Tipul *pointer*

Fiecărui obiect dintr-un program (variabilă, constantă, funcție), i se alocă un spațiu de memorie exprimat în număr de locații (octeți, *bytes*), corespunzător tipului de date prin care a fost definit obiectul respectiv.

Localizarea (găsirea) unui obiect în memorie se face prin intermediul adresei la care a fost memorat.

Adresa este un număr întreg, fără semn si reprezintă numărul de ordine al unei locații de memorie; în C adresele sunt de regulă de tip long int , domeniul int fiind insuficient.

Datele de tip *adres*ă se exprimă prin tipul *pointer*. În limba română, se traduce prin *referin*ță*, reper, localizator, indicator de adres*ă.

	MEMORIE		
Adresa	*Locația pară*	*Locația impară*	*Variabila*
0000	11110000	10101010	. . . .
0002	11010100	10101110	. . . .
. . . . .	1100 1010	1100 1010	. . . .
0026	00000000	10000000	x = 128
0028	00000000	11110000	y = 240
0030	00000000	00000000	z = 1.0
0032	00000000	00000000	
0034	00000000	00000000	
0036	11110000	00111111	
0038	1100 1010	1100 1010	. . . .

În tabelul de mai sus, este reprezentată o zonă de memorie, în care sunt stocate variabilele x, y, z, declarate astfel:

```
int *px, *py, x=128, y=240;
double *pz, z=1.0;
```

Se rezevă doi octeți pentru variabilele de tip întreg si 8 octeți (64 de biți) pentru variabila z, de tip real, dublă precizie.

Variabilele px, py, pz, sunt de tip pointer (adresă); ele pot furniza adresele variabilelor de tip corespunzător, dacă sunt asociate cu acestea prin atribuiri de forma:

```
px = &x; py = &y; pz = &z;
```

Din px = &x si din tabelul de mai sus, rezultă px = 0026;
Analog, py = 0028, pz = 0030.

Observații:

- Variabilele de tip pointer se declară cu prefixul '*' : *p, *u, *v ;

- Variabilele de tip pointer se asociază cu adresa unui obiect prin atribuiri de forma p = &var, unde var este o variabilă declarată; semnul '&' arată că se consideră **adresa** lui var (care se transferă în pointerul p);

- O variabilă de tip pointer trebuie declarată astfel:

<tip de bază> *<identificator> ;

unde tipul de bază reprezintă tipul de date pentru care va memora adrese.

Exemple:

```
float *a1, var_1; //pointerul a1 va memora adrese pentru valori float
int *a2, var_2; // pointerul a2 va memora adrese pentru valori int
a1 = &var_1 ; // atribuire corectă, var_1 este de tip float
a2 = &var_2 ; // atribuire corectă, var_2 este de tip int
a2 = &var_1 ; // gresit , pentru că a2 se asociază cu variabile int
```

- Operația inversă, de aflare a valorii din memorie, de la o anumită adresă, se realizează cu operatorul * pus în fața unui pointer:

```
int *p, x, y;
x = 128; p = &x; // p = adresa lui x
y = *p; // y = valoarea de la adresa p, deci y = 128
y = *&x; // y = valoarea de la adresa lui x, deci y = 128;
```

Asadar, operatorul * are efect invers față de operatorul & si dacă se pun consecutiv, efectul este nul: x = *&x, sau p = &*p.

- Un pointer poate fi *fără tip*, dacă se declară cu *void*:

```
void *p ;
```

caz în care p poate fi asociat cu orice tip de variabile. Exemple:

```
int x;
float y;
void *p;
. . . . . .
p = &x ; // atribuire corectă, p este adresa lui x
p = &y ; // atribuire corectă, p este adresa lui y
```

- O variabilă pointer poate fi inițializată la fel ca oricare altă variabilă, cu condiția ca valoarea atribuită să fie o adresă:

```
int k ;
int *adr_k = &k; // pointerul adr_k este egal cu adresa lui k
```

Un pointer nu poate fi inițializat cu o adresă concretă, deoarece nu poate fi cunoscută adresa atunci când se scrie un program; adresa este dată de sistemul de operare după compilare, în funcție de zonele libere de memorie, disponibile în momentul execuției. Astfel, declarații de tipul:

```
p = 2000 ;
&x = 1200 ;
```

sunt eronate si semnalizate de compilator ca erori de sintaxă.

O expresie de tipul *<adresa> poate să apară si în stânga:
*adr_car = 'A' ; // caracterul A se memorează la adresa=adr_car.
Exemple de programe care utilizează pointeri:

1)
```
#include <stdio.h>
main ( )
{
int x = 77 ;
printf("Valoarea lui x este: %d \n", x);
printf("Adresa lui x este: %p \n", &x);
}
```

Funcția printf() utilizează specificatorul de format %p, pentru scrierea datelor de tip adresă.

2)
```
#include <stdio.h>
main ( ) // afisarea unei adrese si a valorii din memorie
{
char *adr_car, car_1='A' ;
adr_car = car_1; // pointerul ia valoarea adresei lui car_1
printf("Valoarea adresei: %p \n", adr_car);
printf("Continutul de la adr_car %c \n", *adr_car);
}
```

3)
```
#include <stdio.h>
main ( ) //perechi de instructiuni cu acelasi efect
{ int x=177, y, *p ;
y = x + 111; // y = 177+111=288
p = &x; y = *p + 111; // y = 177+111=288, acelasi efect
x = y ; // x = 288
p = &x; *p = y ; // x = 288, adică valoarea lui y
x++ ; // x = x+1, deci x = 299
p = &x ; (*p)++ ; // x = 299+1=300 }
```

Structuri de date: LISTE

1. Definiție

Multe aplicații impun ca informațiile prelucrate cu calculatorul să fie organizate sub forma unor liste de date.

O listă ordonată sau liniară este o structură de date omogenă, cu acces secvențial formată din elemente de acelasi tip, între care există o relație de ordine determinată de poziț ia relativă a elementelor; un element din listă conține o informație propriu-zisă si o informaț ie de legatură cu privire la elementele adiacente (succesor, predecesor); primul element din listă nu are predecesor iar ultimul nu are succesor.

De exemplu, în lista candidaților înscrisi la concursul de admitere, un element conține următoarele informații specifice:

- nume, prenume, inițiala tatălui;
- numarul legitimației de candidat;
- notele obținute la probele 1, 2;
- media la bacalaureat;
- media notelor;

Fiecare element al listei mai conț ine informații cu privire la poziția sa în listă (predecesorul si succesorul).

Nr. crt.	Numele şi prenumele	Nr. leg.	Nota 1	Nota 2	Media bac.	Media gen.
1	Popescu I. Ion Lucian	1/P	7.5	9.5	8.67	8.54
2	Anton Gh. Adrian Liviu	1/A	9.5	9	9.37	9.28
3	Moraru V. Vasile	5/M	8.5	9.5	9.89	9.22
4	Marașescu Gh. Liviu Costin	6/M	7.5	10	9.56	8.95
	. . . . .	. . .	. . .	. . .	. . .	. . . .

Fiecare informaț ie poate reprezenta o "cheie" de acces pentru căutare, comparații, reordonare etc. De exemplu, luând drept cheie media, se poate crea o nouă listă în care candidații să apară în ordinea descrescătoare a mediei.

Conținutul unei liste se poate modifica prin următoarele operații:

- adăugarea de noi elemente la sfârsitul listei;
- inserarea de noi elemente în interiorul listei;
- stergerea unui element din listă;
- schimbarea poziției unui element (modificarea unei înregistrări gresite);
- inițializarea unei liste ca listă vidă, fără elemente, în vederea completării ei;

Alte operații ce pot fi efectuate asupra listelor sunt cele de caracterizare, care nu modifică structura listelor, ci doar furnizează informații despre ele:

- determinarea lungimii listei (numărul de elemente);
- localizarea unui element care îndeplineste o anumită condiție;

Operații mai complexe:

- separarea în două sau mai multe liste secundare după un anumit criteriu;
- combinarea a două sau mai multe liste într-una singură;
- crearea unei liste noi, prin selectareaelementelor unei liste pe baza unui anumit criteriu.

2. Reprezentarea listelor în memorie

În memorie, listele pot fi reprezentate prin structuri statice (tablouri) sau prin structuri dinamice (liste înlănțuite), folosind pointeri.

În cazul **reprezentării statice**, elementele sunt indexate prin asocierea unui indice si se depun în locații succesive de memorie.

Avantaje: timpul de acces este acelasi la oricare element din listă, accesul se realizează prin intermediul indicilor iar implementarea este simplă.

Dezavantaje: lungimea fixă a listei (este obligatorie declararea ei), introducerea si stergerea unui element în/din interiorul listei implică deplasarea tuturor elementelor pe alte poziții decât cele inițiale.

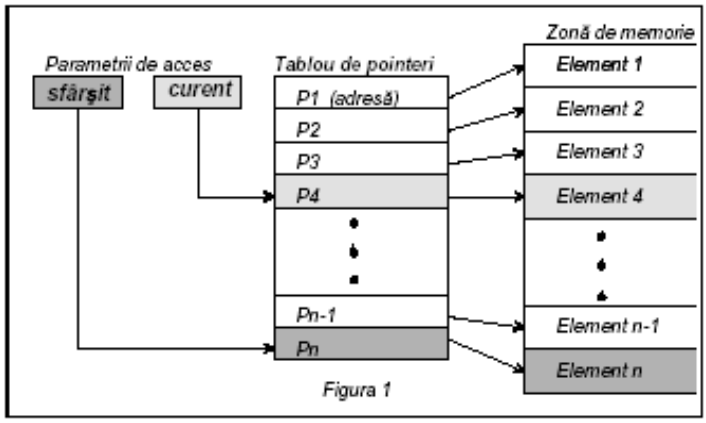

Figura 1

În cazul unei alocări statice (mai eficiente) a unei liste (fig.1), se construieste un tablou de pointeri care permit accesul la informația utilă (elementele listei). Accesul la elemente se realizează prin doi parametri: *curent* (indică poziț ia în tablou a adresei elementului curent) si *sfârsit* (indică poziția în tablou a adresei ultimului element).

În cazul **reprezentării dinamice,** prin liste înlănțuite, elementele listei pot fi dispersate în toată memoria disponibilă (se utilizează eficient zonele libere) iar conectarea elementelor listei se realizează prin pointeri care se adaugă informației utile din fiecare element.

Avantajele reprezentării dinamice sunt: lungimea variabilă a listei (pe parcursul programului de implementare, modificare), introducerea si stergerea fără modificarea poziției pentru celelalte elemente.

Principalul dezavantaj este durata de acces la un element care depinde de poziția sa în listă.

O structură dinamică de listă conține patru parametri de acces (fig. 2):

- lungimea listei (*lungime*), de tip întreg, reprezintă numărul de elemente;
- adresa primului element (*început*), de tip pointer;

- adresa elementului curent (*curent*), de tip pointer;
- adresa ultimului element (*sfârsit*), de tip ponter.

Pentru ca operațiile de inserare si stergere să se facă la fel si pentru elementele de la capetele listei, se pot folosi încă două elemente false (santinele) plasate la început si la sfârsit. Astfel, toate elementele utile din listă au atât predecesor cât si succesor.

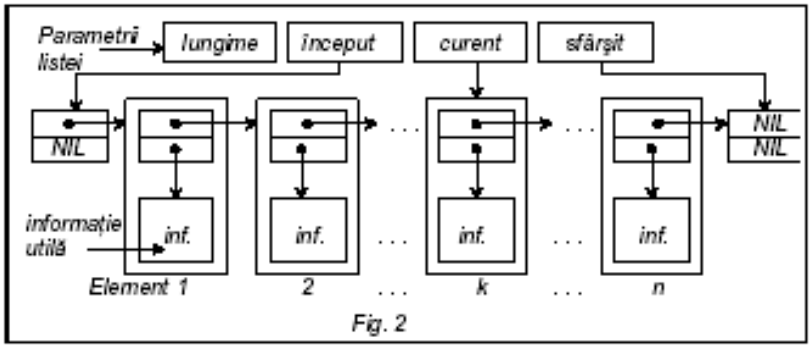

Fig. 2

Fiecare element al listei conține informația propriu-zisă si doi pointeri: unul reprezintă adresa elementului următor - pointer de legătură iar celălalt reprezintă adresa de localizare a informației (nume de persoană si date personale, linia de tabel pentru un candidat, datele de identificare ale unei cărți în bibliotecă etc.)

Pentru introducerea (inserarea) unui element nou în listă, se modifică valoarea pointerului de legătură din elementul predecesor si se copiază vechea valoare a acestui pointer (adresa elementului următor) în pointerul de legătură al noului element .

Căutarea unui element se face prin parcurgerea secvențială a listei, până când pointerul curent ia valoarea dorită sau o componentă a informației corespunde unei condiții.

Lista din fig.2 este o listă simplu înlănțuită - parcurgerea listei se poate face într-un singur sens, dat de pointerii de legătură.

Pentru parcurgere în ambele sensuri, se introduc câte doi pointeri de legătură în fiecare element (câte unul pentru fiecare sens de parcurgere). Se obține astfel o listă dublu înlănțuită (fig.3), care prezintă avantaje suplimentare privind operațiile curente prin micsorarea timpului de acces la un element al listei când elementul curent de acces este anterior sau posterior.

Prin legarea între ele a elementelor de la capetele unei liste simplu sau dublu înlănțuite se conferă listei o structură circulară si se pot elimina elementele "false" de marcare a capetelor (santinele).

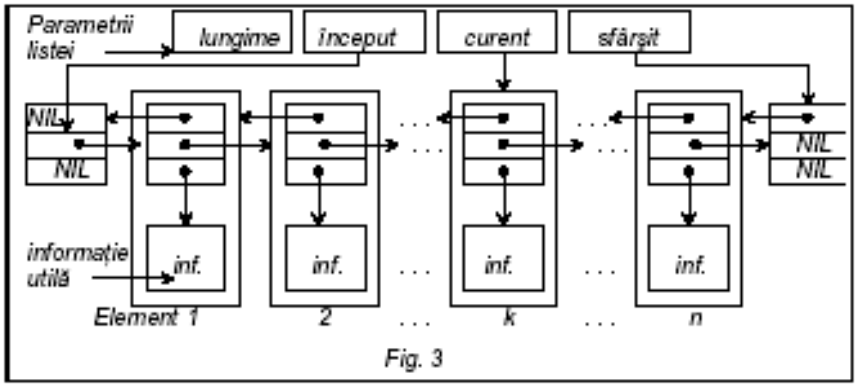

Fig. 3

3. Implementarea operațiilor de bază pentru liste simplu înlănțuite

Presupunem că s-a definit un tip de date DATA, care este specific informației dintr-un element al listei si depinde de aplicație (DATA poate fi un tip simplu sau tip structurat - tablou, structură etc.)

Definim structura C pentru un element al listei, astfel:

```
typedef struct elem
{
DATA data;
struct elem *next;
}
ELEMENT, *LINK ;
```

Tipul ELEMENT corespunde unui element al listei iar tipul LINK unui pointer la un element al listei. Lista este specificată printr-o variabilă de tip LINK care indică primul element al listei. O listă vidă se va specifica printr-un pointer NULL. Câmpul next al ultimului element al listei va conține NULL, indicând că nu mai urmează nici un element.

Se observă că o listă simplu înlănțuită este o structură ordonată, care poate fi parcursă direct de la primul element către ultimul (într-un singur sens). Parcurgerea în ambele sensuri nu se poate face direct, dar se poate realiza prin folosirea unei structuri adiționale de tip stivă. Totusi, această variantă nu este prea des folosită. Când sunt necesare parcurgeri în ambele sensuri, se folosesc liste dublu înlănțuite.

Pentru semnalarea situațiilor de eroare se va folosi următoarea funcție, care tipăreste sirul primit si apoi forțează oprirea programului:

```
void err_exit(const char *s)
{
fprintf(stderr, "\n %s \n", s);
exit(1);
}
```

O primă operație care trebuie implementată este cea de creare a unui element printr-o funcție **new_el()**. Spațiul pentru elementele listei se creează la execuție, prin funcția standard **malloc()**. Este avantajos ca funcția de creare element să inițializeze câmpurile **data** si **next** ale elementului creat. Această funcție va întoarce un pointer la elementul creat.

```
typedef int DATA;
struct el {DATA data; struct el *next; };
typedef struct el ELEMENT, *LINK;
LINK new_el(DATA x, LINK p)
{
LINK t = (LINK) malloc(sizeof(ELEMENT));
if (t==NULL)
err_exit("new_el: Eroare de alocare ");
t->data = x;
t->next = p;
```

```
return t;
}
```

Se observă testarea corectitudinii alocării cu **malloc()**, ca si conversia la tipul LINK si folosirea lui **sizeof()** pentru a obține dimensiunea unui element. Se observă de asemenea că tipul DATA este implementat ca un tip simplu sau ca o structură (ceea ce este recomandat), atunci se poate face atribuirea **t->data = x**, fără probleme.

Iată un prim exemplu de creare a unei liste, pornind de la o structură înrudită, anume de la cea de tablou. Funcția primeşte adresa tabloului şi numărul de elemente, întorcând un pointer la lista creată.

```
LINK arr_to_list_i (DATA *tab, int n)
{
LINK t = NULL, p;
if (n>0) t=p=new_el(*tab++, NULL);
else return NULL;
while (--n)
{
p->next = new_el(*tab++, NULL);
p = p->next;
}
return t ;
}
```

Dacă **n** nu este zero, se crează un prim element si se memorează în **t** si **p**, apoi se crează succesiv elemente, incrementând pointerul **tab** si avansând pointerul **p** la următorul element. În final se întoarce **t**.

O a doua variantă, mai clară, **arr_to_list_r()**, foloseste recursivitatea, testându-se cazul de bază (**n==0**), în care se întoarce **NULL**, altfel se întoarce un pointer la un element creat cu **new_el()**, cu parametrii ***tab** (elementul curent al tabloului) si un pointer întors de aceeasi funcție **arr_to_list_r()**, apelată cu parametrii **tab+1** si **--n**. Ar fi o greseală ca în loc de **tab+1** să se scrie **++tab**, deoarece codul ar depinde de ordinea de evaluare (care nu este garantată !), iar **tab** apare în ambii parametri.

În particular, în Borland C, funcția ar fi incorectă (sare peste primul element).

```
LINK arr_to_list_r(DATA *tab, int n)
{
if (n==0) return NULL ;
else return new_el(*tab, arr_to_list_r(tab+1,--n));
}
```

4. Tipărirea unei liste

Se utilizează o funcție recursivă pentru tipărire, care nu este dependentă de tipul DATA (acest tip depinde de aplicație). Funcția tipăreste câmpul data dintr-un element.

```
void print_data(DATA x){
printf("%6d",x);
}
void print_list(LINK t)
```

```
{
if (t==NULL) printf("NULL \n");
else {
print_data(t->data);
pritf("->");
print_list(t->next);}
}
```

Pentru testarea funcției de tipărire se poate utiliza programul:

```
void main(void){
DATA a[]={00, 11, 22, 33, 44, 55, 66, 77, 88, 99};
print_list(arr_to_list_i(a, 10);
print_list(arr_to_list_r(a, 10); }
```

Formele generale ale funcțiilor de parcurgere a listelor liniare în variantă iterativă si recursivă sunt:

```
void parc_list_r(LINK t)
{
if(t==NULL) {
printf("Lista este vidã ! \n");
return; }
else {
prelucrare(t->data);
parc_list_r(t->next);
}
}
void parc_list_i(LINK t)
{
if(t==NULL) {
printf("Lista este vidã ! \n");
return; }
while(t!=NULL) {
prelucrare(t->data);
t=t->next;
}
}
```

S-a considerat o funcție **`prelucrare(),`** care realizează operațiile dorite asupra datelor din fiecare element; funcția depinde de tipul concret al datelor si deci este dependentă de aplicație.

TEMA:

1. Scrieți un program care citește *n* numere de la tastatura, crează o lista care să conțină aceste valori si apoi o parcurge, afișând doar valorile pare.