4. UTILIZAREA POINTERILOR ȘI REFERINȚELOR. ELEMENTE PRELIMINARII DESPRE FUNCȚII

În acest capitol se vor aborda principalele elemente despre pointeri și referințe. De asemenea se va începe studiul funcțiilor. Se vor trata următoarele topici: pointeri și referințe, pointeri la pointeri, elemente despre aritmetica pointerilor, pointeri și constante, pointeri de tip **void**, diferențe între masiv și pointer, definirea unui pointer la o funcție, trei moduri de pasare a argumentelor unei funcții.

4.1. Pointeri și referințe. Pointeri la pointeri

Se consideră următoarele trei instrucțiuni:

```
int i = 10 ;
int *p = &i ;
int **pp = &p ;
```

Dacă se consideră ca variabila pp a fost implantată în memoria calculatorului la adresa 3000, atunci reprezentarea grafică din Fig. 4.1 a celor trei instrucțiuni lămurește complet noțiunea de pointer și referință, respectiv pointer la pointer.

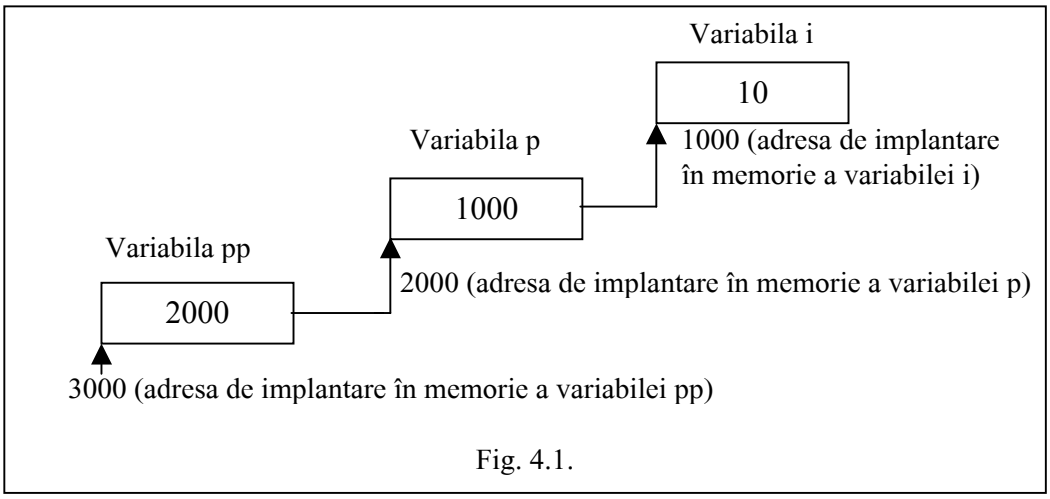

Fig. 4.1.

Referința variabilei pp, notată &pp va fi deci 3000. Conținutul variabilei pp (pointer la pointer, exprimat prin cele două asteriscuri) este 2000. Dar 2000 este referința variabilei p, notată &p. Conținutul variabilei p este 1000 și acesta reprezintă adresa de implantare a variabilei întregi i a cărei valoare este 10.

Când se scrie o line sursă în care se dorește implantarea în memorie a unui text, ca de exemplu:

```
char  *ps = "Acesta este un sir",
```

înseamnă că acesta va fi dispus în memorie octet după octet, variabila ps punctând pe începutul textului, în cazul nostru pe litera A.

Pentru adresarea elementelor șirului se pot folosi două notații:

1. ps[0] = `A`, ps[3] = `s`, ps[10] = `e` sau
2. *ps, *(ps + 3), *(ps + 10).

Observație. Datorită priorității mai mari a operatorului unar ***** în raport cu operatorul **+**, în notația de la punctul 2 a fost nevoie de paranteze. În lipsa lor, de exemplu, expresia *p + 3 ar fi avut valoarea `A` + 3 = 0x41 + 3 = 0x44 = `D`.

4.2. Elemente referitoare la aritmetica pointerilor

Considerăm că șirul punctat de ps începe la adresa 1000, Fig. 4.2. Atunci valoarea expresiei ps + 3 este 1003, deoarece fiecare element punctat de ps are lungimea egală cu 1, fiind de tip **char**.

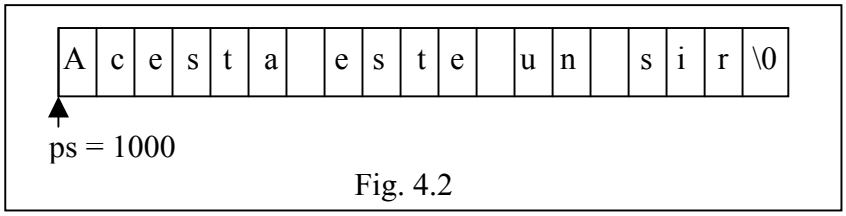

Fig. 4.2

Fie masivul unidimensional

float vm[6] ;

format din șase elemente în virgulă mobilă, simplă precizie. Dacă se consideră că vm[0] este implantat în memorie la adresa 2000 și că fiecare element ocupă patru octeți, vm[4] se va afla la adresa 2016, Fig. 4.3.

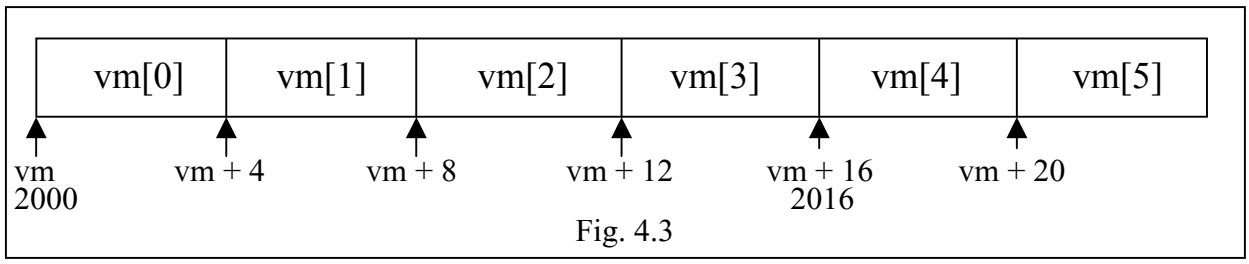

Fig. 4.3

Fie doi pointeri:

float *pm0, *pm1;

inițializați cu adresele (referințele) elementelor vm[0] și respectiv vm[4], adică:

pm0 = &vm[0];

pm1 = &vm[4];

Rezultatul scăderii pm1 - pm0 va da valoarea 4, nu 8 sau 16, deci *numărul elementelor de tipul respectiv* și nu al octeților.

Exemplul din programul următor conține câteva modalități de utilizare a pointerilor și referințelor.

Exemplul 4.1.

```
// Program P4_1.CPP Modalități de utilizare a pointerilor și referințelor
# include <iostream.h>
# define PRINT(x) cout << #x << " = " << x << endl;

int  a[] = {0, 1, 2, 3, 4};          // Dimensiunea masivului global "a" se deduce din
                                     // numărul elementelor de inițializare
void main (void)
{       int  *p ;
        for (int i = 0; i <= 4; i++)
                PRINT (a[i]);                // Se afișează conținutul elementelor a[0], a[1], ...
        for (p = &a[0]; p <= &a[4]; p++)
                PRINT (*p);                  // Se afișează conținutul celulelor punctate de p
        for (p = a, i = 0; p+i <= a+4;  p++, i++)
                PRINT (*(p+i));              // Se afișează conținutul celulelor punctate de p+i
        for (p = a+4; p >= a; p--)
                PRINT (*p);                  // Se afișează conținutul lui a parcurs în sens invers
 }
```

Observație. Operatorul # într-o macrodefiniție este un operator de substituire. Dacă x devine a[0], acesta se va substitui peste tot și în urma preprocesării va rezulta următorul șir:

cout << "a[0]" << " = " << a[0] << endl;

care este o linie sursă corectă.
Comentariile din program sunt suficiente pentru înțelegerea completă a programului.
Pointerii pot fi asociați diferitelor tipuri de variabile, inclusiv obiecte ale unei clase. Pentru exemplificare considerăm declarația unei clase cu numele CLS:

```
class CLS {
        // Declarația clasei
    } ;
```

Presupunem că în funcția main() se introduc instrucțiunile:

```
CLS   obiect ;          // Se creează obiectul obiect
CLS *pointer_la_obiect  =  &obiect ;
```

Variabila pointer_la_obiect punctează adresa lui obiect și este un pointer de tipul CLS (așa cum **double** *pp; ar fi un pointer tip **double**).

4.3. Pointeri la variabile de tip *const*. Pointeri de tip *const*

În C++, o constantă este o "variabilă" de tipul "citește numai", ca de exemplu **const double** PI = 3.14; Tipul **const** a fost introdus și în C, după adoptarea standardului ANSI. Până la apariția standardului ANSI, în C, pentru declararea constantelor se folosea numai instrucțiunea **#define** a preprocesorului. Actualmente, pentru declararea constantelor atât în C cât și în C++, se utilizează cuvântul cheie **const** și instrucțiunea **#define**. Dacă în C, o constantă nu trebuie neapărat inițializată, în C++, acest lucru este absolut obligatoriu. De exemplu, dacă v este o constantă, în C putem scrie

```
const int   v ;          /* Declararea constantei */
```

pe când în C++ trebuie scris:

```
const int   v = 123;     // Declararea și inițializarea constantei v
```

Se putea scrie și **# define** v 123.
Fie definiția următoare: **const int** j = 10; deci j este o variabilă nemodificabilă. O încercare de referire de forma **int** *p = &j; nu ar fi admisă, p fiind un pointer de tip "non const". În schimb, construcția **const int** *pci = &j; desemnează variabila pci drept pointer la o constantă de tip **int**. În acest caz variabila pci (adică referința la j) poate fi modificată, nu și *pci (adică valoarea lui j).
Instrucțiunile:

```
int  k ;
int *const  cpi = &k ;
```

informează compilatorul că variabila cpi este un pointer de tip **const** care punctează pe o variabilă k modificabilă. În acest caz *cpi va putea fi modificat, k fiind o variabilă "non const", în schimb cpi nu. O încercare de adresare a lui j de mai sus de genul **int *const** cpi = &j; ar fi semnalată ca eronată de către compilator, deoarece j este nemodificabilă, iar cpi este un pointer la o variabilă modificabilă.

4.4. Pointeri de tipul void

Tipul **void** semnifică cel puțin două lucruri:

1. O funcție precedată de **void** nu va întoarce nici un rezultat. De exemplu, funcția declarată prin **void** functie (**int** v1, **float** v2) în corpul căreia nu trebuie să apară instrucțiunea **return** nu trebuie să întoarcă nici un rezultat. O încercare de apelare a acestei funcții de genul: rezultat = functie (10, 1.0); va fi semnalată drept eroare de către compilator.
2. Pentru declararea unui pointer de tipul generic. Considerăm următoarea funcție care are ca parametru de intrare un astfel de pointer:

   ```
   int fct (void *p) {... }
   ```

   În momenul apelării ei, p va puncta o adresă de memorie care poate să fie a unei variabile de tip **int**, sau **short**, sau **long** etc. De exemplu, un apel pentru cazul **int** arată astfel:

```
        int  i ;
        ...
        fct (&i) ;
```
Deci, în momentul pasării controlului funcției f(), datorită precizării efectuate, tipul pointerului va fi cunoscut și acceptat.

4.5. Masive de pointeri. Liste

Pentru realizarea unor liste și pentru înlănțuirea elementelor acestora, în practica curentă se folosesc pointerii și masivele de pointeri. Listingul din programul următor prezintă un exemplu de întocmire a unei liste utilizând pointeri și masive de pointeri la pointeri.

Exemplul 4.2.

```
// Program P4_2.CPP Lanț de structuri. Modalități de utilizare a pointerilor și referințelor
# include <iostream.h>
struct S {
        char *sir;
        int   i;
        struct S *next;
};
void rocada(struct S *, struct S *);
// Programul principal
void main (void)
{ static struct S lant[ ] = {           // Definiția unui lanț de celule
        {"elem1", 0, lant+1},
        {"elem2", 1, lant+2},
        {"elem3", 2, lant}
  };
  struct  S *p[3];                      // Definim un masiv de 3 pointeri la structura de tip S
  for (int i = 0; i < 3; i++)
          p[i] = lant[i].next;          // Inițializarea celor trei pointeri de tip structură
     cout << "Rezultatele inaintea primei rocade:" << endl;
     cout << "p[0]->sir = " << p[0]->sir << endl;                    // Se va afișa elem2
     cout << "(*p)->sir = " << (*p)->sir << endl;                    // Se va afișa elem2
     cout << "(**p).sir = " << (**p).sir << endl;                    // Se va afișa elem2
     cout << "(*p)->next->sir = " << (*p)->next->sir << endl;        // Se va afișa elem3
 rocada (*p, lant);
     cout << "Rezultatele dupa prima rocada:" << endl;
     cout << "p[0]->sir = " << p[0]->sir << endl;                    // Se va afișa elem1
     cout << "(*p)->sir = " << (*p)->sir << endl;                    // Se va afișa elem1
     cout << "(*p)->next->sir = " << (*p)->next->sir << endl;        // Se va afișa elem1
 rocada (p[0], p[2]);
     cout << "Rezultatele dupa a doua rocada:" << endl;
     cout << "p[0]->sir = " << p[0]->sir << endl;                    // Se va afișa elem2
     cout << "(*++p[0]).sir = " << (*++p[0]).sir << endl;            // Se va afișa elem3
     cout << "++(*++(*p)->next).sir = " << ++(*++(*p)->next).sir << endl;  // Se va afișa lem2
     cout << "++(*++(*p)->next).sir = " << ++(*++(*p)->next).sir << endl;  // Se va afișa lem3
}
void rocada (struct S *p1, struct S *p2)
{       struct S *temp;
```

```
*temp = *p1;
*p1 = *p2;
*p2 = *temp;     }
```

Înainte de executarea primei instrucțiuni de afișare, lista va atăta ca în Fig. 4.4.

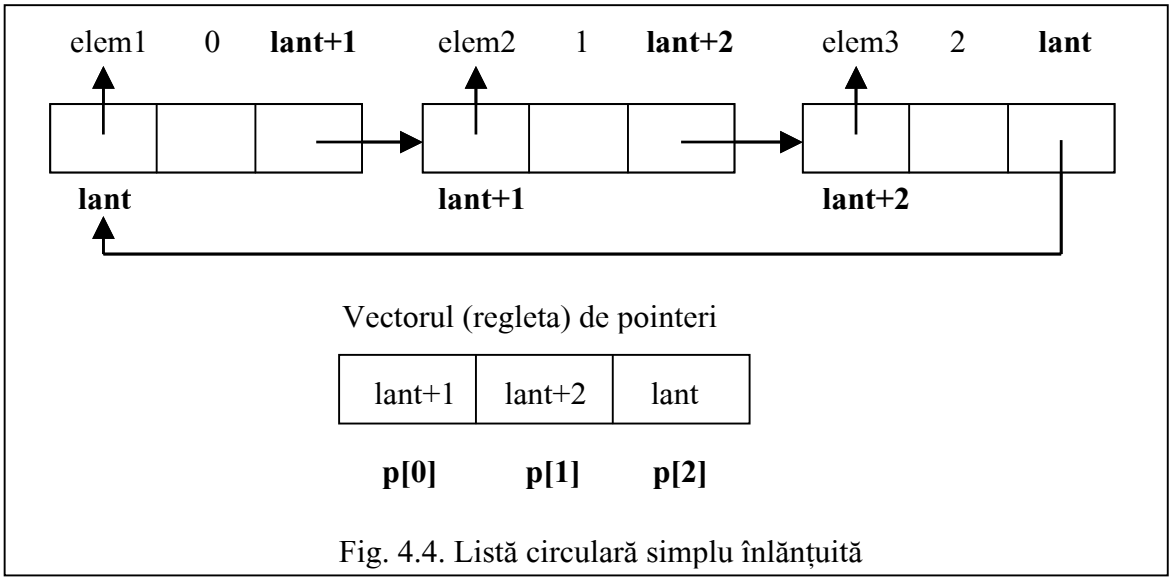

Fig. 4.4. Listă circulară simplu înlănțuită

Primul element al vectorului (regletei) de pointeri, p[0] conține adresa lant+1, adică adresa celui de al doilea element al listei circulare, p[1] conține adresa lant+2, adică adresa celui de al treilea element al listei circulare, iar p[2] conține adresa lant, adică adresa primului element al listei circulare.

Expresia p[0] -> sir înseamnă de fapt adresa de început a literalului "elem2".

Grafic, evoluția vectorului (regletei) de pointeri este ilustrată în Fig. 4.5 a, b și c.

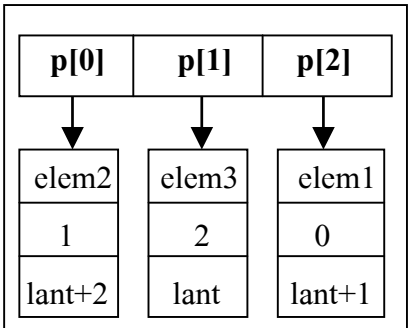

Fig. 4.5a
Înainte de prima rocadă

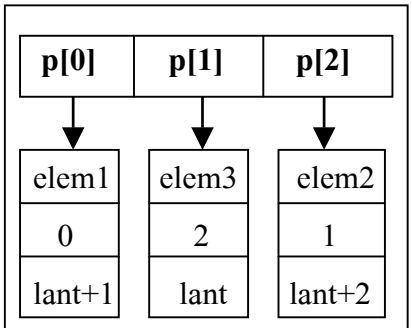

Fig. 4.5b
După prima rocadă

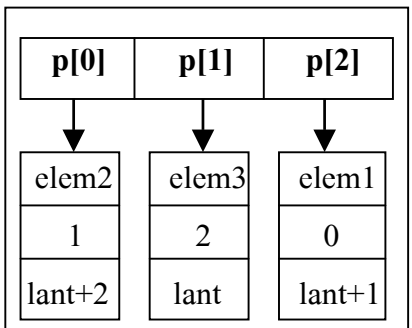

Fig. 4.5c
După cea de-a doua rocadă

Expresia (*p) -> sir înseamnă că se preia conținutul elementului p[0] care este o adresă la o celulă de înlănțuire, celulă ce are trei componente; -> sir înseamnă referirea la câmpul sir al acelei celule. În Fig.4.5 a și c, p[0] punctează de fapt șirul "elem2", iar în Fig.4.5b, șirul "elem1". Construcția (**p).sir se traduce astfel: (**p) = *(*p) = *p[0] = lant+1, deci o adresă; atunci, (lant+1).sir desemnează șirul "elem2". În sfârșit, expresia (*p) -> next -> sir se va descifra astfel: *p înseamnă p[0], o adresă a unei celule; p[0] -> next reprezintă conținutul punctatorului next al celulei punctate de p[0]. Să considerăm că p[0] punctează celula din Fig. 4.5a care conține "elem2". Atunci next va puncta celula următoare, lant+2. Din această celulă reținem câmpul sir, care este la rându-i o adresă la șirul "elem3".

În urma rulării acestui program se obțin următoarele rezultate:

- înaintea primei rocade: **elem2 elem2 elem2 elem3**

- după prima rocadă: **elem1 elem1 elem1**
- după a doua rocadă: **elem2 elem3 lem2 lem3**

Fie următorul masiv unidimensional cu zece elemente în virgulă mobilă: **float** masiv[10]; Primul element, masiv[0] este totodată și adresa de început a masivului. O expresie de tipul masiv[i] înseamnă indexarea elementului i al masivului. Folosind pointeri, expresia masiv[i] se scrie sub forma *(masiv+i). În mod similar, pentru masivul bidimensional m[10][20], în notația prin pointeri va fi : *(*(m + i) + j), aceasta deoarece m va fi considerat un masiv de 10 vectori, fiecare cu câte 20 elemente. Mai întâi m+i selectează al i-lea vector al masivului, iar j, al j-lea element din acel vector.

4.6. Diferențe dintre masiv și pointer

Considerăm următoarele definiții:

```
char  *sir = "sir1";
char   s[ ] = "sir2";
```

Se creează în memorie două șiruri. Pointerul *sir, va puncta pe litera s din literalul "sir1". Deci, *sir conține adresa de început a șirului sir1. Variabila s[ ] este un masiv dimensionat în mod dinamic, în funcție de numărul "inițializatorilor" (în acest caz, patru litere `s`, `i`, `r`, `2` și "terminatorul", `\0`). Totuși s[ ] înseamnă de fapt &s[0], adică adresa primului element al masivului. Nu se acceptă s++, în schimb se acceptă sir++. Nu se admite atribuirea s = sir; dar atribuirea sir = s este acceptată. Șirurile de tip pointer (cazul lui *sir), sunt inițializate numai o singură dată la pornirea executării programului. Mai concret, fie funcția: f() { char *s = "abcd"; puts(s); }. Compilatorul va aloca cinci octeți pentru literele `a`, `b`, `c`, `d` și `\0`. La lansarea în execuție a programului va fi creat spațiu pentru toate constantele, deci și pentru constanta "abcd" terminată cu `\0`. La fiecare apel al funcției f() nu se va realoca spațiu pentru întregul literal, efectuându-se inițializarea, ci se va aloca doar spațiu pentru păstrarea pointerului la acest literal. În acest spațiu, compilatorul își va copia adresa de memorie unde a fost asamblat (o singură dată) literalul "abcd".

4.7. Definirea unui pointer la o funcție

În exemplul de mai sus am întâlnit următorul prototip:

```
void rocada (struct S *p1, struct S *p2);
```

Funcția rocada() are în lista de intrare doi pointeri de tipul **struct S** și nu întoarce nici un rezultat. Deoarece în această descriere denumirile p1 și p2 sunt redondante, pentru compilator ar fi fost suficient un prototip de genul:

```
void rocada (struct S *, struct S *);
```

Se consideră următoarea formă de prototip:

```
unsigned (*f) ();
```

În acest prototip, *f este un pointer la o funcție fără parametri de intrare, care întoarce un întreg fără semn.

Prototipul:

```
unsigned *f ();
```

este declarația unei funcții fără parametri de intrare, care întoarce un pointer la un întreg fără semn.

Tabelul următor prezintă câteva exemple de expresii în care apar pointeri:

Expresia	Semantica expresiei
(*x)	x este un pointer
(*x)[ ]	x este un pointer la un masiv
(*(*x)[ ])	x este un pointer la un masiv de pointeri
(*(*x)[ ])()	x este un pointer la un masiv de pointeri la funcții
(*(*(*x)[ ])())	x este un pointer la un masiv de pointeri la funcții care întorc pointeri

int (*pa)[ ];	pa este un pointer la un masiv de întregi
int (**p1)[ ];	p1 este un pointer la un pointer la un masiv de întregi
int (**p2)( );	p2 este un pointer la un pointer la o funcție
int *(*p3)[ ];	p3 este un pointer la un masiv de pointeri la întregi
int (*p4)[ ][ ];	p4 este un pointer la un masiv de masive de întregi, adică un masiv bidimensional
int *(*t)( );	t este un pointer la o funcție care întoarce un pointer la un întreg
int **fpp();	fpp este o funcție care întoarce un pointer la un pointer la un întreg

Exemplu de utilizare a pointerilor la funcții

Pentru a exemplifica noțiunea de pointer la funcție, considerăm existența unui vector de pointeri în care fiecare element al acestui vector conține adresa de început a unei funcții. Fiecare funcție va trebui să realizeze o acțiune a unui automat finit. Considerăm că la/din acest vector se pot adăuga/elimina elemente. Alegerea funcției adăugare/eliminare se face conform valorii unei variabile de stare a automatului finit.

Creăm o clasă denumită VECTOR_FUNCTII care în secțiunea **private** conține variabila nr_efectiv_fct ce conține numărul de funcții existente la un moment dat în vector. Tot în secțiunea **private** se află și vectorul propriu-zis de pointeri la funcții, denumit **void** (*ptr_la_fct [nr_max_fct])(). Numărul maxim de funcții, notat nr_max_fct este ales de programator. În secțiunea publică a clasei se declară următoarele funcții membre:

- AdaugFct (), o funcție prin care se adaugă un element la vector, deci un pointer la o nouă funcție;
- EliminFct (), o funcție de eliminare a unui element din vector;
- UltimaFct (), o funcție cu care se află care este numărul efectiv de funcții existente la un moment dat în vector;
- ExecFct (**int** nf), un "selector" destinat lansării funcției cu numărul nf;
- ExecMultipla(), o funcție care declanșează executarea tuturor funcțiilor existente în vector.

Listingurile definiției clasei și ale programului care operează cu această clasă sunt prezentate în exemplul următor:

```
// Exemplul 4.3.  Program VECT_FCT.H
// Dfinirea clasei VECTOR_FUNCTII
# ifndef FR_TIME
# define FR_TIME
const int nr_max_fct = 10;
class VECTOR_FUNCTII {
        int nr_efectiv_fct;                          // Numărul efectiv al funcțiilor din masiv
        void (*ptr_la_fct [nr_max_fct])();           // Un masiv de pointeri
        void eroare (char *mesaj1, char *mesaj2 = " ");
public:
        VECTOR_FUNCTII () ;                          // Declararea constructorului
        void  AdaugFct (void (*pf)() );
        void  EliminFct (int nf);
        int    UltimaFct () { return nr_efectiv_fct;}
        void  ExecFct (int nf);
        void  ExecMultipla ();
};
# endif
// Program P4_3.CPP  - Metode (funcții) lansate la momentul execuției
```

```
# define TEST
# include <stdio.h>
# include <stdlib.h>
# include "vect_fct.h"
void VECTOR_FUNCTII :: eroare (char *mesaj1, char *mesaj2) {
        fprintf (stderr, "VECTOR_FUNCTII eronat: %s %s\n", mesaj1,mesaj2);
        exit (1);
}
VECTOR_FUNCTII :: VECTOR_FUNCTII () {     // Constructorul clasei
        nr_efectiv_fct = 0;
        for (int i = 0; i < nr_max_fct; i++)
                ptr_la_fct [i] = NULL;
}
void VECTOR_FUNCTII :: AdaugFct (void (*pf)() ) {
        if (nr_efectiv_fct >= nr_max_fct )
                eroare ("***AdaugFct: nu mai exista spatiu in vector");
        ptr_la_fct [nr_efectiv_fct++] = pf;
}
void VECTOR_FUNCTII :: EliminFct (int nf) {
        if (nf < 0 || nf >= nr_max_fct )
                eroare ("***EliminFct: depasire de indici");
        for (int i = nf; i < nr_efectiv_fct; i++)
                ptr_la_fct [i] = ptr_la_fct [i+1];
        ptr_la_fct [nr_efectiv_fct] = NULL;          // Ultimul pointer devine NULL
        nr_efectiv_fct --;                           // Numărul efectiv scade cu 1
}
void VECTOR_FUNCTII :: ExecFct (int nf) {
        if (nf < 0 || nf >= nr_max_fct )
                eroare ("***ExecFct: indicele in afara limitelor");
        (*ptr_la_fct [nf]) ();                       // Se apelează funcția dorită
}
void VECTOR_FUNCTII :: ExecMultipla () {
        for (int i = 0; i < nr_efectiv_fct; i++)
                (*ptr_la_fct [i]) () ;
}
int neff;
# ifdef TEST
# define def_fct(NF,ARG)          void NF() {puts("Execut functia: " #NF);ARG--;}
static int      a = 10;
def_fct(alfa,a);  def_fct(beta,a);  def_fct(gama, a);
def_fct(miu, a);  def_fct(niu, a);  def_fct(omega,a);
void main ()
{
 VECTOR_FUNCTII  regleta;        // Se creează obiectul "regleta"
 regleta.AdaugFct (alfa);        // În vectorul funcțiilor se adaugă 6 obiecte
 regleta.AdaugFct (beta);
 regleta.AdaugFct (gama);
 regleta.AdaugFct (miu);
```

```
  regleta.AdaugFct (niu);
  regleta.AdaugFct (omega);
  neff = regleta.UltimaFct();          // Se determină numărul efectiv de pointeri din masivul de
                                       // pointeri la funcții
  printf ("a = %d\n", a);
  printf ("Numarul de functii efectiv ramase = %d\n", neff);
  printf ("Se executa toate functiile\n");
  regleta.ExecMultipla ();             // Se execută fiecare funcție prezentă în acel moment în vector
  printf ("a = %d\n", a);
  regleta.EliminFct (3);          // Se elimină elementul al patrulea, deci pointerul la funcția miu
  printf ("A fost eliminata functia \"miu\"\n");
  printf ("Acum se executa functia \"niu\"\n");
  regleta.ExecFct (3);            // Elementul al patrulea devine funcția niu, care va fi executată
  printf ("Se executa toate functiile ramase\n");
  regleta.ExecMultipla ();        // O nouă execuție a tuturor funcțiilor din vector
  printf ("Se eliminaa toate functiile\n");
  for (int i = 0; i <= neff + 1; i++)
     regleta.EliminFct (0);       // Deoarece în vector mai erau doar 5 funcții, după acest ciclu, neff=-1
     neff = regleta.UltimaFct();
     printf ("Numărul de functii efectiv ramase = %d\n", neff);
}
# endif    // TEST
```

În secțiunea privată a clasei se află și funcția eroare() care verifică dacă numărul funcției apelate este valid sau nu.

Corpul propriu-zis al unei funcții lansate în execuție conține numai instrucțiunea de anunțare a unui mesaj de tipul "Execut funcția ... numele ei". În program s-a folosit instrucțiunea **# define** pregătită să preia prin substituție parametrii NF (numele funcției) și ARG (argumentul funcției). Macrodefiniția are forma:

```
# define def_fct(NF,ARG) void NF() {puts("Execut functia: " #NF); ARG--;}
```

Substituirea cu valori particulare a macrodefiniției **# define** def_fct(NF,ARG) are loc la momentul execuției unui apel de forma: def_fct (alfa, a);

Programul creează în mod "dinamic" prin intermediul acestor apeluri un număr de șase funcții alfa, beta, gama, miu, niu, omega. Cu funcția UltimaFct() se determină apoi numărul de pointeri "nenuli" din vector (regletă). În continuare se declanșează execuția tuturor celor șase funcții, apărând pe rând mesajele:

Execut funcția: alfa
Execut funcția: beta
. . . . . .

Funcționarea programului poate fi urmărită în continuare fără probleme.

4.8. Elemente preliminare despre funcții

În versiunile de C aliniate la un standard ANSI, orice funcție trebuie "anunțată" printr-un prototip. Declararea unei funcții prototip se face conform următorului format:

```
tip      nume_functie (tip_arg1, tip_arg2, ...) ;
```

unde: tip = tipul valorii întoarse de funcție;
tip_arg1, tip_arg2,... = tipurile argumentelor funcției.

Exemplu: P4_4.CPP. Programul următor va determina compilatorul să emită un mesaj de eroare deoarece acesta încearcă să apeleze funcția func() având al doilea argument de tip **int**, în loc de **float**, cum a fost declarat în funcția func():

```
# include <stdio.h>
float   func ( int, float ) ;                   /* Prototipul funcției func( ) */
void    main ( void ) {
        int     x, y ;
        x = 10;   y = 10 ;
        func (x, y) ;                           /* Se afișează o nepotrivire */
}
float   func (x, y)                             /* Parametrii funcției sunt: */
        int     x ;                             /* x - întreg */
        float   y ;                             /* y - real   */
{
  printf ("%f", y / (float) x ) ;
}
```

Apelul unei funcții înseamnă referirea funcției, împreună cu valorile actuale ale parametrilor formali, precum și preluarea valorii returnate, dacă este necesar.

Din cele de mai sus se observă că folosirea funcțiilor prototip ne ajută la verificarea corectitudinii programelor, deoarece nu este permisă apelarea unei funcții cu alte tipuri de argumente, decât tipul celor declarate.

În C și C++, transmiterea argumentelor de la funcția apelantă spre funcția apelată se face în două moduri: prin valoare sau prin adresă. Acesta din urmă mod, la rându-i se întâlnește sub două forme: pointer și referință.

4.8.1. Apelul prin valoare

În cazul transmiterii argumentului prin valoare, se realizează copierea (atribuirea) valorilor fiecărui argument în (la) câte un parametru formal al funcției apelate.

```
// Program P4_5.CPP   Apelul prin valoare
# include <stdio.h>
long f(int a, double b);                // Prototipul funcției
void main ()
{       int a =10;   double b = 11.2;  long rez;
        printf ("a = %d\n", a);  printf ("b = %f\n", b);
        rez = f(a,b);
        printf ("rez = %ld\n", rez);
}
long f(int k, double v)
{ return (long) (k+v); }
```

Prin această metodă, se observă că, valorile variabilelor a și b sunt atribuite variabilelor k și v într-o zonă de memorie alocată de compilator la activarea funcției f(). Asupra acestor variabile au loc conversiile corespunzătoare și în final va rezulta valoarea 21 ce va fi pasată funcției main() tot prin valoare.

4.8.2. Apelul prin pointeri

Dacă transmiterea argumentului se realizează prin adrese, atunci la apelul funcției în loc de valori se folosesc adrese, iar în definiție parametrii formali se declară ca pointeri.

Exemplu: Programul următor arată modul de apel al unei funcții swap() care interschimbă valorile a două variabile reale.

```
// Program P4_6.CPP   Apelul prin pointeri
# include <stdio.h>
```

```
void    swap (float *x, float *y) ;      // Prototipul funcției
void    main ( void )  {
        float    x, y ;
        scanf ( "%f, %f", &x, &y ) ;
        printf ( " x = %f,   y = %f \n " , x , y ) ;
        swap ( &x, &y ) ;    // Apelul funcției swap () având ca argumente, adresele lui x și y
        printf (" x = %f,    y = %f \n " , x , y ) ;
}
void    swap (float *x, float *y)
{       float    *temp ;
        *temp = *x ;                  //  La adresa temp se copiază valoarea de la adresa  x
        *x = *y ;                     //  Valoarea de la adresa y este copiată la adresa x
        *y = *temp ;                  //  La adresa y se copiază valoarea de la adresa temp
}
```

Se observă că parametri formali ai funcției swap() sunt pointeri la **float**. Programul transferă prin &x și &y adresele lui x și y funcției swap() și nu valorile lui x și y. Pentru a avea acces la valorile realilor din funcția swap() se recurge la mecanismul de extragere a conținutului (*dereferencing*) prin folosirea operatorului * în fața variabilelor dorite.

4.8.3. Apelul prin referință (nume)

Acest mecanism este ilustrat tot cu ajutorul funcției swap() din exemplul de mai sus. În acest caz programul P4_6.CPP devine:

```
// Program P4_7.CPP   Apelul prin referințe
    # include <stdio.h>
    void    swap (float &x, float &y) ;    // Prototipul funcției
    void    main ( void )  {
            float    x, y ;
            scanf ( "%f, %f", &x, &y ) ;
            printf ( " x = %f,   y = %f \n " , x , y ) ;
            swap ( x, y ) ;    // Apelul funcției swap () având ca argumente, referințele lui x și y
            printf (" x = %f,    y = %f \n " , x , y ) ;
    }
    void    swap (float &x, float &y)
    {       float    temp ;
            temp = x ;                  //  La adresa temp se copiază valoarea de la adresa  x
            x = y ;                     //  Valoarea de la adresa y este copiată la adresa x
            y = temp ;                  //  La adresa y se copiază valoarea de la adresa temp
    }
```

O referință este un alt nume (*alias*) dat obiectului. Spre deosebire de pointer, odată ce am adresat un obiect prin intermediul unei variabile **&referinta**, ea nu mai poate fi schimbată spre un alt obiect. Altfel spus nu vom putea despărți referința la acel obiect de respectivul obiect. De fapt referința "ascunde" informația asociată obiectului respectiv.

Referințele vor fi înlocuite de pointeri când trebuie să adresăm obiectele pe "dinăuntrul" lor. Dacă însă dorim să ne referim la obiect ca întreg, vom prefera referința, deci ca și când ne-am situa la "suprafața" obiectului.

Reluăm mecanismele de mai sus, dar de această dată pasând obiecte sau adrese la obiecte. Programul următor conține un obiect OB al clasei ADHOC, clasă care are două variabile private i și j asupra cărora acționează trei funcții membre ale clasei, ale căror denumiri semnifică modul lor de apel.

```
// Program P4_8.CPP  Mecanisme de pasare a argumentelor la intrare și ieșire
# include <stdio.h>
class ADHOC {
        int      i, j;
public:
        ADHOC (int u = 0, int v = 0)  { i = u; j = v;}          // Constructorul clasei
// Declarații ale funcțiilor membru
        ADHOC  ApelPrinValoare (ADHOC OB);
        ADHOC  *ApelPrinPointer (ADHOC *OB);
        ADHOC  &ApelPrinReferinta (ADHOC &OB);
        void Imp (char *mesaj = " ")                            // Definiția funcției membru Imp()
        { printf ("%s ==> i = %d  j = %d\n", mesaj, i, j); }
};
ADHOC ADHOC::ApelPrinValoare (ADHOC OB)
        {       OB.i = OB.j = 100;
                return OB;                              // Se returnează obiectul OB
        }
ADHOC *ADHOC::ApelPrinPointer (ADHOC *OB)
        {       OB->i = OB->j = 10;
                return OB;                              // Se returnează un pointer la obiectul OB
        }
ADHOC &ADHOC::ApelPrinReferinta (ADHOC &OB)
        {       OB.i = OB.j = -100;
                return OB;                              // Se returnează o referință la obiectul OB
        }
void main ()
{
  ADHOC ob1, ob2;                               //Se creează două obiecte denumite ob1 și ob2
  ob1.ApelPrinValoare (ob2).Imp("Rezultatul in urma apelului functiei ApelPrinValoare()");
  ob1.ApelPrinPointer (&ob2)->Imp("Rezultatul in urma apelului functiei ApelPrinPointer()");
  ob1.ApelPrinReferinta (ob2).Imp("Rezultatul in urma apelului functiei ApelPrinReferinta()");
}
```

Funcția ApelPrinValoare() al cărei prototip este ADHOC ApelPrinValoare (ADHOC OB); permite modificarea datelor private ale obiectului OB, întrucât aceasta este o funcție membră a clasei ADHOC. Voaloarea întoarsă de această funcție va fi copiată în exteriorul obiectului OB. Când se execută instrucțiunea ob1.ApelPrinValoare (ob2), cele două variabile i și j devin egale cu 100. Funcția Imp() afișează valorile variabilelor acestei clase.

Funcția ApelPrinPointer() al cărei prototip este ADHOC *ApelPrinPointer (ADHOC *OB); adresează obiectul de modificat prin intermediul unui pointer la acel obiect și returnează adresa obiectului modificat.

Funcția ApelPrinReferință() al cărei prototip este ADHOC &ApelPrinReferinta (ADHOC &OB); adresează obiectul de modificat prin referința la acel obiect și returnează adresa obiectului modificat. Se observă că pasarea parametrilor prin referință conduce la folosirea unei notații identice cu cea întrebuințată la apelul prin valoare.

Pentru fiecare formă de apel se observă notația:

```
ob1.ApelPrin...(ob2).Imp("Rezultatul in urma apelului functiei ApelPrin...() ");
```

în care, după numele obiectului ob1, se înlănțuie prin intermediul operatorului "." două funcții membre ale clasei ADHOC. Controlul este dat pe rând fiecăreia dintre acestea.