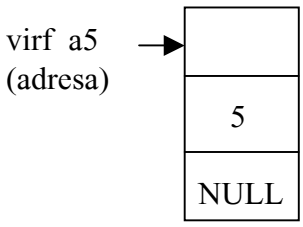


Fig. 5.1. Alocarea primului nod al șirului

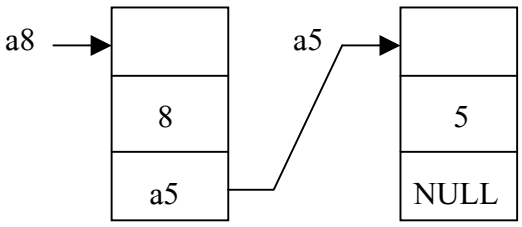


Fig. 5.2. Lista, după inserarea celui de-al doilea nod

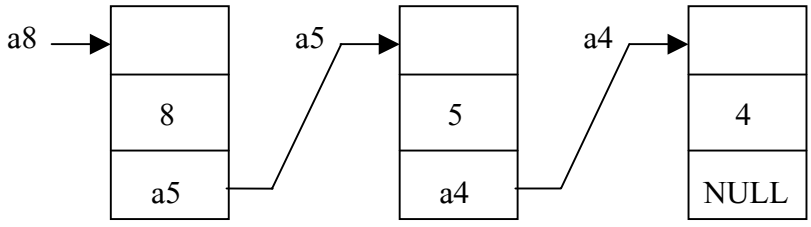


Fig. 5.3. Lista, după adăugarea celui de-al treilea nod

Urmează instrucțiunea nr.Inserez (**new int** (8)); Trebuie deci inserat un nod cu prioritate mai mare decât a celui deja existent. Urmărim aceeași cale: IntVSir :: Inserez() -> VSir :: Inserez() și ajungem din nou la instrucțiunea **if**(!virf || Gt(...)). De data aceasta rezultatul evaluării subexpresiei !virf este FALSE și se intră în evaluarea funcției de comparație Gt(). Această funcție este definită atât în clasa de bază, cât și în clasa derivată. Din această cauză, Gt() a fost declarată ca virtuală în clasa de bază, unde Gt() are ca unică instrucțiune doar **return** 0. În cadrul clasei derivate, IntVSir, Gt() este definită în detaliu prin: {**return** *((**int** *)a) > *((**int** *)b);}. Funcția Gt() preia cei doi pointeri de tip **void** la variabilele a și b și întoarce rezultatul comparației. Când a > b, se întoarce TRUE. În cazul nostru, se compară valorile 8 cu 5, prima, abia transmisă în variabila d. Deoarece rezultatul comparației este adevărat, se va executa din nou instrucțiunea virf = **new** VNod (d, virf); Acum constructorul VNod este apelat cu parametrii d = 8 și virf = a5. Se alocă acum spațiu pentru al doilea nod și cu aceasta variabila virf preia noua adresă de vârf a stivei (vezi Fig. 5.2).
Se trece la instrucțiunea nr.Inserez (**new int** (4)); Urmărind subexpresiile instrucțiunii **if**, se constată că !virf este FALSE și întrucât 4 este mai mic decât 8, Gt() va întoarce FALSE, ceea ce face ca expresia instrucțiunii **if** să fie FALSE. Drept urmare se va executa instrucțiunea VNod *n = virf. Cu alte cuvinte, se crează un obiect (nod) provizoriu, iar n capătă valoarea a8. Bucla **while** va "palpa" pe rând nodurile sortate deja în ordine descrescătoare. Prima dată, expresia n -> urmat are valoarea

a8 -> urmat = a5, adică ceva diferit de NULL. Acum se compară d (d = 4) cu data din nodul punctat de n -> urmat, adică a5 (valoare 5). Rezultatul comparației va fi FALSE. Deci !Gt() = TRUE. Ca urmare, se va executa instrucțiunea n = n -> urmat; adică se avansează la nodul a5 etc. După inserarea celui de-al treilea nod, lista arată ca în Fig. 5.3.
În continuare, în funcția main() urmează instrucțiunea **int** *p = Nr.Obtin(); În pointerul p va rezulta de fapt adresa vârfului stivei. Aceasta se obține după cum urmează. Se apelează mai întâi funcția IntVSir :: Obtin() care va apela VSir :: Obtin() în cadrul căreia se întâlnesc următoarele instrucțiuni:

```
void *VSir :: Obtin (void)              // Se preia nodul cu prioritatea maximă
{       if (!virf)      return 0;
        void *adresa = virf->data;      // Se preia adresa datei din nod
        VNod *sterg = virf;             // Se salvează într-un obiect adhoc (sterg) vârful actual
        virf = virf->urmat;             // Se trece la nodul cu prioritatea următoare
        delete sterg;                   // Se elimină fostul nod de vârf, dar s-a preluat adresa acestuia
        return adresa;                  //
}
```

Deci, prima dată, adresa va conține valoarea a8, iar adresa punctează valoarea 8. Funcția Obtin() este o funcție distructivă; ea îndepărtează nodurile din listă pe rând, nod cu nod, extrăgând valoarea priorității nodului curent eliminat.
În urma executării funcției main(), se va obține lista valorilor variabilelor data. Paralel, din lista de noduri, "se consumă" nod după nod, eliberându-se cu **delete**, memoria ocupată de acel nod.
Observații cu privire la programul anterior:

1. Programul este scris în manieră C.
2. Am întâlnit conceptul de clasa de tip **friend**, clasă derivată și funcție virtuală.
3. În program s-a operat cu trei clase:

a) clasa VNod care se referă la entitatea *nod generic al unei liste*;
b) clasa de bază VSir, referitoare la lista nodurilor generice, admisă ca prieten al clasei VNod;
c) clasa IntVSir, derivată din clasa de bază VSir, în care nodurile se referă la date de tip întreg.

4. Destructorul clasei VSir nu a apelat în mod automat operatorul **delete**, ci a apelat o funcție membră a clasei, și anume funcția Elimin(), care la rându-i a apelat funcția Distrug() care recurge la operatorul **delete**.

5.4.2. Varianta C++

Unul dintre cele mai mari avantaje ale programării în C++ este conceptul de "moștenire" a caracteristicilor clasei de bază în noua clasă derivată. Una din formele sintactice de definire a unei clase derivate este următoarea:

```
class nume_clasa_derivară : public nume_clasa_de_baza {
        // Declararea de noi membri
};
```

Întradevăr, în exemplul de mai sus, definiția clasei derivate IntVSir, care nu are date private este de forma:

```
class IntVSir : public VSir
{public:
        void Inserez (int *d) { VSir :: Inserez (d);}
        int *Obtin () {return (int *)VSir :: Obtin ();}
        int Gt (void *a, void *b)       {return *((int *)a) > *((int *)b);}
        void Distrug (void *d) {delete ((int *)d);}
};
```

Efectul cuvântului cheie **public** din fața clasei de bază constă în acordarea permisiunii de access la funcțiile clasei de bază din clasa derivată, IntVSir. Într-adevăr, funcția Inserez(**int** *d) din clasa IntVSir are acces la funcția Inserez(d) membră a clasei de bază. Precizăm de asemenea că același cuvânt cheie **public** ar fi dat dreptul și la adresarea variabilelor publice.
În exemplul de mai sus s-a observat de asemenea că două din funcțiile membre ale clasei de bază VSir și anume Gt() și Distrug() sunt precedate de cuvântul cheie **virtual**. Aceste funcții sunt apoi definite în clasa derivată IntVSir. Menționăm că pentru o funcție virtuală, algoritmul (codul) corespunzător nu trebuie definit în clasa de bază, acesta definindu-se în clasa derivată.

În cele ce urmează ne propunem să scriem un cod mai clar pentru tratarea nodurilor listei înlănțuite din exemplul de mai sus. În acest sens, vom reorganiza programul P5_6.CPP și headerul VSIR.H după tehnica containerului. Pentru aceasta vom alcătui patru clase denumite Obiect, ONod, OSir și Intreg. Clasele ONod și OSir sunt similare claselor VNod și VSir. Clasa Obiect va conține așa-numitele containere. De asemenea, aceasta va conține două funcții virtuale și anume destructorul clasei și funcția de comparație Gt() definită sub forma:

virtual int Gt(Obiect *d) {**return** d;}.

Se vede că Gt() nu compară nimic, ci pune doar la dispoziție pointerul la unul dintre obiectele ce vor fi comparate. În clasa Intreg, derivată din clasa Obiect, apare precizat algoritmul funcției Gt(). Cu precizarea că, acum, prioritatea unui obiect din listă este păstrată într-o variabilă denumită "prioritate", funcția Gt() este definită sub forma:

int Gt (Obiect *d) {**return** prioritate > ((Intreg *) d) -> prioritate;}

Se compară deci containerul punctat de către pointerul (Intreg *) d, adică obiectul d, cu data echivalentă a obiectului curent.
Listingul programului care implementează aceste clase este redat în cele ce urmează.

```
// Fișier OSIR.H    Utilizat în programul P5_7.CPP
class Obiect {                  // Clasa de bază
public:
        virtual ~Obiect () {}
        virtual int Gt (Obiect *d) {return 0;}
};
class ONod                      // Definiția unui nod
{       friend class OSir;
        Obiect *data;
        ONod *urmat;
        ONod (Obiect *d, ONod *n) {data = d; urmat = n;}
};
class OSir                      // Clasa de bază
{       ONod *virf;
public:
        void Inserez (Obiect *d);
        Obiect *Obtin (void);
        void Elimin ();
        OSir () {virf = 0;}
        ~OSir () { Elimin ();}
};

// Program P5_7.CPP   Lant de obiecte
# include <iostream.h>
```

```
# include "osir.h"
void OSir :: Inserez (Obiect *d)    // Definiția funcției de inserare a unui nod în ordinea sortată
{
        if (!virf || d->Gt (virf->data)) {
                virf = new ONod (d, virf);       // virf preia adresa nodului cel mai prioritar
                return;
        }
        ONod *n = virf;                          // Exista noduri în listă
        while (n->urmat && !d->Gt (n->urmat->data))
        n = n->urmat;
        n->urmat = new ONod (d, n->urmat);
}
Obiect *OSir :: Obtin (void)                     // Se preia nodul cu prioritatea maximă
{       if (!virf) return 0;                     // Listă vidă
        Obiect *adresa = virf->data;
        ONod *sterg = virf;
        virf = virf->urmat;
        delete sterg;
        return adresa;
}
void OSir :: Elimin ()                           // Ștergerea listei
{       while (virf) {
                ONod *n = virf;
                virf = virf->urmat;
                delete n->data;
                delete n;
        }
}
// Definitia clasei derivate denumită Intreg
class Intreg : public Obiect
{
public:
        int prioritate;
        Intreg (int v) {prioritate = v;}
        int Gt (Obiect *d)      {return prioritate > ((Intreg *) d)->prioritate;}
};

void  main (int, char **)
{
 OSir nr;                                // Se declară obiectul "nr"
 nr.Inserez (new Intreg (5));
 nr.Inserez (new Intreg (8));
 nr.Inserez (new Intreg (4));
 nr.Inserez (new Intreg (1));
 Intreg *p = (Intreg *) nr.Obtin ();
 while (p) {
        cout << p->prioritate << endl;
        delete p;
```

```
        p = (Intreg *) nr.Obtin ();
        }
  return 0;
}
```

Funcția Inserez() are cod similar cu cel din listingul P5_6.CPP și realizează sortarea obiectelor în ordinea descrescătoare a priorității lor.

5.5. Tipuri de funcții în limbajul C++

În limbajul C, programul este alcătuit din funcții, una dintre acestea fiind funcția main(). Din aceste funcții, utilizatorul recurge la diferite servicii, care sunt de fapt tot funcții aflate în diferite biblioteci. Prototipurile acestor servicii se află în diferite fișiere antet. Desigur, pe lângă bibliotecile standard pot exista și biblioteci proprii și deci și fișiere antet proprii.

În limbajul C++, funcțiile se clasifică în două mari categorii:

1. funcții membre ale clasei;
2. funcții care nu aparțin de nici o clasă.

Din prima categorie de funcții fac parte: constructori, destructori, funcții standard de acces, funcții virtuale, funcții de tipul **operator**, funcții statice, funcții constante.

Cealaltă categorie este întâlnită sub una din următoarele forme: funcții tip **friend**, funcții standard de access, funcții de tipul **operator**.

Toate aceste funcții posedă o serie de așa_numite *atribute*, care le conferă o și mai mare elasticitate față de funcțiile normale cu care eram obișnuiți în limbajul C. Tabelul de mai jos înfățișează aceste atribute.

Atributul	Funcții de tip membru	Funcții care nu sunt membru
"inline"	da	da
virtuale	da	nu
friend	nu	da
"redefine"	da	da
operator	da	da

5.6. Avantajele folosirii funcțiilor de tip inline

Funcțiile de tip **inline** sunt într-un fel echivalente *macro*-urilor întâlnite în limbajul C și acceptate și în C++. Deoarece acestea au o serie de avantaje față de macrodefiniții, în acest paragraf sunt prezentate câteva aspecte legate de funcțiile de tip **inline**.

a) Atributul **inline** poate fi atribuit atât unei funcții membru a unei clase, cât și unei funcții nemembru, așa cum se vede în exemplul următor.

```
// Program P5_8.CPP   Interfață cu math.h
# include <iostream.h>
# include <math.h>
inline double pi_to_x (double x)      // Declararea explicită a funcției pi_to_x() de tipul inline
        {return pow (M_PI, x);}
void  main (void)
{       double  x, y;
        cout << "Precizati x (in virgula mobila): ";
        cin >> x;
        y = pi_to_x (x);
        cout << "PI ^ x = " << y << endl;
}
```

Când compilatorul întâlnește o referire la funcția de tip **inline** pi_to_x(), acesta o va implanta, adică o va substitui prin copiere. Deci, prezența cuvântului cheie **inline** anunță compilatorul să nu mai genereze secvențele de apel a acestei funcții și de revenire din această funcție. În concluzie, deși durata compilării crește, excuția capătă rapiditate.
b) Deși funcțiile **inline** sunt într-un fel echivalente *macro*-urilor întâlnite în limbajul C și acceptate și în C++, totuși primele au o serie de avantaje față de macrodefiniții.
b1) Datorită mecanismului de substituire, o macrodefiniție poate altera valoarea unei variabile, pe când o funcție **inline**, datorită mecanismului de "copiere", nu alterează valoarea acelei variabile, cum se poate observa în exemplul următor.

```
// Program P5_9.CPP    Avantajele unei funcții inline în comparație cu o macrodefiniție
# include <iostream.h>

# define m_schimb(k) --k;                    // Definirea macrodefiniției

inline f_schimb (int k) {return --k;}        // Definirea funcției de tip inline

int u[1] = {1};

void  main (void)
{       int  i;
        i = f_schimb (*u);
        cout << "i = " << i << "   u = " << *u << endl;          //  i = 0  u = 1
        i = m_schimb(*u);
        cout << "i = " << i << "   u = " << *u << endl;          //  i = 0  u = 0
}
```

Se observă că atât *macro*-ul m_schimb() cât și funcția de tip **inline** f_schimb() reportează un i = 0. În timp ce macrodefiniția, prin substituirea liniei i = m_schimb(*u) în i = --*u, conduce la "alterarea variabilei punctate de u, funcția **inline** este translatată în felul următor:

```
        int   temporar;          // Se creează o variabilă temporară
        temporar = *u;           // temporar preia valoarea 1
        i = --temporar;          // Se modifică numai variabila temporară
```

Se vede că acest mecanism al copierii lasă nemodificată variabila u.
b2) Se știe că în C nu se poate transmite un nume de *macro* ca parametru într-o listă de apel a unei funcții, regulă ce se menține și în C++. Folosind însă un pointer la o funcție și făcându-l să puncteze chiar funcția **inline**, această funcție poate fi transmisă unei alte funcții.

```
// Program P5_10.CPP  Avantajele unei funcții inline în comparație cu o macrodefiniție
# include <iostream.h>
inline int f (char c) {return  c - '@';}        // Definirea funcției f() de tip inline
void  main (void)
{       int  (*pointer_la_fct) (char  c);       // Declararea unui pointer la funcție
        int car1, car2;
        char c1 = 'a';
        char c2 = 'A';
        pointer_la_fct = f;                     // Pointerul la funcție punctează funcția f
        car1 = (*pointer_la_fct) (c1);
        car2 = f (c2);
        cout << "car1 trebuie sa fie 33 si este : " << car1 << endl;
        cout << "car2 trebuie sa fie  1 si este : " << car2 << endl;
}
```

c) Spre deosebire de funcțiile standard, funcțiile membre ale unei clase pot fi definite ca funcții **inline** în mod implicit sau explicit. Pentru claritate, considerăm definiția unei funcții denumită InitData() membră a unei clase numită DATA_CALENDARISTICA.
Dacă definiția funcției se face în interiorul clasei, se spune că funcția InitData() este definită implicit de tipul **inline** și acest lucru se scrie astfel:

```
class DATA_CALENDARISTICA {
        int  zi, luna, an;
public:
        void InitData (int zz, int ll, int aa)  {zi = zz; luna = ll; an = aa;}
};
```

Dacă definiția funcției se face în exteriorul clasei, se spune că funcția InitData() este definită explicit de tipul **inline** și acest lucru se scrie astfel:

```
class DATA_CALENDARISTICA {
        int  zi, luna, an;
public:
        inline void InitData (int zz, int ll, int aa);          // Prototipul funcției
};
void DATA_CALENDARISTICA :: InitData (int zz, int ll, int aa)  // Definiția funcției
        {zi = zz; luna = ll; an = aa;}
```

Observație. În general, nu se recomandă includerea în corpul clasei a unei definiții de funcție compusă din multe instrucțiuni.