Lucrarea de laborator nr. 1

Tema: Reprezentarea tipurilor de date ale limbajului C++ în memoria calculatorului. Operatorii limbajului C++. Construcţiile elementare ale limbajului C++ (instrucţiunile ***for, while, do-while, if-else, switch-break, goto*****).** Tipuri de date recursive, operaţii asupra listelor, arborilor. Construirea şi elaborarea programelor recursive. Fişierele.

Scopul lucrării: familiarizarea studenţilor cu reprezentarea tipurilor de date ale limbajului C++ în memoria calculatorului, operatorii limbajului C++, construcţiile elementare ale limbajului C++ (instrucţiunile ***for, while, do-while, if-else, switch-break, goto*****),** tipuri de date recursive, operaţii asupra listelor, arborilor, construirea şi elaborarea programelor recursive, lucrul cu fişierele.

Consideraţiile teoretice necesare:

Tipurile simple şi structurate ale limbajului C++ .

Identificatorii limbajului C++ sînt formaţi cu ajutorul caracterelor alfanumerice şi caracterul de subliniere "_". Primul caracter al unui identificator nu poate fi o cifră.

Pi //legal
mesaj //legal
maxx //legal
x3 //legal
3x //ILEGAL

În cadrul mulţimii identificatorilor posibili, remarcăm o clasă aparte, reprezentînd cuvintele-cheie. Cele mai frecvente cuvintele-cheie ale limbajul C++sînt

auto	delete	float	interrupt	register	template
break	do	for	long	return	this
case	double	friend	near	short	typedef
char	else	goto	new	signed	union
class	enum	huge	operator	sizeof	unsigned
const	export	if	private	static	virtual
continue	extern	inline	protected	struct	void
default	far	int	public	switch	while

Cuvintele-cheie nu trebuie utilizaţi ca nume de variabile.

Declararea variabilelor trebuie efectuată înainte de a fi folosite, la începutul programului sau chiar în funcţie de contextul problemei în interiorul programului nemijlocit înainte de utilizare, cînd apare necesitarea introducerii variabilei. 0 declaraţie specifică un tip şi este urmată de o listă de una sau mai multe variabile de acel tip, ca în exemplul de mai jos:

```
int i,n;
char c, linie[80];
```

Domeniu de acţiune a variabilelor. Variabilele pot fi iniţializate în momentul declaraţiei lor. Dacă numele este urmat de semnul egal şi de o constantă, aceasta serveşte la iniţializare, ca în următoarele exemple:

```
char backslash = '\\';
int i = 0;
float eps = 1.0e-5;
```

Dacă variabila este *externă* sau *statică*, iniţializarea are loc o singură dată, înainte ca programul să-şi înceapă execuţia. Variabilele *automate*, iniţializate explicit, sînt iniţializate la fiecare apel al funcţiei în care sînt conţinute. Variabilele automate pentru care nu există o iniţializare explicită au valoare nedefinită. Variabilele externe şi statice se iniţializează implicit cu zero, dar este un bun stil de programare acela de a efectua iniţializarea lor în orice caz.

Fiecare variabilă şi constantă posedă un tip, care determină dimensiunea spaţiului necesar memorării lor. Tipurile datelor se pot divide în două categorii: tipuri fundamentale şi tipuri derivate sau structurate.

Tipurile fundamentale ale limbajului C++ sînt

char	reprezentînd tipul *caracter*	pe 1 octet,
int	întreg	pe 2 octeţi
long	întreg	pe 4 octeţi
float,	însemnînd un număr real	pe 4 octeţi
double,	ataşat unui număr real	pe 8 octeţi

Aceste tipuri admit diferite variante, numite tipuri de bază de date.

Tipurile enumerabile sînt introduse prin sintaxa

nume {membrul,membru2, . . .) varl,var2, . . . ;

De exemplu,

enum CULORI {ROŞU , VERDE, ALBASTRU }
culoarea_punct, culoare_linie;
CULORI culoare_cerc, culoare_fond;

defineşte tipul de dată *CULORI* şi declară variabilele *culoarea_punct* şi *culoare_linie* urmate de declarările a încă două variabile *culoare_cerc* şi *culoare_fond* de tipul enumerabil.

Membrii unui tip enumerabil trebuie să fie numai de tip întreg. Valoarea fiecăruia este obţinută prin incrementarea cu 1 a valorii membrului anterior, primul membru avînd, implicit, valoarea 0. Iniţializarea unui membru cu o valoare oarecare, avîndu-se în vedere că doi membri ai aceluiaşi tip nu pot avea aceeaşi valoare. Valorile membrilor următori se stabilesc conform regulilor menţionate. Exemple de tipuri de date enumerabile:

enum ANOTIMP (IARNA=1, PRIMĂVARA, VARA, TOAMNA) ;
enum BOOLEAN {fals, adevărat) condiţie;
enum DIRECŢIE {SUS, JOS, DREAPTA, STÎNGA=5};

Putem defini tipuri enumerabile fără a specifica numele acestora. Procedînd astfel, putem grupa un set de constante fără a denumi acea mulţime, de exemplu,

enum {bine, foarte_bine, cel_mai_bine};
Utilizarea variabilelor de tip enumerabil. Limbajul C++ permite atribuiri de tipul
condiţie=0;
while (! condiţie)
{ cout <<"Utilizarea variabilei enumerabile";
condiţie=true; }

Este bine ca astfel de atribuiri să fie însoţiţi de conversia de tip corespunzătoare.

condiţie=false;
condiţie=(enum BOOLEAN) 0;

Enumerările, definite în interiorul structurilor limbajului C++, nu sînt vizibile în afara acestora.

Tipurile structurate sînt obţinute de la tipurile de bază. Tipurile derivate acceptate de limbajul C++ sînt: pointeri, referinţe, tablouri, structuri, uniuni şi clase.

Pointerul este o variabilă care conţine adresa unei alte variabile de orice tip. Pentru a defini un pointer, vom specifica tipul datei a cărei adresă urmează să o memoreze.

*int *ip;* // pointer către un întreg

*char **s*; // pointer la un pointer pe caractere.

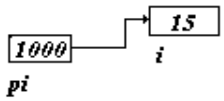
Să considerăm o variabilă de tip *int i* şi un pointer *pi* către un întreg.

int i=15, j;

*int *pi=NULL;*

pi=&i;

**pi=20;* // i=20;



Deoarece operatorul adresă & furnizează adresa unei variabile, instrucţiunea *pi=&i* asignează variabilei *pi* adresa lui *i*.(de exemplu, adresa 1000). Un alt operator unar ce însoţeşte clasa pointerilor este * care furnizează conţinutul locaţiei de memorie de pe adresa indicată de către pointer, de exemplu,

**pi=i;* // adică 15;

Dacă *j* este un alt *int*, atunci *j=*pi* asignează lui *j* conţinutul locaţiei indicate de *pi*. Are loc următoarea echivalenţă: *j=*pi; adică j=i;*

Pointerii pot apărea în expresii. De exemplu, dacă *pi* conţine adresa lui *i*, atunci **pi* poate apărea în orice context în care ar putea apărea *i*, cum ar fi

*j=*pi+l;* // adică *j=i+l;*

*printf("%d\n",*pi);*

*d=sqrt((double)*pi);*

În expresii ca *j=*pi+1*; operatorii unari * şi & sînt prioritari faţă de cei aritmetici, astfel, această expresie adună 1 şi asignează valoarea obţinută lui j ori de cîte ori pointerul pi avansează.

Referiri prin pointeri pot apărea şi în membrul stîng al atribuirilor. Dacă *pi* conţine adresa lui *i*, atunci **pi=0* îl pune pe *i* ca *0*, iar **pi+=1* îl incrementează pe *i*, ca şi *(*pi)++*. În ultimul exemplu parantezele sînt necesare, fără ele se incrementează *pi* în loc
să incrementeze ceea ce indică *pi*, deoarece operatorii unari * şi ++ sînt evaluaţi de la dreapta spre stînga. De exemplu,

```
void main()
{ int *pi, i=10;
float *pf, f=12.5;
double *pd, d=0.001;
char *pc, c='a';
*pi=i; *pf=f; *pd=d; *pc=c;
printf("pi=%p, pf=%p, pd=%p, pc=%p", pi, pf, pd, pc);
printf("*pi=%i, *pf=%f, *pd=%e, *pc=%c", *pi, *pf, *pd, *pc);
printf("pi++ =%p, pf++ =%p, pd++ =%p, pc++=%p", pi++, pf++, pd++, pc++);
printf("(*pi)++ =%p, (*pf)++ =%p, (*pd)++ =%p, (*pc)++ = %p", (*pi)++ , (*pf)+
        +, (*pd)++, (*pc)++);
}
```

Rezultatul îndeplinirii programului:

*pi=8F1C, pf=0758, pd=074C, pc=1330*pi=10, *pf=12.500000, *pd=1.000000e-03, *pc=a pi+ + =8F1C, pf++ =0758,*

*pd++ =074C, pc++=1330(*pi)++ =1B57, (*pf)++ =0000,*

*(*pd)++* =0000, (**pc)++* = 0000

Deoarece pointerii sînt variabile, ei pot fi manevraţi ca orice altă variabilă. Dacă *pj* este un alt pointer la *int*, atunci

pj=pi;

copiază conţinutul lui *pi* în *pj*, adică pj va indica la variabila adresa căreia este indicată în *pi*, astfel că *pj* se va modifica odată cu *pi*. Pointerii pot conţine adrese către elemente fără tip, cu void. Putem atribui unui pointer void valoarea unui pointer non-void fără a fi necesară o operaţie de conversie de tip *typecast.*

```
char *cp;       // pointer către un caracter
void *vp;       // pointer către void
vp=cp;          // legal - pointerul la caracter depus în
                //pointerul către void
cp=vp;          // ILEGAL - lipseşte conversia de tip
```

cp=(char) vp*; // legal - pointerul către *void* depus în pointerul către caracter cu conversie de tip.

Referinţa prezintă o legătură cu o variabilă, conţinînd o adresă. Spre deosebire de pointeri, în a căror declarare se utilizează simbolul "*", pentru a defini o referinţă, vom folosi simbolul "&".

```
int i;          // declararea unui întreg
int *p=&i;      // definirea unui pointer la i
int &r=i;       // definirea unei referinţe la i
```

Atît *p*, cît şi *r* acţionează asupra lui i.

```
i=55;           // acţiune asupra lui i
*p=13;          // acţiune asupra lui i
r=20;           // acţiune asupra lui i.
```

Există însă o diferenţă majoră între *p* şi *r*, nu numai în modul de apelare, ci şi datorită faptului că *p* poate, la un moment dat, să fie în legătură cu o altă variabilă, a cărei locaţie de memorie o va conţine, diferită de cea a lui *i*, în timp ce *r* nu-şi poate schimba referinţa, acesta nefiind altceva decît o redenumire a variabilei *i*. În ceea ce priveşte utilizarea referinţelor, va trebui să ţinem cont de următoarele restricţii:

– referinţele trebuie iniţializate chiar în momentul declarării lor,
– odată fiind iniţializate, referinţelor nu li se pot schimba locaţiile la care se referă,
– nu sînt permise referinţe la referinţe şi pointeri către referinţe, dar putem avea o referinţă la un pointer.

Referinţele pot fi utilizate drept constante, pot fi iniţializate cu constante, funcţii sau chiar structuri.

Tablourile, din rîndul cărora fac parte vectorii şi matricele, sînt tipuri de date foarte apropiate pointerilor şi referinţelor. Vom vedea că orice operaţie, care poate fi rezolvată prin indexarea tablourilor, poate fi rezolvată şi cu ajutorul pointerilor. Astfel, declaraţia

char linie[80];

defineşte *linie* ca fiind un şir de 80 de caractere şi, în acelaşi timp, *linie* va constitui un pointer la caracter. Dacă *pc* este un pointer la un caracter, declarat prin

*char *pc;*

atunci atribuirea *pc=&1inie[0];* face ca *pc* să indice primul element al tabloului *linie* (de indice zero). Aceasta înseamnă că *pc* conţine adresa lui *linie[0]*. Acum atribuirea

*c=*pc;*

va copia conţinutul lui *linie[0]* în *c*. Dacă *pc* indică un element al lui *linie*, atunci, prin definiţie, *pc+1* indică elementul următor şi, în general, *pc-i* indică cu *i* elemente înaintea elementului indicat de *pc*, iar *pc+i* cu *i* elemente după acelaşi element. Dacă *pc* indică elementul *linie[0]*, **(pc+1)* indică conţinutul lui *linie[1]*, *pc+i* este adresa lui *linie[i]*, iar **(pc+i)* este conţinutul lui *linie[i]*. Observăm că operatorul de indexare [], de forma *E1[E2]*, este identic cu**((E1)+(E2))*. Aceste remarci sînt adevărate indiferent de tipul variabilelor din tabloul *linie*. Definiţia adunării unităţii la un pointer şi, prin extensie, toată aritmetica pointerilor constă, de fapt, în calcularea dimensiunii memoriei ocupate de obiectul indicat. Astfel, în *pc+i i* este înmulţit cu lungimea obiectelor pe care le indică *pc*, înainte de a fi adunat la *pc*.

Corespondenţa între indexare şi aritmetica pointerilor este foarte strînsă. Referinţa la un tablou este convertită de către compilator într-un pointer spre începutul tabloului. Numele acestui tablou este o expresie de tip pointer.

Evaluînd elementul *linie[i]*, limbajul C++ îl converteşte în **(linie+i)*, cele două forme fiind echivalente. Aplicînd operatorul & ambilor termeni ai acestei echivalenţe, rezultă că *linie[i]* este identic cu *linie+i*, unde *linie+i* fiind adresa elementului *i* din tabloul *linie*. Dacă *pc* este un pointer, el poate fi utilizat în expresii cu un indice *pc[i]* fiind identic cu **(pc+i)*. Un pointer este o variabilă. Deci,

pc=linie; // şi
pc++;

sînt operaţii permise. Numele unui tablou este o constantă şi nu o variabilă, construcţii de tipul *linie++* fiind interzise. Singurele operaţii permise a fi efectuate asupra numelor tablourilor, în afara celor de indexare, sînt cele care pot acţiona asupra constantelor

Aritmetica adreselor pentru pointeri şi tablouri constituie unul din punctele forte ale limbajului C++. Se garantează că nici un pointer care conţine adresa unei date nu va conţine valoarea zero, valoare rezervată semnalelor de eveniment anormal. Această valoare este atribuită constantei simbolice *NULL* pentru a indica mai clar că aceasta este o valoare specială pentru un pointer. În general, întregii nu pot fi asignaţi pointerilor, zero fiind un caz special.

Există situaţii în care pointerii pot fi separaţi. Dacă *p* şi *q* indică elemente ale aceluiaşi tablou, operatorii <, >, =, etc. lucrează conform regulilor cunoscute. *p<q* este adevărată, de exemplu, în cazul în care p indică un element anterior elementului pe care îl indică q. Relaţiile == şi *!=* sînt şi ele permise. Orice pointer poate fi testat cu *NULL*, dar nu există nici o şansă în a compara pointeri situaţi în tablouri diferite.

Este valabilă şi operaţia de scădere a pointerilor. Astfel, dacă *p* şi *q* indică elementele aceluiaşi tablou, *p-q* este numărul de elemente dintre p şi q. O funcţie, deosebit de utilă în lucrul cu şiruri de caractere, este *strlen()*, care returnează lungimea şirului de caractere transmis ca parametru.

```
int strlen(char *s)
{ char *p=s;
while (*p!='\0') p++;
return p-s; }
```

Prin declarare *p* este iniţializat cu *s* şi indică primul caracter din *s*. În cadrul ciclului *while* este examinat conţinutul şirului de caractere, indirect, prin intermediul pointerului *p*, caracter după caracter, pînă cînd se întîlneşte *'\0"*, acesta din urmă semnificînd sfîrşitul şirului. Dacă *while* ar testa doar dacă expresia este zero, este posibilă omiterea testului explicit, astfel de cicluri fiind deseori scrise sub forma

```
while (*p) p++;
```

Deoarece *p* indică şirul de caractere, *p++* face ca *p* să avanseze de fiecare dată la caracterul următor, iar *p-s* dă numărul de caractere parcurse (lungimea şirului). Aritmetica pointerilor este consistentă: dacă am fi lucrat cu *float*, care ocupă mai multă memorie decît *char*, şi dacă *p* ar fi fost un pointer la *float*, *p++* ar fi avansat la următorul *float*. Toate manipulările de pointeri iau automat în considerare lungimea obiectului referit.

Să implementăm, de exemplu, o funcţie de comparare a două şiruri de caractere. Funcţia *strcmp(s, t)* compară şirurile de caractere *s* şi *t* şi returnează valoare negativă, nulă sau pozitivă, în funcţie de relaţia dintre *s* şi *t* (care poate fi *s<t*, *s=t* sau *s>t*). Valoarea returnată este obţinută prin scăderea caracterului de pe prima poziţie, unde *s* diferă de *t*. Pentru claritatea problemei, vom prezenta două variante, una utilizînd tablourile, iar cea de a doua utilizînd pointerii.

Varianta cu tablourile:

```
strcmp(char s[], char t[])
{ int i=0;
while (s[i]==t[i])
 if (s[i++]=='\0') return 0;
return s[i]-t[i];}
```

Varianta cu pointerii:

```
strcmp(char *s, char *t)
{ for(;*s==*t;s++,t++)
if(*s=='\0') return(0);
return (*s-*t);}
```

Dacă ++ şi - - sînt folosiţi ca operatori prefixaţi, pot apărea alte combinaţii de *, ++ şi - -, deşi mai puţin frecvente. De exemplu: *++*p* incrementează pe *p* înainte de a aduce caracterul spre care indică *p*. *- - *p* decrementează pe *p* în aceleaşi condiţii.

Alte operaţii, în afara celor menţionate deja (adunarea sau scăderea unui pointer cu întreg, scăderea sau compararea a doi pointeri), sînt ilegale. Nu este permisă adunarea, împărţirea, deplasarea logică sau adunarea unui *float* sau *double* la pointer.

Tablourile multidimensionale pot fi definite cu ajutorul tablourilor de tablouri, de exemplu.:

```
char ecran [25][80];
```

excepţie făcînd tablourile de referinţe, acestea din urmă nefiind permise datorită faptului că nu sînt permişi pointerii la referinţe.

Dacă *E* este un tablou *n-dimensional* de dimensiuni *i, j, ..., k*, atunci apariţiile lui *E* în expresii sînt convertite în pointer la un tablou *n-1*-dimensional de dimensiuni *j, ..., k*. Dacă la acesta se aplică explicit sau implicit (prin indexare) operatorul *, rezultatul este tabloul *n-1*-dimensional indicat de pointer, care, la rîndul său, este convertit imediat în pointer.

Tablourile sînt memorate pe linii şi, deci, ultimii indici, de la stînga la dreapta, variază mai repede decît primii. Prima dimensiune a unui tablou se foloseşte numai pentru a determina spaţiul ocupat de acesta, ea nefiind luată în consideraţie decît la determinarea unui element de indici daţi. Este permisă omiterea primei dimensiuni a unui tablou, dacă tabloul este *extern*, alocarea făcîndu-se în cadrul altui modul sau cînd se efectuează iniţializarea tabloului în declaraţie, în acest caz fiind determinată dimensiunea din numărul de elemente iniţializate.

Iniţializarea tablourilor poate avea loc chiar în cadrul declarării acestora

```
int point[2]={10,19};
char mesaj1[6]={'S[1]','a','l','u','t','\0'};
char mesaj2[6]="Salut";
```

Observăm că şirurile de caractere se comportă oarecum ciudat. Atît *mesaj1*, cît şi *mesaj2* sînt şiruri de 6 caractere avînd drept terminator de şir caracterul nul. Diferenţa între cele două şiruri nu se află în conţinutul lor, ci în cadrul iniţializării lor. În cazul iniţializării prin acolade, { }, caracterul nul nu este subînţeles, prezenţa acestuia rămînînd la latitudinea noastră, în schimb, adoptînd o iniţializare prin ghilimele, " ", va trebui să dimensionăm corespunzător şirului de caractere, ţinînd cont de prezenţa terminatorului de şir. În exemplul *mesaj2* avem 5 litere plus caracterul nul, fiind necesare 6 locaţii în vederea memorării cuvîntului "*Salut*".

Dimensionarea tablourilor se realizează în concordanţă cu necesităţile aplicaţiei. Există posibilitatea iniţializărilor parţiale, care nu utilizează întreg spaţiu rezervat. În cazul şirurilor de caractere, restul spaţiului rămas neutilizat va conţine numai caracterul nul. În restul situaţiilor, conţinutul tabloului fiind aleator, se recomandă iniţializarea acestuia în cadrul unui ciclu.

Tablouri de pointeri. Pointerii sînt ei înşişi variabile, de aceea ei sînt utilizaţi în tablouri de pointeri. Pentru exemplificare, vom considera un program care sortează un set de linii de text în ordine alfabetică. Cu toate că algoritmul de sortare este unul comun, deosebirea dintre sortarea unui tablou de numere şi a unuia de şiruri de caractere constă în aceea că liniile de text de lungimi diferite nu pot fi comparate sau deplasate printr-o singură operaţie. Avem nevoie de o reprezentare a datelor care să se poată face eficient şi potrivit regulilor de gestionare a liniilor de text de lungimi diferite.

Introducem noţiunea de *tablou de pointeri*. Dacă liniile de sortare sînt memorate cap la cap într-un şir de caractere, atunci fiecare linie poate fi accesată printr-un pointer la primul său caracter. Pointerii înşişi pot fi memoraţi într-un tablou. Două linii pot fi comparate prin transmiterea pointerilor respectivi lui *strcmp()*. Cînd două linii neordonate trebuie inversate, se inversează pointerii lor în tabelul de pointeri, nu însăşi liniile. Acest mod de lucru elimină cuplul de probleme legate de gestionarea memoriei şi poate deplasa liniile.

Procesul de sortare constă din trei etape:

- citirea tuturor liniilor la intrare,
- sortarea liniilor,
- tipărirea liniilor în ordine.

Împărţim programul în funcţii care efectuează aceste trei etape. Funcţia de intrare trebuie să colecteze şi să salveze caracterele din fiecare linie şi să construiască un tablou de pointeri pe linii. Va trebui să numere liniile la intrare. Această informaţie este necesară pentru sortare şi tipărire. Deoarece funcţia de intrare poate opera doar cu un număr finit de linii, ea va returna o valoare, cum ar fi -1, în cazul în care se vor prezenta mai multe linii. Funcţia de ieşire trebuie doar să tipărească liniile în ordinea în care apar în tabloul de pointeri.

```
#include <stdio.h>
#include <string.h>
#include <conio.h>
#define LINII 100
#define MAXLEN 1000
int citeste_linii(char *s[])
{   printf("Introdu un text (maxlen=1000)\n");
    printf("Pentru introducerea unei linii noi se utilizează tasta       ENTER.\n");
    printf("Sfîrşitul textului se va marca prin '*'. \n\n");
  char c;
   int i=0, j=0;
   while((c=getchar())!='*')
```

```
   {   if (c=='\n')
            {  s[i][j]='\0';  i++;j=0;   }
     else s[i][j++]=c;
    }
  return i+1;   }
void scrie_linii (char *linieptr[],int maxlinii)
{  for( int i=0; i<maxlinii;i++)
     printf("%s\n", linieptr[i]);   }
void sortare_linii(char *v[],int n)
{    char *temp;
  for(int k=n/2;k>0;k/=2)
    for(int i=k;i<n;i++)
      for(int j=i-k;j>=0;j-=k)
           { if (strcmp(v[j],v[j+k])<=0) break;
             temp=v[j];
             v[j]=v[j+k];
             v[j+k]=temp; }  }
void main()
{  clrscr();
  char *linieptr[LINII];
  int nlinii;
  if ((nlinii=citeste_linii(linieptr))>=0)
     {  printf("\n Textul pînă la sortare:\n");
     scrie_linii(linieptr,nlinii);
     sortare_linii(linieptr,nlinii);
     printf("\n Textul după sortare \n");
      scrie_linii(linieptr,nlinii);
     }
  else printf("Input prea mare pentru sortare \n");  }
```

Rezultatul îndeplinirii programului:

```
Introdu un text (maxlen=1000)
Pentru introducerea unei linii noi se utilizează tasta       ENTER.
Sfîrsitul textului se va marca prin '*'.
ashdgasghddgjahsdgjaghdjhagsdhgdhjgdh*
 Textul pînă la sortare:
ashdgasghddgjahsdgjaghdjhagsdhgdhjgdh va marca prin '*'.
 Textul după sortare
ashdgasghddgjahsdgjaghdjhagsdhgdhjgdh va marca prin '*'.
```

Declararea variabilei *linieptr:*

```
char *linieptr[LINII];
```

arată că *linieptr* este un tablou de *LINII* elemente, fiecare element fiind un pointer la *char*. *linieptr[i]* este un pointer la caractere, iar **linieptr[i]* accesează un caracter. Dacă *linieptr* este el însuşi un tablou care este transmis lui *scrie_linii()*, el poate fi tratat ca un pointer, iar funcţia poate fi scrisă.

```
void scrie_linii(char *linieptr[],int nlinii)
 {   while (- - nlinii>=0)
```

*printf("%s\n",*linieptr++); }*

**linieptr* adresează iniţial prima linie, iar, cu fiecare incrementare, el avansează linia următoare pînă cînd *nlinii* se epuizează.

Sortarea are loc în cadrul funcţiei *sortare_linii()*. Dacă orice element individual din *v* este un pointer la caractere, *temp* va fi un astfel de pointer, încît cei doi pot fi copiaţi unul în altul.

Fiind date declaraţiile

*int a[10][10]; int *b[10];*

utilizările lui *a* şi *b* pot fi similare, în sensul că *a[5][5]* şi *b[5][5]* sînt referinţe legale ale aceluiaşi *int*. Toate cele 100 celule de memorie ale tabloului *a* trebuie alocate, iar găsirea fiecărui element se face prin calculul obişnuit al indicelui. Pentru tabloul *b*, prin declararea sa, se alocă 10 pointeri, fiecare dintre aceştia urmînd să indice un tablou de întregi. Presupunînd că fiecare indică la 10 elemente din tablou, vom obţine 100 celule de memorie rezervate, plus cele 10 celule pentru pointeri. Astfel, tabloul de pointeri utilizează mai mult spaţiu şi poate cere un mod explicit de iniţializare. Dar există două avantaje: accesarea unui element se face indirect prin intermediul unui pointer, în loc să se facă prin înmulţire şi adunare (cum este în cazul tabloului multidimensional), iar liniile tabloului pot fi de lungimi diferite. Aceasta înseamnă că nu orice element al lui *b* este constrîns să indice la un vector de 10 elemente, unii pot indica la cîte 2 elemente, alţii la cîte 20 de elemente sau chiar la nici unul.

Structura este o colecţie de elemente de tipuri diferite şi care pot fi referiţi atît separat, cît şi împreună. Definirea unei structuri se realizează cu ajutorul cuvîntului-cheie *struct***.** Ea are următoarea sintaxă:

struct [tip structură] { tip_1 element_1;
….…..…
tip_ n element_n
} obiect_ de_ tip_ structură;

unde *tip structură* descrie organizarea structurii,

tip1,..., tipn indică tipul elementelor structurii,

element1, ..., element sunt numele elementelor structurii,

obiect_ de_ tip_ structură este una sau o listă de variabile pentru care se alocă memorie. De exemplu,

struct punct { float x,y;} p;

S-a definit o structură *p* ca fiind de tip *punct***,** punctul fiind compus din două elemente *x* şi *y* reale. Asupra componentelor unei structuri putem acţiona prin intermediul operatorului de apartenenţă, ".", de exemplu:

p.x=10; p.y=30;

Există posibilitatea efectuării operaţiilor cu întreaga structură, atribuirea fiind, de exemplu :

p={10,30);

0 declaraţie de structură care nu este urmată de o listă de variabile nu produce alocarea memoriei, ci descrie organizarea structurii., de exemplu:

typedef struct { char name[25]; int id, age;
char prp; } student;

Definirea unei structuri permite determinarea unui nou tip de date. În continuare definind pointeri la această structură, tablouri, ale căror elemente sînt de tipul acestei structuri, şi elemente de acest tip pot fi definite noi structuri de date mai compuse.

Un alt aspect al utilităţii structurilor îl constituie tratarea tablourilor de structuri, de exemplu:

punct hexagon[6]; punct octogon[8];

Accesul către membrii componenţi ai fiecărui element al vectorului se realizează prin combinarea accesului indexat, caracteristic tablourilor, cu cel utilizat în cazul structurilor:

hexagon[i].x=10;

În cazul definirii unui pointer la o structură, accesul la componentele acelei structuri se va efectua prin expresii de forma

*punct *pptr;*

pptr->x=10; // Echivalent cu *p.x=10;*

*(*pptr) .y=30;* // Echivalent cu *p.y=30*;

Parantezele au rolul de a indica ordinea în care acţionează cei doi operatori "*" şi ".", prioritar fiind "*".

Unele elemente ale unei structuri pot fi *cîmpuri* de biţi. Un *cîmp* de biţi este o configuraţie de biţi adiacenţi, ce apar într-un element de tip ***int*.** Cîmpurile sînt declarate de tip *unsigned,* iar numele cîmpului este urmat de două puncte ":" şi un număr ce reprezintă numărul de biţi ocupaţi de cîmpul respectiv:

unsigned nume_cîmp:nr_biţi;

Cîmpurile pot fi accesate ca orice alt element de structură. Orice cîmp trebuie să aibă toţi biţii în interiorul unei zone de tip *int* (nu poate avea biţi în două cuvinte diferite). Ordinea de alocare a memoriei pentru cîmpuri este dependentă de sistem. Unele sisteme fac alocarea de la stînga la dreapta, iar altele invers. Nu se pot utiliza tablouri de cîmpuri. Cîmpurile nu au adresă şi nu li se poate aplica operatorul de adresare *&***.**

Un caz special de structuri îl constituie ***union*.** Acestea sînt structuri alternative pentru care dimensiunea spaţiului necesar memorării lor este egală cu cea mai mare dimensiune necesară memorării unei componente a acelei structuri. De exemplu, *variabila* este de tipul

union un_tip

{ int uint; float ufloat; char uchar;

punct upunct; // upunct este de tipul structurii *punct*

} variabila;

Toate componentele uniunii ocupă aceeaşi zonă în cadrul memoriei. Spre deosebire de structuri, în uniune este accesibilă o singură componentă a unei uniuni. Uniunea se defineşte în aceeaşi manieră ca şi structurile, cuvîntul-cheie utilizat fiind *union.*

Operatorii

Operatori şi expresii. Acţiunile desfăşurate în cadrul oricărui program, în marea lor majoritate, se efectuează prin expresiile formate prin combinaţii de date şi operatori. Limbajul C++ posedă toţi operatorii limbajului C şi completează această listă cu operatori proprii. Din lista operatorilor disponibili ai limbajului C++ indicăm operatorii caracteristici lui *new* – pentru alocarea memoriei, *delete* – pentru eliberarea memoriei alocate cu operatorul *new*, :: – operatorul de scop sau de rezoliţie.

În funcţie de numărul de operanzi, operatorii se pot clasifica în trei categorii: operatori unari, binari şi ternari.

Operatori unari. Formarea expresiilor în care intervin operatorii unari se produce de la dreapta la stînga.

Operatorul de indirectare*:* * se poate aplica unei expresii de tip pointer *(*expresie)* şi are drept rezultat o valoare (*lvalue* sau adresă) care se referă la obiectul indicat de pointer.

Operatorul de adresare*:* & poate fi aplicat unei valori *(&lvalue)* şi are ca rezultat un pointer la obiectul definit de *lvalue* şi are acelaşi tip ca şi tipul *lvalue*.

***Operatorul unar minus**:* - se aplică unei expresii *(-expresie)* în vederea inversării semnului acesteia.

Operatorul negaţie logică: ! se poate aplica unei expresii aritmetice sau unui pointer *(! expresie)* şi are ca rezultat 1, dacă valoarea operandului este 0, şi 0, în caz contrar, tipul rezultatului fiind *int.*

***Operatorul negaţie pe biţi**:* ~ se aplică unei expresii de tip întreg *(~expresie)* şi transformă 0 în 1 şi 1 în 0 în toţi biţii rezultaţi după conversiile uzuale.

***Operatorul de incrementare**:* ++ incrementează cu 1 valoarea operandului.

***Operatorul de decrementare**:* - - decrementează cu 1 valoarea operandului.

Operatorul ++, ca şi operatorul - -, poate fi utilizat atît ca prefix, cît şi ca sufix. În cazul utilizării lor ca prefix, întîi se acţionează cu operatorul asupra valorii operandului şi apoi se utilizează noua valoare a acestuia. În cazul utilizării lor ca sufix, întîi se utilizează valoarea acestuia, apoi se acţionează cu operatorul asupra valorii operandului.

***Conversia unei expresii (typecast*)**: este de tipul *(tip) expresie* sau *(expresia)* şi produce conversia valorii expresiei la tipul specificat.

Operatorul dimensiune: *sizeof* este de tipul *sizeof (expresie)* sau *sizeof (tip)* şi ne indică dimensiunea în octeţi a operandului, determinată din declaraţiile elementelor ce apar în expresie.

Operatorul de alocare a memoriei: new apare sub forma

pointer_la_nume = new nume [iniţializator]

şi încearcă să creeze un obiect *nume* prin alocarea unui număr egal cu *sizeof(nume)* de octeţi în memoria heap, adresa acestuia fiind returnată. În cazul în care alocarea nu este efectuată cu succes, se returnează valoarea *NULL*

Operatorul de eliberare a memoriei**: **delete are sintaxă de forma *delete pointer_la_nume*

şi eliberează memoria alocată începînd de la adresa conţinută de *pointer_la_nume*.

***Operatorul virgulă**:* , produce expresii de forma

expresie, expresie;

El efectuează evaluarea expresiilor de la stînga la dreapta şi are ca rezultat şi tip valoarea şi tipul ultimei expresii. Gruparea cu paranteze este permisă şi produce o singură valoare. De exemplu:

```
void main()
{int s;
for(int i=0,s=0;i<10,i++)  s+=I;
cout<< “Suma este de ”<<s<<endl;
}
```

Operatori binari

***Operatorii aritmetici**: +, -, *, /* acţionează respectînd regulile binecunoscute de calculare a expresiilor. Trebuie făcută o observaţie asupra operatorului de împărţire /. În cazul în care ambii operanzi sînt întregi, rezultatul este întreg (prin trunchierea rezultatului real).

***Operatorul modulo**:* % furnizează restul împărţirii primului operand la cel de al doilea. De exemplu, un număr este par, dacă este divizibil cu 2. Deci

```
if (x%2==0) cout « "x este par";
else   cout « "x este impar";
```

***Operatorul de deplasare la stînga**:* « are ca rezultat deplasarea către stînga a valorii operandului stîng cu un număr de biţi egal cu valoarea operandului drept, biţii eliberaţi astfel fiind completaţi cu valoarea 0.

***Operatorul de deplasare la dreapta**:* » acţionează în mod similar cu precedentul, singurul element care diferă faţă de operatorul anterior fiind sensul deplasării. De exemplu, funcţia definită mai jos *Biti* (x,4,3) returnează 3 biţi din poziţiile 4,3 şi 2, aliniaţi la dreapta.

```
Biti (unsigned x,unsigned p,unsigned n)
{return (x»(p+1-n))&~ (~0«n) ;  }
```

***Operatorii de comparaţie**: <, <=, >, >=, = =(egal),* !=*(neegal)* au ca rezultat o valoare de tip *int* care este 0 în cazul în care condiţia nu este îndeplinită şi 1– în caz contrar. Pointerii pot fi comparaţi numai pe aceeaşi structură de date, iar ultimii doi operatori permit compararea pointerului cu *NULL*, care corespunde adresei vide.

***Operatorii logici binari pe biţi**:* & *(şi),* | *(sau),*^ (*sau exclusiv*) furnizează un rezultat de tip *int* (*0* pentru valoarea *false* şi *1* pentru valoarea *true).* De exemplu, funcţia *număr_biti()* controlează numărul de biţi pe 1 dintr-un argument întreg

```
număr_biti (unsigned n)
{ int b;
for (b=0; n!=0; n»=1)
if (n&O1) b++;
return b;  /}
```

***Operatorii logici binari**:* && (*şi),* || *(or*). Pentru ambii operatori se efectuează evaluări de la stînga spre dreapta pînă la prima expresie de valoare 0 (pentru &&) sau, respectiv, nenulă (pentru ||), cînd valoarea întregii expresii devine 0 şi, respectiv, 1.

***Operatorii de atribuire**: op=* unde *op* face parte din mulţimea *{ +, -, *, /, %, «, », &, ^, |}* se grupează de la dreapta la stînga, tipul expresiei de atribuire este tipul operandului stîng, iar rezultatul acţiunii operatorului se regăseşte tot în acest operand. Orice expresie de tipul *x op= y* este echivalentă cu *x =x op y*.

Operatori ternari

***Operatorul condiţional**: (condiţie) ? :* produce expresii de forma *(expr1) ? expr2 : expr3*, în care se evaluează *exp1*. În cazul în care aceasta este nenulă, se returnează valoarea expresiei *expr2*, în caz contrar, se returnează valoarea lui *expr3.* De exemplu, ciclul următor tipăreşte *N* elemente ale unui tablou, 10 pe linie, cu fiecare coloană separată printr-un blanc şi cu fiecare linie (inclusiv ultima) terminată cu un singur caracter ‘\n’– linie nouă:

```
for (i=0;i<N;i++)
printf("%6d %c",a[i],(i%10==9||i==N-1)?'-\n':' ');
```

Acest exemplu poate fi scris prin intermediul instrucţiunii *if* în felul următor:

```
for (i=0;i<N;i++)
if,(i%10==9||i==N-1)
printf("%6d %c",a[i],'-\n');
else printf("%6d %c",a[i],' ');
```

Instrucţiuni

Expresiile sînt utilizate în scrierea instrucţiunilor. O instrucţiune este o expresie care se încheie cu punct şi virgulă “;". Instrucţiunile pot fi scrise pe mai multe linii program, spaţiile nesemnificative fiind ignorate. Pe o linie de program putem scrie multe instrucţiuni. Instrucţiunile pot apărea în diferite forme: de atribuiri, de declaraţii, instrucţiuni condiţionale, de ciclare, de salt, instrucţiuni compuse.

Instrucţiunea compusă (blocul de instrucţiuni) grupează declaraţii şi instrucţiuni în vederea utilizării unui bloc de instrucţiuni echivalent cu o instrucţiune compusă. Forma generală

este:

{lista_declaraţii lista_instrucţiuni}

Instrucţiunea condiţională if, if-else are una din formele*:*

if (expr) instrucţiune;

if (expr) instrucţiune_1; else instrucţiune_2;

Instrucţiunea *if* evaluează expresia *expr*. În cazul în care se obţine o valoare nenulă, se trece la executarea *instrucţiune_1*, iar dacă această valoare este nulă şi există *instrucţiune_2*, se va executa *instrucţiune_2*. În cazul absenţei variantei *else* se va trece la execuţia instrucţiunii imediat următoare instrucţiunii *if*.

Instrucţiunea de ciclu condiţionată anterior while este

while (expr) instrucţiune;

Atît timp cît valoarea expresiei *expr* este nenulă, se execută *instrucţiune*. De exemplu,

```
i=0;
while (i<n)  a[i++]=0.0;
```

Evaluarea expresiei are loc *înaintea* execuţiei instrucţiunii, fapt pentru care această instrucţiune este din clasa ciclurilor cu precondiţie. Din acest motiv este posibil ca corpul ciclului să nu se execute nici măcar o dată, dacă condiţia ciclului este falsă. Execuţia programului va trece la instrucţiunea imediat următoare instrucţiunii de ciclu.

Instrucţiunea de ciclu condiţionată posterior do-while are forma *do instrucţiune while (expr);*

instrucţiune se va executa pînă ce valoarea expresiei *expr* devine falsă. Spre deosebire de instrucţiunea *while*, în ciclul *do-while* evaluarea expresiei are loc *după* fiecare executare a corpului ciclului. Datorită acestui fapt *instrucţiune* se va executa cel puţin o singură dată, iar instrucţiunea *do* se încadrează în categoria ciclurilor cu postcondiţie. De exemplu:

```
i=0;
do
{ a[i++]=0.0; }
while (i<n);
```

Ciclul *do-while* este folosit mai puţin decît ciclul *for*. Cu toate acestea, prezenţa sa se impune în cazurile în care este necesară executarea corpului unui ciclu cel puţin o dată, urmînd ca ulterior să se execute în funcţie de îndeplinirea condiţiei finale, indicate în contextul ciclului *while*.

Instrucţiunea de ciclu aritmetic for are următoarea formă generală *for (expr_1; expr_2; expr_3) instrucţiune;*

şi este echivalentă cu următoarea succesiune de instrucţiuni:

```
 expr_1;
while (expr_2)
(   instrucţiune;
    expr_3;  )
```

Oricare dintre cele trei expresii poate lipsi, absenţa expresiei *expr_2* fiind înlocuită, implicit, cu valoarea 1. De exemplu,

```
for (int i=0; i<=n; i++) a[i]=0.0;
for ( int k=0, number_of_nums=0, number_of_chars=0; k<strlen(text); k++)
{  cout « text[k] «'\n';
 if (is_num(text[k]))        number_of_nums++;
 if (is_alpha(text[k]))      number_of_chars++;
}
```

Ciclul *for* este util de folosit atunci cînd există o simplă iniţializare şi reiniţializare, deoarece se păstrează instrucţiunile de control al ciclului împreună.

Instrucţiunea switch face parte din categoria instrucţiunilor de selectare. Transferul controlului se va efectua la una din variantele posibile, în funcţie de valoarea unei expresii de control. Sintaxa instrucţiunii este *switch (expr) instrucţiune;*
unde *instrucţiune* este o instrucţiune compusă, în care fiecare instrucţiune individuală trebuie etichetată cu o etichetă de forma

case expresie_constanta:
unde *expresie_constanta* trebuie să fie de tip *int* şi nu pot fi două etichete egale în aceeaşi instrucţiune *switch*. Cel mult o instrucţiune poate fi etichetată cu *default:*

La execuţia unei instrucţiuni *switch* se evaluează expresia *expr* şi se compară valoarea obţinută cu fiecare constantă ce apare în etichetele asociate instrucţiunii. Dacă se găseşte o astfel de constantă, controlul este dat instrucţiunii ce urmează ei, în caz contrar, controlul fiind transferat la instrucţiunea de după eticheta *default*, dacă aceasta există, sau instrucţiunii imediat următoare instrucţiunii *switch.*

Pentru a se menţiona sfîrşitul unei instrucţiuni ataşate unui caz, se va utiliza una dintre instrucţiunile *goto, break* sau *return*. La începutul unei instrucţiuni *switch* pot apărea declaraţii, dar nu se vor efectua iniţializări ale variabilelor de tip *auto* sau *register*. De exemplu,

```
switch (text[k])
{       case 'A' : număr_caractere++; break;
        case  'B': număr_caractere++; break;
        …           // se vor completa toate cazurile posibile
        case 'Z' : număr_caractere++; break;
        case ' a' : număr_caractere++; break;
        …           // se vor completa toate cazurile posibile
        case 'z': număr_caractere++; break;
        case '0': :număr cifre++;
        …           // se vor completa toate cazurile posibile
        case '9':număr cifre++; }
```

Instrucţiunea break are forma *break;*
Are ca efect terminarea execuţiei unui ciclu de tip *while, do-while, for* sau *switch,* controlul fiind transferat primei instrucţiuni din corpul blocului cel mai interior.

Instrucţiunea continue are forma *continue;*
Are drept efect trecerea controlului următorului ciclu într-o instrucţiune de tip *while* sau *for* în care apare şi nu are nici un efect dacă nu apare în corpul unor astfel de instrucţiuni. Cînd este întîlnită, ea se trece la următoarea iteraţie a ciclului (*while, for, do-while*). În cazul lui *while* şi *do-while,* aceasta înseamnă că partea de control se execută imediat. În cazul ciclului *for*, controlul va trece la faza de reiniţializare. De obicei, instrucţiunea *continue* nu se va aplica instrucţiunii *switch*. Ca exemplu, fragmentul următor sumează numai elementele pozitive dintr-un tablou *a*, în care valorile negative sînt omise.

```
int s;
for (int i=0,s=0; i<N; i++)
{   if (a[i]<0) continue; //sare indicii elementelor negative
s+=a[I]; }
```

Instrucţiunea return admite următoarele două forme
return; sau *return (expr);*

cea din urmă fiind echivalentă cu următoarea

return expr;

Efectul instrucţiunii *return* este trecerea controlului la funcţia care a apelat funcţia respectivă fără transmiterea unei valori în prima variantă sau cu transmiterea unei valori în ultimele două variante.

Instrucţiunea goto şi etichete. Limbajul C++ oferă instrucţiunea *goto* pentru ramificare. Formal, instrucţiunea *goto* nu este necesară şi uşor se poate scrie programe fără ea. Cu toate acestea, există cîteva situaţii în care *goto* îşi poate găsi locul. Cea mai obişnuită folosire este aceea de a abandona prelucrarea în anumite structuri puternic imbricate, de exemplu, de a ieşi afară din două cicluri deodată, instrucţiunea *break* nu poate fi folosită, deoarece ea părăseşte numai ciclul cel mai din interior. Astfel:

```
for(...)
 for(...)
  ( ...
  if(dezastru) goto error;}
…error:;
```

O posibilitate de căutare a primului element negativ într-un tablou bidimensional ar fi:

```
for (i=0; i<N; i++)
  for(j=0; j<M; j++)
    if (v[i][j]<0) goto found;
    found: // s-a găsit în poziţia i, j
```

Programul cu un *goto* poate fi scris întotdeauna fără *goto*, chiar dacă preţul pentru aceasta este o variabilă suplimentară sau nişte controluri repetate. De exemplu, căutarea în tablou devine:

```
found=0;
for (i=0; i<N && found; i++)
    for (j=0; j<M && found; j++) found = v[i][j]<0;
 if (found) {  ……...     } // a fost găsit la i-1, j-1
else {…….……...}        // nu s-a găsit
```

Instrucţiunea vidă are forma ";" şi este utilizată pentru a evita existenţa unei etichete chiar în faţa unei acolade de închidere a unui bloc sau în cazul în care corpul unui ciclu nu conţine nici o instrucţiune.

Tipuri de date recursive, operaţii asupra listelor, arborilor.

Listele simplu şi dublu lănţuite, arborii sînt formate din elemente definite de structuri cu autoreferire. Ele sînt consecutivităţi de elemente de acelaşi tip, numărul cărora se schimbă dinamic în procesul de executare a programului. Lista liniară *F*, care constă din elemente *D1, D2,...,Dn,* grafic se poate reprezenta în modul următor:

D1 D2 D3 ... Dn

Asupra elementelor listelor se pot efectua următoarele operaţii:

- de căutare a elementului după criteriul dat;
- de determinare a primului element în lista liniară;
- insertarea unui element nou înainte sau după o componentă indicată a listei liniare;
- eliminarea unui element din listă;
- sortarea componentelor listei.

Metodele de stocare a listelor liniare se divid în metode consecutive şi stocare lănţuită.

Elementele listei liniare, utilizate de metodele consecutive, se alocă într-un tablou *d* de dimensiune fixă, de exemplu, 100, şi lungimea listei este indicată de variabila *l*, adică se declară

float d[100]; int l;

Dimensiunea 100 mărgineşte dimensiunea maximală a listei liniare. Lista *F* în tabloul *d* se formează în modul următor:

d[0]=7; d[1]=10; l=2;

Lista obţinută se păstrează în memorie în conformitate cu schema:

l: 2

d: 7 10 ...

[0] [1] [2] [3] [98] [99]

Pentru organizarea elementelor în formă de listă simplu lănţuită, se utilizează structurile care sînt legate cîte o componentă în lanţ, începutul căreia (prima structură) este indicat de pointerul *dl*. Structura care defineşte elementul listei conţine în afară de componenta informaţională şi un pointer la următorul element din listă. Descrierea acestui tip de structură cu autoreferire şi pointerul în cauză se face în modul următor:

```
typedef struct nod   // structura cu autoreferire
{float  val;        // valoarea componentei informaţionale
 struct nod *urm ;    // pointerul la următorul element din lanţ
} DL;
DL *p;              // pointerul la elementul curent
DL *prim;           // pointerul la începutul listei
```

Pentru alocarea memoriei elementelor listei în C++, se utilizează *operatorul de alocare*: *new* care apare sub forma

pointer_la_nume = new nume [iniţializator];

care încearcă să creeze un obiect *nume* prin alocarea unui număr egal cu *sizeof(nume)* de octeţi în memoria heap, adresa acestuia este returnată şi asignată variabilei *pointer_la_nume.* În cazul în care alocarea nu este efectuată cu succes, se returnează valoarea *NULL*.

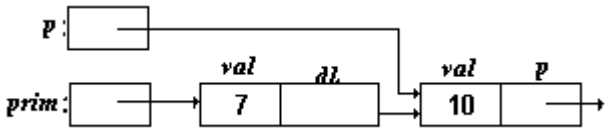
Operatorul de eliberare delete este apelat printr-o instrucţiune de forma

delete pointer_la_nume ;

eliberează memoria alocată începînd cu adresa conţinută de *pointer_la_nume*. De exemplu,

```
p=new(DL);
p->val=10;
p->n=NULL;
dl=new(DL));
dl->val=7;
 dl->n=p;
```

În ultimul element al listei pointerul la elementul vecin are valoarea *NULL*. Lista are următoarea formă:



Operaţii asupra listelor simplu lănţuite

Fiecare element al listei simplu lănţuite reprezintă o structură alcătuită din două componente: *val* – folosit pentru componenta informaţională şi *p* pentru pointer la următorul element din lista lănţuită. Pointerul *dl* indică adresa de alocare pentru primul element al listei. Pentru toate operaţiile asupra listei se va utiliza următoarea descriere a structurii elementelor liste:

```
typedef struct nod
          {  float val;
          struct nod * urm;
          }  NOD;
         int i,j;
         NOD * prim, * r, * p;
```

Pentru executarea operaţiilor pot fi utilizate următoarele fragmente de program:

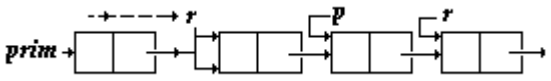
1) formarea listei simplu lănţuite:

```
 float x=5;    int n=1;
 p=new(nod);
 r=p;
p->val=x;
p->urm=NULL;
 prim=p;
 while (p->val !=0)
 {  p=new(nod); n++;
  p->val=x-1.0*n;
 r->urm=p;
 p->urm=NULL;
 r=p; }
```

2) tiparul elementului *j:*

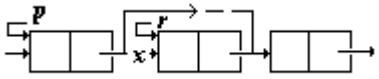
```
r=prim;j=2;
while(r!=NULL && j<n-1)
{ if (r==NULL) printf("\n nu este elementul  %d ",j);
else printf("\n elementul %d este egal cu %f ",j++,r->val);
r=r->urm;  }
```

3) tiparul ambilor vecini ai elementului determinat de pointerul *p* :



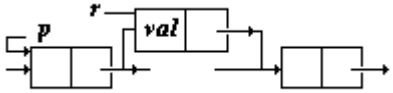

```
p=prim;
if((r=p->urm)==NULL) printf("\n nu are vecin din dreapta");
else  printf("\n  vecinul din dreapta este  %f",  r->val);
if(prim==p) printf("\n nu are vecin din stînga" );
else {   r=prim;
     while( r->urm!=p ) r=r->urm;
     printf("\n vecinul de stînga este %f", r->val);     }
```

4) eliminarea elementului care este succesorul elementului în cauză, la care indică pointerul *p*




```
p=prim;
if ((r=p->urm)==NULL) printf("\n  nu este succesorul ");
p->urm=r->urm; delete(r->urm);
```

5) insertarea noului element cu valoarea *newval=100* după elementul determinat de pointerul *p*:



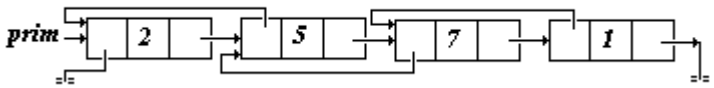

```
r=new(NOD);
r->urm=p->urm;   r->val=100;   p->urm=r;
```

Organizarea listelor dublu lănţuite

Lista dublu lănţuită este o listă în care fiecare element conţine doi pointeri: unul la precedentul element, altul – la succesorul element din listă. Lista dublu lănţuită în program se poate determina cu ajutorul următoarelor descrieri:

```
  typedef struct ndd
{  float val;      // valoarea informaţională a componentei
 struct ndd * succesor; // pointer la succesorul element al //listei n
 struct ndd *precedent; // pointer la precedentul element al //listei m
     } NDD;
  NDD * prim, * p, * r;
```

Interpretarea grafică a listei *F=< 2,5,7,1 >* ca listă dublu lănţuită este următoarea:



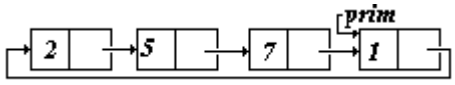
Insertarea noului element cu valoarea *newval* după elementul determinat de pointerul *p*, se efectuează de operatorii

```
 r=new(NDD);
r->val=newval;
r->succesor=p->succesor;
(p->succesor)->precedent=r;
p->=r;
```

Eliminarea elementului urmat de elementul la care indică pointerul *p* se efectuează în modul următor:

```
p->succesor=r;
p->succesor=(p->succesor)->succesor;
( (p->succesor)->succesor )->precedent=p;
delete r;
```

Lista liniară este *ciclică*, dacă ultimul element al listei indică la primul element, iar pointerul *dl* indică la ultimul element al listei. Schema listei ciclice pentru lista *F=< 2,5,7,1 >* este următoarea:



La rezolvarea problemelor pot apărea diferite tipuri de liste lănţuite.

Stivă şi coadă

În funcţie de metoda de acces la elementele listei liniare, pot fi cercetate următoarele tipuri de liste liniare: stive, cozi şi cozi de tip vagon.

Stiva este o consecutivitate de elemente de acelaşi tip – variabile scalare, tablouri, structuri sau uniuni. Stiva reprezintă o structură dinamică, numărul de elemente a căreia variază. Dacă stiva n-are elemente, ea este vidă.

Asupra elementelor stivei pot fi efectuate următoarele operaţii:

- verificarea dacă stiva este vidă,
- includerea unui element nou în vîrful stivei;
- eliminarea elementului din vîrful stivei;
- accesarea elementului din vîrful stivei, dacă stiva nu este vidă.

Astfel, operaţia de includere şi eliminare a elementului, de asemenea, accesarea elementului are loc numai asupra elementului din vîrful stivei.

Coada este o listă liniară în care elementele listei se elimină din capul listei şi elementele noi se includ prin coada listei.

Coadă de tip vagon este o listă liniară în care includerea şi eliminarea elementelor din listă se efectuează din ambele capete (vîrful şi sfîrşitul) ale listei.

Stiva şi coada se organizează atît static prin intermediul tabloului, cît şi dinamic – prin listă (simplu sau dublu lănţuită).

Vom cerceta cum se utilizează lista în formă de stivă pentru implementarea calculării expresiei aritmetice în formă inversă poloneză. În astfel de mod de prezentare a expresiei operaţiile se înregistrează în ordinea executării lor, iar operanzii se află nemijlocit în faţa operaţiei. De exemplu, expresia (6+8)*5-6/2 în forma inversă poloneză are forma: *6 8 + 5 * 6 2 / -*

Utilizînd noţiunea de stivă, expresia aritmetică în formă inversă poloneză se execută print-o singură trecere de examinare a expresiei. Fiecare număr se introduce în stivă, iar operaţia se execută asupra următoarelor două elemente din vîrful stivei, înlocuindu-le cu rezultatul operaţiei efectuate. Dinamica schimbărilor din stivă va fi următoarea:

S = < >; <6>; <6,8>; <14>; <14,5>; <70>;
<70,6>; <70,6,2>; <70,3>; <67>.

Mai jos este descrisă funcţia *eval*, care calculează valoarea expresiei indicate în tabloul *m* în formă de expresie inversă poloneză, *m[i]>0* indică numărul nenegativ, iar valoarea *m[i]<0* - operaţia. În calitate de coduri pentru operaţiile de adunare, scădere, înmulţire şi împărţire se aleg numerele: -1, -2, -3, -4. Pentru organizarea stivei se utilizează tabloul interior *stack*. Parametrii funcţiei sînt tabloul de intrare *m* şi lungimea sa *l*.

```
float eval (float *m, int l)
{ int p,n;
  float stack[50],c;
   for(int i=0; i < l ;i++)
   if ((n=m[i])<0)
    {  c=st[p--];
       switch(n)
       { case -1:  stack[p]+=c;  break;
         case -2:  stack[p]-=c;  break;
         case -3:  stack[p]*=c;  break;
         case -4:  stack[p]/=c;   }  }
```

else stack[++p]=n;
return(stack[p]); }

Arbori

Arborii sînt structuri de date dinamice, cu autoreferire. Prin arbore se înţelege o mulţime finită şi nevidă de elemente (noduri): *A={A1, A2,..., An}, n>0* cu următoarele proprietăţi:

- există un nod şi numai unul care se numeşte *rădăcina arborelui*,
- celelalte noduri formează submulţimi ale lui *A*, care formează fiecare cîte un arbore, arborii respectivi se numesc *subarbori ai rădăcinii*.

Într-un arbore există noduri cărora nu le corespund subarbori. Astfel de noduri se numesc *terminale*.

În multe aplicaţii se utilizează noţiunea de *arbori binari*. Dacă mulţimea de elemente a arborelui binar este vidă, se consideră că arborele constă numai din rădăcină. Dacă mulţimea de elemente este nevidă, arborele binar se divide în două submulţimi: *subarborele drept* şi *cel de stînga*. Arborele binar este ordonat, deoarece în fiecare nod subarborele stîng se consideră că precede subarborele drept. Un nod al unui arbore binar poate să aibă numai un descendent: subarborele drept sau subarborele stîng. De exemplu, un nod al unui arbore binar poate fi o structură care poate fi definită în felul următor:

```
typedef struct tnod
 {  int nr,     int f;          //declaraţii
  struct tnod *st; // este pointerul spre subarborele stîng al //nodului curent
  struct tnod *dr; // este pointerul spre subarborele drept al //nodului curent
} TNOD;
```

Asupra arborilor binari pot fi definite următoarele operaţii:

- afişarea componentelor informaţionale ale nodului,
- specificarea criteriului de determinare a poziţiei în care să se inserteze în arbore nodul curent;
- determinarea echivalenţei a doi arbori;
- insertarea unui nod terminal într-un arbore binar;
- accesarea unui nod al arborelui,
- parcurgerea unui arbore;
- ştergerea unui arbore.

Afişarea componentelor informaţionale ale nodului se poate de efectuat prin funcţia:

```
void prelucrare (TNOD *p)
{printf("numărul = %d apariţii= %d \n", p->nr, p->f);}
```

Criteriul de determinare a poziţiei, în care să se inserteze în arbore nodul curent, se defineşte de funcţia:

```
int criteriu(TNOD *p, *q)
{ if (q->nr < p -> nr )
return –1; // insertarea nodului curent
            //în subarborele stîng al nodului spre care indică
            //pointerul p
 if (q->nr > p-> nr )
 return 1;   // insertarea nodului curent
             //în subarborele drept al nodului spre care indică      //pointerul p
}
```

Insertarea unui nod terminal într-un arbore binar poate fi efectuată prin următoarea funcţie:

```
TNOD* insert_nod()
{  TNOD *parb, *p, *q;
   int  n=sizeof(TNOD);
  if (parb ==0)
  { parb=p;  return p; }
int i;
q=parb;
for(;;)
  if ((i=criteriu(q,p)) <0)  {q->st=p;    return p; }
  else  {   q=q->st;   continue;   }
if (i>0)
        if (q->dr ==0)
          {q->dr=p;   return p;}
        else  {q=q->dr;  continue;  }
return eq(q,p); }
}
if(p==0)
{  printf("eroare: memorie insuficientă\n");  exit(1);}
elibnod(p);      return 0;   }
```

Accesarea unui nod al unui arbore poate fi realizată prin următoarea funcţie:

```
TNOD * cauta (TNOD *p)
{TNOD *parb, *q;
if (parb==0) return 0;
int i;
for (q=parb;q;)
     if ((i=criteriu(q,parb))==0) return q;
     else     if(I<0)    q=q->st;
              else       q=q->dr;
return 0;  }
```

Parcurgerea unui arbore poate fi efectuată în trei modalităţi: *în preordine; în inordine; în postordine.*

Parcurgerea *în preordine* presupune accesul la rădăcină şi apoi parcurgerea celor doi subarbori ai săi: mai întîi subarborele stîng, apoi cel drept.

```
void preord (TNOD *p)
{ if (p!=0)
 {   prelucrare(p);  preord(p->st);  preord(p->dr); }
}
```

Parcurgerea în inordine presupune parcurgerea mai întîi a subarborelui stîng, apoi accesul la rădăcină şi în continuare se parcurge subarborele drept.

```
void inord (TNOD *p)
{   if (p!=0)
    {inord(p->st);   prelucrare(p);  inord(p->dr);}
}
```

Parcurgerea *în postordine* presupune parcurgerea mai întîi a subarborelui stîng, apoi a arborelui drept şi, în final, accesul la rădăcina arborelui.

```
void postord (TNOD *p)
{  if (p!=0)
    {  postord(p->st);  postord(p->dr);  prelucrare(p);  }
}
```

Ştergerea unui arbore poate fi efectuată de următoarea funcţie:

```
void elib_nod(TNOD *p)
{   delete(p);  }
void sterge_arbore (TNOD *p)
{  if (p!=0)
    {   postord(p->st);  postord(p->dr);  elibnod(p);  }
}
```

Recursivitatea ca metodă de programare

Recursivitatea presupune o repetare. Ea constă în apelarea unei funcţii de către ea însăşi.

Funcţia se numeşte *recursivă* dacă în momentul executării sale funcţia se apelează pe ea însăşi, sau indirect, printr-o succesivitate de apeluri ale altor funcţii.

Funcţie este nemijlocit recursivă dacă ea se apelează din corpul aceleiaşi funcţii. De exemplu:

```
int a()
{.....a().....}
```

Funcţia este indirect recursivă dacă se efectuează apel recursiv prin intermediul unei succesivităţi de apeluri ale altor funcţii. Toate funcţiile componente ale acestei succesivităţi de apeluri se socot recursive. De exemplu,

```
a(){.....b().....}
b(){.....c().....}
c(){.....a().....} .
```

Funcţiile *a,b,c* sînt recursive, deoarece la apelul unei din funcţii are loc apelul altor funcţii inclusiv şi pe ea însăşi.

Execuţia algoritmului recursiv presupune crearea unui număr (finit) de copii ale algoritmului, care corespund diferitelor valori ale unei variabile. În construirea algoritmului recursiv este inclusă o condiţie de terminare a apelării recursive de o expresie; care prin apelări succesive valoarea ei creşte pînă la o valoare ce satisface condiţia de finalizare a recursivităţii. La executarea programului cu funcţii recursive se creează copii ale acestora, fiecare din ele corespunzînd unei valori a expresiei de recursie. Atît timp cît expresia recursiei se calculează pînă cînd creşte pînă la o valoare ce satisface condiţia de finalizare a recursivităţii, se spune că are loc recursia *înainte*. Cînd expresia atinge valoarea soluţiei recursiei, se execută copiile create, astfel încît se obţine soluţia problemei. În acest caz are loc recursia *înapoi.* Executarea programelor cu funcţii recursive necesită multă memorie şi mult timp de calcul, cu *o complexitate mai mare* decît cele nerecursive.

Recursivitatea ca metodă de programare este mai eficientă, codul programelor cu funcţii recursive este mai compact şi mai uşor de înţeles.

În limbajul C++ funcţiile pot să se autoapeleze. Exemplul clasic de funcţie recursivă este calcularea factorialului numărului *N! = 1*2*3*...*N.*

Vom numi această funcţie *factorial().*

```
long factorial(int n) {return((n==1)?1: n*factorial(n-1) ); }
```

Apelul funcţiei recursive creează noi copii ale variabilelor locale şi ale parametrilor pentru clasa de memorie *auto* şi *register*, valorile lor din apelurile precedente se păstrează. Pentru fiecare moment sînt accesibile numai valorile apelului curent. Variabilele declarate cu clasa de memorie *static* nu necesită crearea noilor copii. Valorile lor sînt accesibile în orice moment de executare a programului. În corpul funcţiei recursive este necesar de indicat condiţia de ieşire din procesul recursiv, în caz contrar sistemul de calcul poate intra în impas. De exemplu, funcţia de tipărire unui număr (ca un şir de caractere): poate fi apelată de ea însăşi, adică să fie recursivă

```
void  print_cifre(int n)
{ int i;
 if (n<0)
{  putchar('-'); n=-n;  }
if ((i=n/10)!=0) print_cifre(i);
putchar(n%10+'0');  }
```

Programul de mai jos calculează funcţia Akkerman cu utilizarea funcţiei recursive *ackr* şi funcţiei auxiliare *smacc*:

```
//     calculul recursiv al funcţiei Akkerman
 # include <stdio.h>
int smacc( int n,int x )         //  funcţie   auxiliară
int ackr( int n, int x, int y)    //  funcţie   recursivă
      void   main ()             //  funcţia în care se apelează funcţia
     {  int x,y,n,t;
     int ackr(int, int, int);
     scanf("%d %d %d",&n,&x,&y);
     t=ackr(n,x,y);
     printf("%d",t);    }
int smacc( int n,int x )         //  funcţie   auxiliară
{   switch (n )
    {  case 0:  return(x+1);
       case 1:  return (x);
       case 2:  return (0);
       case 3:  return (1);
       default: return (2);    }
  }
int ackr( int n, int x, int y)    //  funcţie   recursivă
{  int z;
   int smacc( int,int);
   if(n==0 || y==0)  z=smacc(n,x);
   else  { z=ackr(n,x,y-1);        //  apeluri   recursive ackr(...)
         z=ackr(n-1,z,x);   }
   return z;      }
```

Rezultatul îndeplinirii programului:

```
1 4 6       // datele iniţiale
            -
10          // rezultatul obţinut
```

Fişierele input/output ale limbajul C++. Deschiderea şi închiderea fişierelor. Citirea şi scrierea în fişiere.

Limbajul C++ include în sine funcţiile standard input/output ale limbajului C de prelucrare a fişierelor la nivelul jos, inferior şi superior.

Funcţiile de prelucrare a fişierelor de nivel inferior pot fi utilizate prin includerea fişierelor *io.h, fcntl.h* şi *stat.h.*

Pentru deschiderea unui fişier se utilizează funcţia *open* care are următoarea formă sintactică:

df = open(…);

unde *df* este variabilă de tip *int* (descriptorul de fişier).

Funcţia *open* are următorii parametri: se indică *calea de acces* la fişier şi *modalitatea de accesare* a componentelor fişierului. Modalitatea de accesare a componentelor se indică prin una din următoarele constante:

O_RDONLY	fişierul se deschide numai pentru citirea componentelor lui
O_WRONLY	fişierul se deschide numai pentru înregistrarea componentelor lui
O_RDWR	fişierul se deschide pentru citirea şi înregistrarea componentelor lui
O_APPEND	fişierul se deschide pentru adăugarea componentelor noi la sfîrşitul lui
O_BINARY	fişierul se prelucrează binar
O_TEXT	fişierul se prelucrează textual

Pentru a crea un fişier, se utilizează funcţia *creat* cu următorii parametri: *calea de acces* la fişierul creat şi *modalitatea de utilizare* a fişierului. Al doilea parametru se indică de una din următoarele constante:

S_IREAD	fişierul va fi creat numai pentru citire
S_IWRITE	fişierul va fi creat numai pentru înregistrare
S_IEXE	fişierul va fi creat numai pentru executare

Citirea dintr-un fişier se efectuează prin funcţia *read* indicîndu-se următorii parametri:

read(df, buf, lung)

unde *df* este descriptor de fişier; *buf* – pointer spre zona de memorie în care se va păstra înregistrarea citită din fişier; *lung* – lungimea în octeţi a înregistrării citite.

Înregistrarea în fişier se efectuează prin funcţia *write*. Această funcţie are aceiaşi parametri ca şi funcţia *read*.

Poziţionarea într-un fişier se efectuează prin funcţia *fseek(df, deplasare, origine)*. Această funcţie are următorii parametri: *df* – descriptorul fişierului, *deplasarea* indică numărul de octeţi pentru a deplasa capul de citire sau scriere al discului, *origine* are următoarele valori pentru efectuarea deplasării: 0 – faţă de începutul fişierului, 1- faţă de poziţia curentă a capului de citire sau înregistrare, 2 – faţă de sfîrşitul fişierului.

Închiderea fişierului se efectuează de funcţia *close (df).*

Exemplu de utilizare ale acestor funcţii:

```
char nfis[]="fisier1.dat";
int df;      char *p;
df=open(nfis,O_RDONLY);
read(df,p,80);
close(df);
```

Prototipurile funcţiilor de prelucrare a fişierelor de nivel superior pot fi utilizate prin includerea fişierului *stdio.h.*

Fişierele input/output standard se efectuează prin intermediul funcţiilor *scanf* şi *printf, gets, getc, getch* şi *puts, putc*, respective.

Funcţiile *getc(), getch()* citesc cîte un caracter din fişierul standard input **.**

Funcţiile *scanf* şi *fscanf, printf* şi *fprintf* permit citirea, respectiv, afişarea uneia sau a mai multor valori la intrarea standard sau dintr-un fişier, respectiv, ieşirea standard sau înregistrare într-un fişier. Prototipurile acestor funcţii se află în biblioteca *stdio.h.*

Principiul de utilizare al funcţiei *printf* constă în asocierea unei liste, care conţine indicaţii de formatare, dată sub forma unui şir de caractere, o listă de variabile. Ambele funcţii utilizează specificaţiile de scriere sau citire plasate într-o constantă de tip şir de caractere, urmată de o listă de argumente.

Funcţia de afişare *printf* utilizează ca argumente nume de variabile, iar funcţia de citire *scanf* utilizează drept argumente adrese de variabile. De exemplu,

```
#include<stdio.h>
void main()
{printf("intreg:%6i \n  real: %9.3f   ",316,144.82) ;
int z;
printf("Introdu valoarea z:");
scanf("%d",&z);
printf("%6d",z);
}
```

Rezultatul îndeplinirii programului:

```
intreg:   316
real:   144.820   Introdu valoarea z:500
Valoarea z:   500
```

Lista este parcursă de la stînga la dreapta. Fiecare semn este asociat cu caracterul care îl urmează şi este interpretat drept caracter de control. Caracterele de control utilizate sînt:

\n	avans la început de linie nouă;
\r	poziţionare la începutul liniei curente;
\t	tabulator;
\a	emite un semnal sonor .

Fiecare semn % este interpretat ca începutul descrierii caracteristicilor de tipărire a unei valori. Cele mau utilizate semne sînt următoarele:

Semnul	Descrierea
%d,	un întreg zecimal este aşteptat la intrare; argumentul
% i	corespunzător trebuie să fie un pointer la întreg;
%o	un întreg octal este aşteptat la intrare; argumentul corespunzător trebuie să fie un pointer la întreg;
%x	un întreg hexazecimal este aşteptat la intrare;argumentul corespunzător trebuie să fie un pointer la întreg;
%h	un întreg short este aşteptat la intrare; argumentul trebuie să fie un pointer la un întreg short;
%u	un întreg fără semn zecimal este aşteptat la intrare; argumentul să fie pointer la întreg;

%f	un număr în virgulă flotantă este aşteptat; argumentul corespunzător trebuie să fie un pointer la un cîmp *float.* Caracterul de conversie e este **f.* Formatul prezentat la intrare pentru un float este alcătuit dintr-un semn opţional;
%e	un număr în virgulă flotantă este aşteptat; argumentul corespunzător trebuie să fie un pointer la un cîmp *double*. Caracterul de conversie e este **e*. Formatul prezentat la intrare pentru un double este alcătuit dintr-un semn opţional, un şir de numere care pot să conţină şi un punct zecimal şi un cîmp de exponent care este format din *E* sau *e*, urmat de un întreg cu semn;
%c	un singur caracter este aşteptat la intrare; argumentul corespunzător trebuie să fie un pointer la caracter. În acest caz, ignorarea caracterelor albe este suprimată; pentru a citi următorul caracter, altul decît caracterele albe se va utiliza *%1s*;
%s	un şir de caractere este aşteptat; argumentul corespunzător trebuie să fie un pointer al unui tablou de caractere, destul de mare pentru a încăpea şirul şi un terminator '*\0*', care va fi adăugat.

Caracterele de conversie *d, u, i ,o,* şi *x* pot fi precedate de litera *l*, pentru a indica un pointer la *long*, mai degrabă decît la *int*, care apare în lista de argumente. Similar, litera *l* înaintea lui *e* sau *f* indică un pointer la double în lista de argumente. De exemplu:

```
int i;
float  x;
char  nume[50];
scanf ("%d%f%s",&i,&x,nume) ;
```

cu linia de intrare

```
25 244.32E-1 Mircea
```

va asigna lui *i* valoarea 25, lui *x* valoarea 244.32E-1, iar lui *nume* valoarea "Mircea". Cele trei cîmpuri de la intrare pot fi separate de oricîte spaţii, taburi şi caractere de linie nouă. Apelarea

```
int i;
float  x;
char  nume[50];
scanf ("%2d%4.2f%2s",&i,&x,nume) ;
```

cu linia de intrare

```
25 244.32E-1 Mircea
```

va asigna 25 lui i, 44.32 lui x, iar nume va obţine valoarea "Mi".

Cele mai utilizate secvenţe asociate valorilor de tip întreg sînt *%ssNX* sau *%ssNU*, unde *s este* semnul + dacă se doreşte afişarea explicită a semnului, - arată că se va face o aliniere la stînga. N este un număr care arată pe cîte poziţii se va face afişarea. De exemplu,

```
#include<stdio.h>
void main()
{ printf("% -+5i",3);  }
```

programul indicat va afişa valoarea +3 prin aliniere la stînga în conformitate cu specificaţiile date, astfel semnul - cere alinierea la stînga, în cîmpul afectat valorii, semnul + cere afişarea explicită a semnului, cifra 5 arată că afişarea se va face pe 5 poziţii, simbolul i arată că va fi afişată o valoare de tip întreg.

Valorile de tip real pot fi tipărite utilizînd secvenţe asociate de forma ***%ssNMf*** sau *%sSN.tte*, în care simbolul *M* semnifică precizia cu care vor fi reprezentate numerele (numărul de cifre după punctul zecimal).

Toate caracterele care nu aparţin secvenţelor de control sînt afişate şi sînt tratate ca şiruri de caractere.

De exemplu,

```
#include<stdio.h>
void main ()
{ char * p="abracadabra";
char ch='B';
 printf("%s %c ", p, ch);  }
```

programul afişează şirul de caractere adresat prin intermediul pointerului *p* şi valoarea variabilei *ch*.

Funcţia *scanf* se utilizează la iniţializarea unor variabile. Funcţiile *scanf* şi *printf* sînt utilizate împreună. Prin funcţia *printf* se afişează un mesaj care adesea este un comentariu legat de valoarea care urmează să fie introdusă. De exemplu, în programul de mai jos am afişat mesajul "*număr real*", după care urmează apelul funcţiei *scanf* care aşteaptă introducerea unei valori de tip real sau întreg:

```
#include<stdio.h>
void main()
{ float a;
  int i;
printf ("Introdu un număr real: ");//utilizatorul va introduce, //la
scanf("%f", &a);        //tastatura, un număr urmat de Enter.
printf("\n Introdu un număr intreg: ");  //la fel se procedează //în cazul variabilei i
scanf ("%i,", &i);   // afişarea textului "număr intreg"
printf("\n Număr întreg: %6i \n Număr real: %9.3f", i, a);
} // va fi urmată de introducerea, la tastatură, a numărului    //dorit.
```

Rezultatul îndeplinirii programului:

```
Introdu un număr real: 3.14
 Introdu un număr intreg: 20
 Număr întreg:     20
 Număr real:     3.140
```

Funcţiile de scriere şi citire anterioare pot fi folosite şi în cazul fişierelor. Biblioteca *stdio.h*, specifică limbajului C, conţine definiţia unui tip de date *FILE*. Accesul la un fişier se face printr-un pointer de tip *FILE*. Etapele care trebuie să fie parcurse sînt definirea unui pointer de tip *FILE* şi asocierea unui fişier fizic. Pointerul va primi drept valoare adresa unei variabile de tip *FILE*, obţinută prin intermediul funcţiei *fopen()*. Această funcţie are două argumente: un şir de caractere care conţine numele fişierului fizic recunoscut de sistemul de operare şi un alt şir de caractere care conţine indicaţii relativ la modul de utilizare al fişierului. Ultimul parametru poate conţine caracterul

R pentru fişiere deschise pentru citire,

W pentru fişiere deschise pentru creare sau scriere,
T pentru fişiere de tip text sau
B pentru fişiere binare

În exemplul de mai jos am creat un fişier prin intermediul unui pointer de tipul *FILE* şi o iniţializare prin funcţia *fopen()*. Pointerul *p* conţine o valoare de tip *FILE* fumizată de funcţia *fopen()*, care deschide pentru înregistrare fişierul *disk.dat*. Am scris în acest fişier două valori în acelaşi mod cum am făcut afişarea la ieşirea standard.

```
#include<stdio .h>
void main()
{  FILE *p;
   p = fopen("disk.dat","wt");
  fprintf(p,"%6i\n%9.3f", 29, 2.71);
  fclose (p) ;
}
```

Rezultatul îndeplinirii programului:
s-a format fişierul “disc.dat” (pentru înregistrare) cu următorul conţinut:

```
  29
   2.710
```

Sarcina pentru lucrările de laborator:

1.

a. Scrieţi un program care ar număra biţii semnificativi, de la dreapta spre stînga, pentru un număr introdus de la tastatură.

b. Scrieţi un program care compară două stive de numere întregi.

2.

a. Scrieţi un program care efectuează înmulţirea cifrelor unui număr dat.

b. Scrieţi un program care calculează numărul de elemente dintr-o listă simplu lănţuită care sînt mai mici ca valoarea medie aritmetică a tuturor elementelor acestei liste.

3.

a. Scrieţi un program care determină un număr obişnuit din inversul cifrelor numărului dat.

b. Scrieţi un program care converteşte întregii fără semn dintr-o listă dublu lănţuită în reprezentare binară.

4.

a.Scrieţi un program care converteşte un întreg într-un număr hexazecimal.

b. Scrieţi un program care permite crearea unui arbore binar şi traversarea lui în inordine, preordine, postordine

5.

a. Scrieţi un program care calculează cel mai mare divizor comun al elementelor dintr-o consecutivitate.

b. Scrieţi un program care determină numărul de ordine a numărului minimal dintr-o listă dublu lănţuită.

6.

a. Scrieţi un program care calculează suma cifrelor pentru fiecare număr din consecutivitatea de *100* de numere aleatoare.

b. Scrieţi un program care determină numărul maximal şi cel minimal într-o listă circulară de 100 de numere aleatoare. Să se determine consecutivitatea de elemente ce se află între numerele maximal şi cel minimal determinate.

7.

a. Scrieţi un program care inversează un şir de caractere s.

b. Scrieţi un program care inversează cele *n* elemente ale unei liste simplu lănţuită care încep de pe poziţia *p*, lăsîndu-le pe celelalte locul lor.

8.

a. Scrieţi un program care calculează cel mai mare divizor comun dintr-un şir de numere date.

b. Scrieţi un program care atribuie unei liste simplu lănţuite elementele altei liste în ordine inversă.

9.

a. Scrieţi un program care determină cîte numere din consecutivitatea de 100 de numere aleatoare sînt mai mari ca "vecinii" săi.

b. Scrieţi un program care decide dacă o valoare *x* aparţine unei liste dublu lănţuite *v*. Elementele lui *v* trebuie să fie în ordine crescătoare. Se tipăreşte numărul elementului din listă (un număr între 0 şi n-1), dacă x apare în *v***,** şi –1, dacă nu apare.

10.

a.Scrieţi un program care converteşte un întreg într-un număr binar.

b.Scrieţi un program care va tipări în ordine inversă subconsecutivitatea de numere dintre valoarea minimă şi maximă ale unei liste simplu lănţuită.

11.

a. Scrieţi un program care converteşte numărul întreg n în baza zecimală într-un şir de caractere.

b. Scrieţi un program care inversează fiecare element de tip şir de caractere dintr-o listă simplu lănţuită.

12.

a. Scrieţi un program care converteşte literele mari în litere mici al unui şir de caractere.

b. Să se scrie un program care din lista *L1* ce conţine numere întregi să se extragă în lista *L2* elementele cu numere impare din lista *L1*.

13.

a.Scrieţi un program care converteşte un întreg într-un număr octal.

b. Scrieţi un program care roteşte fiecare element al listei dublu lănţuite la dreapta cu *b* poziţii.

14.

a. Scrieţi un program de convertire a unui număr întreg într-un şir de caractere.

b. Scrieţi un program care creează o listă circulară a căror valori ale elementelor sînt cuprinse între 1 şi 100. Să se determine frecvenţa cu care a fost generat fiecare element al listei create.

15.

a. Scrieţi un program care din 100 de numere aleatoare se determină numărul maximal şi cel minimal. Să se determine diferenţa dintre numărul maximal şi cel minimal determinat.

b. Scrieţi un program care determină cîte numere ale unei cozi de 100 de numere aleatoare sînt mai mari ca "vecinii" săi.