Introducere

Apărut în 1972, *limbajul C* este un limbaj de nivel înalt cu utilizare universală (neorientat spre un domeniu limitat). Autorii acestui limbaj sunt Dennis M. Ritchie şi Brian W. Kernighan de la Bell Laboratories.

Limbajul C a fost proiectat în ideea de a asigura implementarea portabilă a sistemului de operare UNIX. Un rezultat direct al acestui fapt este acela că programele scrise în limbajul C au o *portabilitate* foarte bună.

În mod intuitiv, spunem că un program este *portabil* dacă el poate fi transferat uşor de la un tip de calculator la altul. În principiu, toate limbajele de nivel înalt asigură o anumită portabilitate a programelor. În prezent, se consideră că programele scrise în C sunt cele mai portabile.

După 1980, limbajul C a fost utilizat de un număr tot mai mare de programatori, cunoscând o răspândire spectaculoasă. În prezent are o poziție dominantă în domeniul limbajelor de programare. Principalele caracteristici care au asigurat succesul său, sunt:

- Se utilizează pentru aplicații ştiințifice şi tehnice, cu facilități grafice;
- Permite accesul la toate resursele *hardware* ale calculatorului până la nivel de locație de memorie, ceea ce se obține în general numai în limbaj de asamblare;
- Este un limbaj profesional, care prezintă atât avantajele unui limbaj de nivel înalt cât şi avantajele limbajelor de nivel coborât (l. maşină, l.de asamblare);
- Este un limbaj *structurat* sau *procedural* (ca şi Pascal) şi bazat pe compilator; forma ***.exe*** a unui program poate fi rulată fără o nouă complilare;
- Unitatea de bază a unui program în C este funcția; limbajul dispune de o mare varietate de clase de funcții predefinite, puse la dispoziția utilizatorului, fiind grupate în biblioteci;
- Se consideră că efortul pentru scrierea unei aplicații în C este mai mic decât în Pascal dar durata de învățare este mai mare.

Odată cu evoluția limbajului (şi răspândirea sa) a fost necesară adoptarea unui standard internațional care să definească limbajul, biblioteca de funcții şi să includă o serie de extensii utile, impuse de marea varietate de aplicații. Limbajul C - standard sau C - ANSI

(*American National Standard Institute*) a apărut în 1988 şi reprezintă varianta inclusă în toate dezvoltările ulterioare ale limbajului (cu mici excepţii).

Anii '90 sunt consacraţi programării orientate spre obiecte (*Object Oriented Programming* - OOP) la fel cum anii '70 au fost numiţi anii *programării structurate*.

Programarea orientată spre obiecte este un stil de programare care permite abordarea eficientă a aplicaţiilor complexe, ceea ce se realizează, în principiu, prin:

- ♦ elaborarea de componente reutilizabile, extensibile şi uşor de modificat, fără a reprograma totul de la început;
- ♦ construirea de biblioteci de module extensibile;
- ♦ implementarea simplă a interacţiunilor dintre programe şi sistemul de calcul, prin utilizarea unor elemente standardizate.

În anii '80, interesul pentru programarea orientată spre obiecte a crescut, ceea ce a condus la apariţia de limbaje de programare care să permită utilizarea ei. Limbajul C a fost dezvoltat şi el în această direcţie, astfel că în 1980 a fost lansat limbajul C++. Acesta, elaborat de Bjarne Stroustrup de la AT&T, este o extindere a limbajului C şi permite utilizarea principalelor concepte ale programării orientate spre obiecte.

Limbajul C++ a fost implementat pe microcalculatoare compatibile IBM PC, în mai multe variante. Cele mai importante implementări sunt cele realizate de firmele Borland şi Microsoft.

Conceptele programării orientate spre obiecte au influenţat în mare măsură dezvoltarea limbajelor de programare în ultimii ani. Multe limbaje au fost extinse astfel încât să permită utilizarea conceptelor programării orientate spre obiecte. Au apărut chiar mai multe extensii ale aceluiaşi limbaj. De exemplu, în prezent asistăm la extensiile limbajului C++. O astfel de extensie este *limbajul E* ai cărui autori sunt Joel E. Richardson, Michael J. Carey şi Daniel T. Shuh, de la universitatea Wisconsin Madison.

Limbajul E a fost proiectat pentru a permite exprimarea simplă a tipurilor şi operaţiilor interne, pentru sistemele de gestiune a bazelor de date.

O altă extensie a limbajului C++ este *limbajul O*, dezvoltat la Bell Laboratories. Cele două limbaje (E şi O) sunt în esenţă echivalente.

Interfeţele de utilizator au atins o dezvoltare mare datorită facilitaţilor oferite de componentele hardware ale diferitelor tipuri de calculatoare. În principiu, ele simplifică interacţiunea dintre programe şi utilizatorii lor. Astfel, diferite comenzi, date de intrare şi rezultate

pot fi exprimate simplu şi natural utilizând ferestre de dialog, bare de meniuri, cutii, butoane etc.

Implementarea interfețelor de utilizator este mult simplificată prin utilizarea limbajelor orientate spre obiecte. Aceasta, mai ales datorită posibilităților de utilizare a componentelor standardizate aflate în biblioteci specifice.

Firma Borland comercializează o bibliotecă de componente standardizate pentru implementarea interfețelor de utilizator folosind unul din limbajele C++ sau Pascal. Produsul respectiv se numeşte Turbo Vision.

De obicei, interfețele de utilizator gestionează ecranul în modul grafic.

Una din cele mai populare interfețe de utilizator grafice, pentru calculatoarele IBM PC, este produsul *Windows* al firmei Microsoft. Lansat în 1985, produsul Windows a parcurs mai multe variante şi are un succes fără precedent. Începând cu Windows 95, a devenit *sistem de operare,* care a înlocuit treptat sistemul MS DOS.

Aplicațiile sub Windows se pot dezvolta folosind diferite medii de programare, ca: Turbo C++, Pascal, Microsoft C++7.0, Microsoft Visual Basic, Visual C şi Visual C++.

1 Elemente de bază ale limbajului C

1.1. Setul de caractere

Reprezintă mulţimea de caractere ce pot fi utilizate în textul unui program. Setul de caractere al limbajului C este un subset al standardului ASCII (*American Standard Code for Information Interchange*) şi conţine:

- ♦ Literele mari: A, B, C, . . . , V, W, Z;
- ♦ Literele mici: a, b, c, . . . , v, w, z;
- ♦ Cifrele zecimale: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;
- ♦ Caractere speciale (29) :

. , ; : ? ' " ( ) [ ] { } ! | \ / ~ _ # % & ^ + - * = < >

Alte caractere şi semne pot să apară în comentarii, pot fi constante de tip caracter sau şiruri de caractere.

Caractere de tip spaţiu (neimprimabile):

\n - *newline* (produce trecerea la începutul rândului următor);

\t - tab. orizontal (introduce mai multe spaţii libere succesive; se utilizează pentru organizarea imprimării pe coloane);

\v - tab. vertical (introduce mai multe rânduri libere pe verticală);

\r - *carriage return* (deplasarea carului de imprimare la începutul rândului curent);

\f - *form feed* (salt la pagină nouă).

Setul de caractere neimprimabile depinde de varianta limbajului.

1.2. Cuvinte rezervate

Sunt cuvinte în limba engleză cu semnificaţie predefinită. Ele nu pot fi utilizate ca identificatori sau cu altă semnificaţie decât cea standard.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned

continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Cuvintele rezervate (sau cuvintele cheie) sunt utilizate în construirea instrucțiunilor, declararea tipurilor de date sau drept comenzi; în anumite situații, pot fi redefinite (înlocuite cu alte cuvinte).

Cuvintele rezervate se scriu cu litere mici.

1.3. Identificatori

Sunt *nume* date funcțiilor, variabilelor, constantelor, în vederea utilizării lor în programe.

Un *identificator* este o succesiune de litere şi cifre, primul caracter fiind obligatoriu o literă. Se pot utiliza literele mari şi literele mici din setul de caractere, precum şi caracterul subliniere _ considerat literă.

Numărul de caractere din componența unui identificator nu este limitat dar în mod implicit, numai primele 32 de caractere sunt luate în considerație. Aceasta înseamnă că doi identificatori sunt diferiți, numai dacă diferă în primele 32 de caractere.

Standardul ANSI garantează 6 caractere / identificator.

Mediile de programare Turbo C şi Turbo C++ permit utilizatorului modificarea limitei de 32.

În limbajul C, literele mari diferă de cele mici, aşadar identificatorii: aria_trg, Aria_trg, ARIA_TRG, diferă între ei.

Se recomandă ca identificatorii să fie *nume sugestive* pentru rolul obiectelor pe care le definesc. De exemplu, funcția care calculează o sumă se va numi sum, cea care calculează factorialul, fact etc.

În mod tradițional, identificatorii de variabile se scriu cu litere mici, constantele şi macrodefinițiile cu litere mari etc.

Exemple:

Identificatori corecți	**Identificatori incorecți**	**Observații**
d1, x1, x2, y11, alfa, beta	$contor	$ - caracter interzis
contor_1, data_16	struct	cuvânt cheie
delta_ec_2, tabel_nume	tab nr.2	conține un spațiu şi punct
TipElementCurent	2abs, 5xx,	încep cu o cifră

Observații:

- Cuvintele cheie nu pot fi utilizate ca identificatori;

- Pentru identificatori se utilizează numai setul de caractere C; ca urmare, punctul, virgula, semnele de operații etc., nu se utilizează în construirea identificatorilor;
- Identificatorii nu trebuie să conțină spațiu liber;
- Pentru un identificator format din mai multe cuvinte, fie se utilizează liniuța de subliniere, fie se inserează litere mari pentru separarea cuvintelor.

1.4. Comentarii

Sunt texte explicative plasate în textul programului, pentru a facilita înțelegerea lui şi pentru a marca principalele acțiuni.

Comentariile nu sunt luate în considerație de către compilator şi ca urmare trebuie delimitate prin caractere speciale. Se delimitează prin /* începutul de comentariu şi prin */ sfârşitul de comentariu. Exemplu:

/* Acesta este un comentariu */

Dacă se omite marcarea sfârşitului de comentariu, textul programului (aflat după comentariu) se consideră ca făcând parte din acesta!

În limbajul C++ s-a mai introdus o convenție pentru a insera comentarii şi anume succesiunea :

//

marchează început de comentariu.

Comentariul delimitat prin // se termină pe acelaşi rând pe care se află şi începutul lui, fără să mai fie necesară marcarea sfârşitului.

Observații:

- Comentariile pot fi introduse oriunde în programul sursă;
- Comentariile pot fi scrise pe una sau mai multe linii de text;
- Nu este permisă introducerea unui comentariu în interiorul altui comentariu.

În general, se recomandă introducerea de comentarii după antetul unei funcții, care să precizeze:

- Acțiunea sau acțiunile realizate de funcție;
- Tipul datelor de intrare şi ieşire;
- Algoritmii realizați de funcție, dacă sunt mai complicați;
- Limite impuse datelor de intrare, dacă este cazul.

1.5. Structura unui program

Un program conține una sau mai multe *funcții*. Dintre acestea, una singură este *funcția principală*.

Fiecare funcție are un nume. Numele funcției principale este ***main***. Celelalte funcții au nume date de utilizator sau au nume predefinit, dacă sunt funcții standard, de bibliotecă.

Programul se păstrează într-un fişier sau în mai multe fişiere. Acestea au extensia .c pentru limbajul C şi .cpp pentru C++.

Un fişier care conține textul programului sau o parte a acestuia se numeşte *fişier sursă*. Prin compilarea unui fişier sursă, compilatorul generează un *fişier obiect*, cu extensia .obj.

Fişierele sursă care intră în compunerea unui program pot fi compilate separat sau împreună. Fişierele obiect, corespunzătoare unui program, pot fi reunite într-un program executabil, prin editarea de legături (*link* - editare), rezultând un *fişier executabil*, cu extensia .exe .

Execuția unui program începe cu prima instrucțiune a funcției *main*.

Dacă funcția *main*( ) face apel la alte funcții, acestea vor intra în acțiune când vor fi chemate de *main*( ).

Exemplu:

```
# include <stdio.h>
void main(void)
{
  printf("Cel mai simplu program C \n");
}
```

- Prima linie este o *directivă* pentru compilator, pentru a include în textul sursă fişierul cu numele *stdio.h* (*st**a**ndar**d** **i**nput **o**utput*) aflat într-un director predefinit. Directivele de compilator au marcajul #.
- A doua linie este *antetul* funcției *main*() - funcția principală. Cuvântul *void* arată că funcția este fără tip (nu furnizează valori) iar *void* dintre paranteze arată că funcția este fără parametri.
- Corpul funcției este delimitat de paranteze acolade { }. Funcția *main* conține un apel la altă funcție, *printf*() - funcție de bibliotecă. Ea realizează tipărirea sau afişarea unui mesaj, conform unui anumit format (***print f**ormat*). Funcția *printf* este o funcție de intrare/ieşire, care se găseşte în fişierul *header* ***stdio.h***; de aceea acest fişier trebuie inclus în toate programele în care se fac operații de citire / scriere.

În limbajul C, nu există instrucţiuni de intrare / ieşire (pentru citire şi scriere date) ca de exemplu *read*, *write* din Pascal sau Fortran. Acest fapt se justifică prin dependența instrucţiunilor de intrare/ieşre de sistemul de operare. Autorii limbajului C au optat pentru realizarea operațiilor de intrare / ieşire cu ajutorul funcţiilor de bibliotecă.

Semnul ; este separator de instrucţiuni (ca în Pascal).

Funcţia *printf*() are un singur parametru = şir de caractere, delimitat de ghilimele ". . . ". Ultimele caractere sunt de tip neimprimabil şi determină trecerea la linie nouă, după imprimarea textului.

1.5.1. Structura unei funcții

În programul sursă, o funcţie are două părți: *antetul* şi *corpul funcției*. O funcţie poate avea un număr de parametri (variabile) dar o singură valoare - *valoarea funcției*. Antetul conține informații despre tipul valorii funcţiei, numărul şi tipul parametrilor şi conține identificatorul (numele) funcţiei.

Structura unei funcții:

```
<tip>  <nume> (<lista declarațiilor parametrilor formali>)
{
      <declarații>
      <instrucțiuni>
}
```

Primul rând este antetul; acesta apare obligatoriu înaite de corpul funcţiei dar şi la începutul programului (înainte de main()) situaţie în care se numeşte *prototip*.

În cazul tipurilor standard, <tip> din antet este un cuvânt cheie care defineşte tipul valorii returnate de funcţie.

În C există două categorii de funcţii:

- funcţii cu tip, care returnează o valoare, de tipul <tip>;
- funcţii fără tip, care nu returnează nici o valoare după execuţie, dar efectuează anumite prelucrări de date; pentru acestea se va folosi cuvântul rezervat ***void*** în calitate de <tip>.

O funcţie poate avea zero, unul sau mai mulţi parametri. Lista declarațiilor parametrilor (formali) este vidă când funcţia nu are parametri.

Exemple:

1. void ff_1(void)

```
{
    . . . .
}
```

Funcţia ff_1 este fără tip, deci nu returnează o valoare concretă şi nu are parametri.

2.
```
void ff_2()
{
    . . . .
}
```

Funcţia ff_2 este fără tip şi nu are parametri (variantă de scriere).

3.
```
int ff_3()
{
    . . . .
}
```

Funcţia ff_3 returnează o valoare de tip întreg, după execuţie, dar nu are parametri.

4.
```
float ff_4(int a, int b)
{
    . . . .
}
```

Funcţia ff_4 returnează o valoare de tip real (***float**ing point*) şi are doi parametri de tip întreg (funcţie reală cu două variabile întregi).

În corpul funcţiei pot să apară una sau mai multe instrucţiuni return, care au ca efect revenirea în programul apelant. Dacă nu apare nici o instrucţiune return, revenirea în programul apelant se face după execuţia ultimei instrucţiuni din corpul funcţiei.

În cazul în care funcţia are mai mulţi parametri, declaraţiile de tip pentru parametri se separă prin virgulă.

Parametrii se utilizează pentru a permite transferul de date, către o funcţie, în momentul utilizării ei în program. La construirea unei funcţii, se face abstracţie de valorile concrete. Acestea vor fi prezente numai la execuţia programului.

În momentul compilării, este necesară doar cunoaşterea tipurilor valorilor pe care le vor primi parametrii la execuţie. Aceste declaraţii

de tip sunt indicate în antetul funcției şi vor servi compilatorului să rezerve memoria necesară pentru fiecare parametru.

Parametrii declarați în antetul unei funcții se numesc *formali,* pentru a evidenția faptul că ei nu reprezintă valori concrete, ci numai țin locul acestora, pentru a putea exprima procesul de calcul. Ei se concretizează la execuție prin valori ce rezultă din apelarea funcției, când sunt numiți *parametri efectivi*.

Observații:

- ♦ Dacă nu se specifică tipul unei funcții, se consideră automat că ea va returna o valoare de tip *int*.
- ♦ În limbajul C++, controlul cu privire la tipul valorii unei funcții este mai strict şi ca urmare se recomandă, ca tipul să fie efectiv specificat, în toate cazurile.
- ♦ Pentru funcția principală ***main***, se pot utiliza antetele:

```
int   main()
int   main(void)
void  main()
void  main(void)
      main()
      main(void)
```

Primele două antete arată că funcția returnează o valoare întreagă; de asemenea ultimele două antete arată că se returnează o valoare întreagă, chiar dacă nu se specifică tipul *int* în antet.

Se utilizează foarte frecvent varianta fără tip `main()`.

Funcția *main* poate avea şi parametri.

1.5.2. Declararea variabilelor. Clase de alocare

Obiectele de bază ale limbajului sunt variabilele. Acestea se declară la începutul funcțiilor (variabile locale) sau în exteriorul tuturor funcțiilor (variabile globale), precizându-se clasa de alocare (eventual, vezi cap.5), **tipul**, **numele** şi valorile inițiale (eventual).

Pentru a putea folosi variabile într-un program, acestea trebuie declarate. O declarație de variabile conține obligatoriu un specificator de *tip* (tipul de date) şi o listă de identificatori, de exemplu:

```
int a,b,c;
```

prin care se declară (se rezervă spațiu de memorie) trei variabile de tip `int` (întregi cu semn), numite `a`, `b`, `c`. Dimensiunea unui tip de date depinde de implementarea limbajului şi se exprimă prin numărul de locații de memorie (octeți, un octet = 8 biți) necesare pentru stocarea valorii.

Dimensiunea tipului `int` este 2 sau 4 octeți (16 sau 32 de biți). Un întreg pe 16 biți poate lua valori în domeniul [-32768, 32767].

Alte tipuri uzuale sunt:

char - tipul caracter, dim.=1 octet;
long - tipul întreg lung, dim.=4 octeți;
short - tipul întreg scurt;
float - tipul real, simplă precizie, dim.=4 octeți;
double - tipul real, dublă precizie, dim.=8 octeți;
long double - tipul real, precizie extinsă, dim.=10 octeți.

Variabilele pot fi inițializate cu valori, chiar la declarare. În exemplul de mai jos, variabilele reale x şi z sunt inițializate iar y, nu.

```
float x =10.07 ,  y,  z = 2.0;
```

Programul următor produce tabelarea valorilor unei funcții de două variabile, în domeniul [0, 1]x[0, 1], cu pasul 0.1.

```
# include <stdio.h>
double fm(double, double);
void main(void)
{
double x,  y,  pas=0.1;
for(x=0.0; x<=1.0; x=x+pas)
     for(y=0.0; y<=1.0; y=y+pas)
          printf("x=%lf  y=%lf  f(x,y)=%lf \n", x, y, fm(x, y));
{
double fm(double x, double y)
{
     return (3.0*x*y + 1.0)/(1.0 + x + y + x*y);
}
```

Programul conține două funcții: main() şi fm(). Funcția matematică fm() are doi parametri de tip real, dublă precizie şi anume x şi y iar tipul valorii funcției este tot real, dublă precizie. Numele funcției reprezintă valoarea întoarsă de aceasta, de exemplu, fm(0.0, 1.0)=0.5 este valoarea lui fm când argumentele sale iau valorile x=0 şi y=1.

Instrucțiunea return <expresie>, întoarce valoarea expresiei în programul apelant, această formă fiind obligatorie la funcțiile cu tip.

Declarația

```
double fm(double, double);
```

de la începutul programului este un *prototip* al funcției fm(), care descrie tipul funcției, numele şi tipul fiecărui parametru. În acest mod, funcția fm() este recunoscută în main(), chiar dacă definiția ei se află după main().

În general, este indicat să se scrie prototipurile tuturor funcțiilor, atât ale celor definite în modulul curent de program, cât şi ale celor definite în alte module dar folosite în modulul curent.

Funcțiile de bibliotecă sunt complet definite (inclusiv prototipurile) în fişiere header, deci includerea acestor fişiere, asigură şi prezența prototipurilor. De exemplu, prototipul lui printf() este în fişierul stdio.h.

În funcția princială main() se declară trei variabile, x, y şi pas, ultima fiind inițializată cu valoarea 0.1.

Cele două instrucțiuni for implementează execuția repetată a funcției printf(), pentru 11 valori ale lui x şi 11 valori ale lui y, în total funcția se repetă de 11 x 11 = 121 de ori. Prima acțiune (x = 0.0) este o inițializare şi se execută o singură dată. A doua expresie din for este condiția de repetare: dacă x<=1, execuția va fi reluată pentru valoarea curentă a lui x.
Ultima acțiune (x = x + pas) este actualizarea variabilei x şi se execută la fiecare iterație, după execuția corpului ciclului. Primul ciclu for conține un alt ciclu for, cu variabila y, care se va executa de 11 ori pentru fiecare valoare a lui x. La fiecare execuție a ciclului cu y, se tipăresc valorile x, y, fm(x, y), cu ajutorul funcției printf().

Şirul de caractere din printf() conține *specificatori de format*, anume secvențele %lf, care indică modul de interpretare a valorilor ce se afişază. Ele vor fi interpretate ca valori reale în dublă precizie. Evident, numărul şi tipul specificatorilor de format trebuie să coincidă cu parametrii care urmează după şirul ce descrie formatul. Restul caracterelor din şir vor fi tipărite prin copiere. Secvența \n asigură trecerea la rândul următor după scrierea celor trei valori.

2 Tipuri de date, operatori şi expresii

Tipul scalar	Tipul structurat	Tipul void (fără tip)
♦ caracter (*char*) ♦ întreg (*int*) ♦ real (*float*) ♦ adresă (*pointer*) ♦ enumerativ (*enum*)	♦ tablou ♦ structură ♦ uniune ♦ şir de caractere	♦ void

2.1. Tipuri de bază (scalare)

Pentru utilizarea eficientă a memoriei, limbajul C oferă mai multe tipuri de reprezentare pentru numerele întregi şi reale:

- ♦ pentru întregi: int, short int, long int, cu sau fără semn;
- ♦ pentru reale: float, double, long double.

Fiecare variabilă sau funcție trebuie asociată cu un tip de date înainte de a fi efectiv folosită în program, prin declarații de forma:

```
int c1, k, m;
float x, y, w=21.7;
long double ff_1(double, double);
```

Pe baza acestor declarații, compilatorul rezervă memorie pentru fiecare variabilă, corespunzător tipului său.

2.1.1. Tipul întreg

Este o submulțime a numerelor întregi; se reprezintă pe 1, 2 sau 4 octeți, respectiv pe 8, 16 sau 32 de biți.

Tip	*Lungime (biți)*	*Domeniul valorilor (IBM - PC)*
int	16	-32 768 . . . +32 767
short int	16	-32 768 . . . +32 767
long int	32	-2 147 483 648 . . . +2 147 483 648
unsigned short	16	0 . . . 65 535 (fără semn)
unsigned long	32	0 . . . 4 294 967 295 (fără semn)
signed char	8	-128 . . . +127 (cu semn)

unsigned char	8	0 . . . 255 (fără semn)

Restricțiile impuse de standardul ANSI sunt:

dim (short) ≥ 16 biți ;

dim (int) ≥ 16 biți ;

dim (long) ≥ 32 biți ;

dim (short) ≤ dim (int) ≤ dim (long) ;

Observație: declarația **short int** este echivalentă cu **short** iar **long int** este echivalentă cu **long**, fără să apară confuzii.

Constante de tip întreg

Pot fi scrise în sistemul de numerație zecimal (baza 10), octal (baza 8) sau hexazecimal (baza 16).

O constantă zecimală întreagă este un şir de cifre zecimale, cu prima cifră diferită de zero. Constantele întregi reprezentate pe 16 biți sunt de tip int iar cele reprezentate pe 32 de biți sunt de tip long.

Dacă dorim să reprezentăm o constantă întreagă pe 32 de biți, deşi ea se poate reprezenta pe 16 biți, constanta trebuie să aibă sufixul L sau l.

Constantele întregi sunt, de regulă, cu semn. Dacă dorim să fie fără semn, constanta se termină cu litera U sau u.

Constantele întregi fără semn se utilizează pentru a economisi memorie. Astfel, constantele de tip int din intervalul [32768, 64535] se păstrează în memorie pe 32 de biți, iar constantele de tip unsigned din acelaşi interval se reprezintă pe 16 biți.

Pentru a reprezenta constante întregi de tip long, fără semn, se utilizează sufixele echivalente: ul, lu, LU, UL.

O constantă octală întreagă este o succesiune de cifre octale (0. .7), prima cifră fiind un zero, nesemnificativ. Sufixele L, l, U, u, se pot utiliza cu aceleaşi semnificații ca la constantele zecimale.

O constantă hexazecimală întreagă este o succesiune de cifre hexazecimale (0. .9, A, B, C, D, E, F), sau (0. . 9, a, b, c, d, e, f) primele două caractere fiind 0x sau 0X. Sufixele L, l, U, u, se pot utiliza cu aceleaşi semnificații ca la constantele zecimale.

Exemple:

132 constantă zecimală de tip int;
12345L constantă zecimală de tip long;
0400 constantă octală;
04000L constantă octală de tip long;
0xabbf constantă hexazecimală;

0X2ACFFBL constantă hexazecimală de tip long;

Indicarea tipului este utilă când se folosesc funcții de bibliotecă având argumente de tip long sau fără semn; tipul constantei trebuie să fie compatibil cu tipul predefinit al argumentului.

Operațiile de citire şi scriere a valorilor de tip întreg, cu funcțiile de bibliotecă printf(. . .) şi scanf(. . .), necesită specificatori de format corespunzători.

Tipul	În zecimal	În octal	În hexazecimal
int	%d	%o	%x
long	%ld	%lo	%lx
short	%hd	%ho	%hx
unsigned	%du	%ou	%xu

Programul următor citeşte o valoare de tip zecimal şi scrie pe ecran aceeaşi valoare în octal şi în hexazecimal:

```
# include <stdio.h>
void main()
{
      int val;
      printf("Introduceți o constantă de tip întreg:");
      scanf("%d",&val);
      printf("Valoarea %d in baza 8 este: %o\n", val, val);
      printf("Valoarea %d in baza 16 este: %x\n", val, val);
}
```

Tipărirea unei valori cu semn, cu formatul %d poate duce la valori graşite, dacă valoarea nu se încadrează în domeniul int:

```
# include <stdio.h>
main()
{
      unsigned short n1=40000, n2=400;
      printf("n1 si n2 considerate ca intregi fara semn:\n");
      printf("n1=%hu n2=%hu \n", n1, n2);
      printf("n1 si n2 considerate ca intregi cu semn:\n");
      printf("n1=%hd n2=%hd \n", n1, n2);
}
```

Programul are ca efect pe ecran:

```
n1 si n2 considerate ca intregi fara semn:
n1=40000      n2=400
n1 si n2 considerate ca intregi cu semn:
n1=-25536     n2=400
```

2.1.2. Tipul caracter

Este asociat cu tipul întreg fără semn, pe 8 biți, deoarece fiecare caracter are un număr de ordine, în codul ASCII, cuprins între 0 şi 255.

O constantă de tip caracter imprimabil se reprezintă prin caracterul respectiv, cuprins între semne apostrof: 'a', 'A', 'b', 'B', '7' etc.

Limbajul C permite efectuarea de operații aritmetice asupra unor valori de tip caracter şi se pot da valori numerice unor variabile de acest tip. Exemplu:

```
char c, car;     //se definesc doua variabile caracter
. . . . . . .
car ='a';        // car = 97, codul lui 'a'
c   = 97         // c = 97, deci c = car
```

2.1.3. Tipul real

Este o submulțime a numerelor reale, mai precis, o submulțime a numerelor raționale. O constantă de tip real se compune din:

- ♦ o parte întreagă, număr întreg cu semn (poate lipsi);
- ♦ o parte fracționară, şir de cifre zecimale cu punct în față;
- ♦ un exponent, număr întreg cu semn (poate lipsi).

Partea fracționară poate lipsi numai când este prezentă partea întreagă, menținând punctul de separare: 2. sau 0. sau -314.

Exponentul este un întreg cu sau fără semn, precedat de litera e sau E şi reprezintă o putere a lui 10.

În mod implicit, constantele reale se reprezintă în dublă precizie, deci pe 64 de biți. Pentru economie de memorie, se poate opta pentru reprezentare pe 32 de biți, prin adăugarea literei f sau F la sfârşit.

Pentru reprezentare pe 80 de biți, se adaugă la sfârşit litara L sau l.

Exemple:

Constantă reală	*Valoare*
123.	123,0
-765.77	-765,77
.255	0,255
69e4	690 000,0
.23E-2	0, 0023
222.7e2	22270
3377.5678e-5	0,033775678

Variantele utilizate pentru tipurile reale sunt descrise în tabelul următor.

Tip	*Lungime (biți)*	*Domeniul valorilor absolute*
float	32	$3,4\times10^{-38}\ldots3,4\times10^{38}$
double	64	$1,7\times10^{-308}\ldots1,7\times10^{308}$
long double	80	$3,4\times10^{-4932}\ldots3,4\times10^{4932}$

Fişierele header *limits.h* şi *float.h* conțin constante care definesc implementarea tipurilor de bază.

Variabilele de tip real se declară prin specificarea tipului şi listei de identificatori:

```
float        val_1, val_f, zz, t, w;
double       xx_1, y;
long double      epsilon, pass_1;
```

Observații:

- Nu trebuie utilizate inutil variabile şi constante de tip double sau long double, deoarece determină creşterea timpului de calcul şi ocupă inutil memorie suplimentară;
- Descriptorii de format la tipărire cu printf( ) sunt: %f sau %lf pentru scriere normală, şi %e sau %E, pentru scriere cu exponent.

Exemplu:

```
# include <stdio.h>
void main()
{
float nr_real=0.0057;
      printf("nr_real=%f \n", nr_real);
      printf("nr_real=%6.4f \n", nr_real);
      printf("nr_real=%e \n", nr_real);
}
```

Rezultatul pe ecran este:

```
nr_real=0.005700
nr_real=0.0057
nr_real=5.70000e-03
```

Codul 6.4 asociat cu %f realizează controlul formatului de scriere prin indicarea spațiului total (6 caract.) şi a părții fracționare (4 caract.).

Valorile de tip float sunt convertite automat la double de către funcția printf( ), deoarece reprezentarea implicită a constantelor reale este pe 64 de biți, deci double.

Valorile de tip long double necesită specificatori de format %Lf sau % lf.

Constantele simbolice se definesc prin directiva #define, de exemplu:

```
# define MAX  4500        //constanta simbolica de tip intreg
# define X_REF 37.99f    //constanta simbolica de tip real
```

Constantele simbolice sunt substituite la compilare cu valoarile prin care au fost definite. Se recomandă scrierea lor cu litere mari. Valoarea unei constante simbolice nu poate fi modificată la execuție.

2.1.4. Constante de tip şir de caractere

Un şir de caractere este o succesiune de caractere, eventual vidă, încadrată de ghilimele ". . . . " , de exmplu:

```
"Limbajul  C++ este un limbaj profesional"
```

În limbajul C, nu există o declaraţie specială, ca de exemplu, *string* în Pascal, pentru şiruri de caractere. Acestea sunt considerate tablouri de caractere, fiecare element al tabloului fiind un caracter, care ocupă în memorie un octet (8 biți).

Declararea datelor de tip şir de caractere, se face astfel:

```
char  num_pr[20];
```

unde num_pr reprezintă un identificator (nume şi prenume) iar 20 este lungimea şirului, adică şirul are maxim 20 de caractere. Compilatorul adaugă un caracter special, de sfârşit de şir, neimprimabil, '\0'.

Şirurile de caractere pot fi scrise şi citite cu funcțiile printf( ) şi scanf( ), specificatorul de format fiind %s.

```
# include <stdio.h>
void main()
{
char prenume[20];
     printf("Care este prenumele tau? \n");
     scanf("%s" , &prenume);
     printf("Buna ziua %s \n", prenume);
}
```

2.1.5. 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 şi 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 *adresă* se exprimă prin tipul *pointer*. În limba română, se traduce prin *referință, reper, localizator, indicator de adresă*.

	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 şi 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 şi din tabelul de mai sus, rezultă px = 0026;

Analog, py = 0028, pz = 0030.

Observaţii:

- 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     ; // greşit, pentru că a2 se asociază cu variabile int
```

- Operaţia inversă, de aflare a valorii din memorie, de la o anumită adresă, se realizează cu operatorul * pus în faţa 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;
```

Aşadar, operatorul * are efect invers faţă de operatorul & şi 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 şi semnalizate de compilator ca erori de sintaxă.

O expresie de tipul *<adresa> poate să apară şi 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 }
```

2.1.6. Tipul enumerativ

Se utilizează când o variabilă trebuie să ia un număr redus de valori, de natură nenumerică. Sintaxa conține cuvântul cheie *enum* :

enum <identificator> {lista de valori} ;

unde identificatorul este numele *tipului de date*.

Exemple:

```
enum  culoare {ALB, ROSU, GALBEN};
enum  culoare fanion_1, fanion_2;
```

A doua declarație definește două variabile de tip "culoare", fanion_1 şi fanion_2.

Se pot înlocui cele două declarații, prin una singură:

```
enum  culoare {ALB, ROSU, GALBEN}  fanion_1, fanion_2;
```

Fiecare constantă din listă va primi o valoare întreagă dată de poziția sa în listă, 0, 1, 2, . . . , n. În exemplul de mai sus, ALB=0, ROSU=1, GALBEN=2.

Tipul enumerativ nu este, de fapt, un tip nou; este o nouă alocare de nume pentru tipul întreg.

Este permisă asocierea constantelor simbolice cu valori întregi, altele decât în asocierea implicită:

```
enum  cod_err  {ER1=-2, ER2, ER3=3, ER4, ER5}  ;
```

Asocierea de valori se va face astfel:

ER1=-2, ER2=-1, ER3=3, ER4=4, ER5=5.

Se observă că pentru constantele neasociate cu valori la declarare, se dau valori succesive în ordine crescătoare.

Tipul enumerativ produce economie de memorie, pentru că se utilizează un singur octet, pentru o valoare din listă.

2.2. Operatori şi expresii

Un **operator** este o **comandă** ce acţionează asupra unor obiecte (termeni) ce se numesc generic **operanzi;** fiecare operator este definit printr-un simbol special sau grup de simboluri. Operatorul determină executarea unei anumite prelucrări (operaţii) asupra operanzilor.

O formulă în care există operanzi şi operatori este numită **expresie**.

Un operand poate fi: constantă, variabilă simplă, nume de funcţie, nume de tablou, nume de structură, expresie închisă între paranteze rotunde. Orice operand are un *tip* şi o *valoare concretă* în momentul în care se efectuează operaţia.

Operatorii se pot clasifica, după numărul de operanzi pe care îi combină, în:

- ♦ unari (acţionează asupra unui singur operand);
- ♦ binari (acţionează asupra a doi operanzi);
- ♦ ternar (acţionează asupra a trei operanzi) - există doar unul în C.

După tipul operaţiei ce se execută, operatorii sunt de mai multe categorii: aritmetici, logici, relaţionali, condiţionali, de atribuire.

La evaluarea unei expresii, se ţine seama de categoria de *prioritate* a fiecărui operator, de *asociativitatea* operatorilor de aceeaşi prioritate şi de regula *conversiilor* implicite. În tabelul următor sunt prezentaţi sintetic operatorii din limbajul C - standard, cu principalele lor caracteristici.

Niv.	Categorie	Operatori	Semnificaţie	Asociere
1	Selectori	( ) [ ] -> .	Expresie, apel de funcţie Expresie cu indici Selectori la structuri	stânga - dreapta
2	Op. unari	! ~ + - ++ -- & * sizeof (tip)	Negare logică Negare bit cu bit Plus, minus (semne aritmetice) Incrementare/ Decrementare Luarea adresei Indirectare Dimensiune operand (în octeţi) Conversie explicită de tip	**dreapta - stânga**
3	Op. aritm._1	* / %	Înmulţire, împărţire, rest	st - dreapta
4	Op. aritm._2	+ -	Adunare, scădere	st - dreapta
5	Deplasări	<< >>	Deplasare stânga, dreapta	st - dreapta
6	Relaţionali	< <= > >=	mai mic/egal, mai mare/egal	st - dreapta
7	Egalitate	== !=	Egal logic, diferit de	st - dreapta

<table>
<tr><td>8</td><td rowspan="5">Operatori logici</td><td>&</td><td>Şi logic bit cu bit</td><td>st - dreapta</td></tr>
<tr><td>9</td><td>^</td><td>Sau exclusiv bit cu bit</td><td>st - dreapta</td></tr>
<tr><td>10</td><td>|</td><td>Sau logic bit cu bit</td><td>st - dreapta</td></tr>
<tr><td>11</td><td>&&</td><td>Şi logic (între expresii)</td><td>st - dreapta</td></tr>
<tr><td>12</td><td>||</td><td>Sau logic (între expresii)</td><td>st - dreapta</td></tr>
<tr><td>13</td><td>Op. condiţional</td><td>? :</td><td>(expresie)? expr_1 : expr_2</td><td>dreapta-st.</td></tr>
<tr><td>14</td><td>Op. de atribuire</td><td>=
*= /= %= += -=
&= ^= |=
<<= >>=</td><td>Atribuire simplă
Atribuire produs, cât, rest, sumă, dif.
Atribuire şi, sau excl., sau bit cu bit
Atribuire şi deplasare stg., dreapta</td><td>dreapta - stânga</td></tr>
<tr><td>15</td><td>Virgula</td><td>,</td><td>Evaluează op1, op2. Valoarea expr. este op2.</td><td>st - dreapta</td></tr>
</table>

Nivelul de prioritate arată cum se evaluează expresiile care conţin doi sau mai mulţi operatori, când nu există paranteze care să impună o anumită ordine a operaţiilor.

Operatorii de nivel 1 au prioritate maximă; în tabel operatorii sunt prezentaţi în ordinea descrescătoare a priorităţii. Operatorii cu acelaşi nivel de prioritate se execută conform regulii de asociere: stânga - dreapta sau dreapta - stânga. De exemplu, în expresia:

```
x= *p++ ;
```

apar doi operatori: cel de indirectare (dereferenţiere) şi cel de post - incrementare; se pune întrebarea: ce se incrementează, p sau *p ?, adică ce operator acţionează primul ?. Din tabel, se vede că atât ++ cât şi * au acelaşi nivel de prioritate, dar ordinea de asociere este dreapta-stânga, aşadar, ++ şi apoi *. Se incrementează aşadar p nu *p. Dacă dorim să incrementăm *p, trebuie scris:

```
x=(*p)++;
```

Asociativitatea operatorilor arată cum se evaluează expresiile, în care, apare acelaşi operator de mai multe ori. De exemplu, în expresia:

```
c=x*y*z ;
```

apar două înmulţiri. Expresia se poate evalua ca (x*y)*z sau x*(y*z). Operatorul * are asociativitatea de la stânga la dreapta, adică expresia se interpretează

```
c=(x*y)*z;
```

Operatorul de atribuire are asociativitatea de la dreapta la stânga, deci expresia:

```
a=b=c=d;
```

se evaluează ca şi cum ar fi scrisă:

```
c=d;
b=c;
a=b;
```

Asociativitatea poate fi modificată prin paranteze rotunde, ordinea de evaluare fiind "*mai întâi parantezele interioare*".

Operatorii unari (nivelul 2), cel condiţional (nivelul 13) şi cei de atribuire (nivelul 14) se asociază de la dreapta la stânga. Toţi ceilalţi operatori se asociază de la stânga la dreapta.

2.2.1. Operatori aritmetici (+, -, *, /, %)

- Se pot aplica numai la tipurile întregi (char, short, int, long).
- Operatorii + şi - pot fi şi unari, adică se poate scrie x=+2; y=-7;
- Asociativitatea este de la dreapta la stânga pentru cei unari şi de la stânga la dreapta pentru cei binari.
- Dacă operanzii au tipuri diferite, se aplică conversii implicite de tip, unul din operanzi fiind convertit la tipul celuilalt, rezultatul fiind de tipul comun, după conversie.

2.2.2. Operatori relaţionali şi de egalitate (< , <=, >, >=, ==, !=)

Toţi sunt binari, iar expresiile formate se evaluează la 0 sau la 1, corespunzător valorilor logice *fals* şi *adevărat*. În limbajul C nu există constante logice de tip `TRUE` şi `FALSE`, întregii 1 şi 0 fiind asociaţi cu cele două valori (`1 = TRUE, 0=FALSE`). Mai general, orice valoare numerică diferită de 0 este interpretată logic ca "adevărat". Operatorul == produce valoarea 1 când cei doi operanzi sunt egali şi 0 când sunt diferiţi iar operatorul != , invers.

O eroare frecventă este confuzia dintre = (care este atribuire, adică se transferă valoarea din dreapta, către variabila din stânga) şi == , adică testarea egalităţii: x==y înseamnă *este x egal cu y ?* (Da? sau Nu?). Adesea, eroarea nu este semnalată de compilator, de exemplu, dacă în loc de :

```
if (x==2) . . .
```

se scrie

```
if (x=2) . . .
```

se atribuie lui x valoarea 2, care fiind diferită de 0 se interpretează ca fiind "adevărat" şi se execută istrucţiunea din ramura if, indiferent de valoarea iniţială a lui x (care se pierde). Astfel de erori, care nu pot fi detectate de compilator sunt greu de remediat.

Asociativitatea este de la dreapta la stânga, iar prioritatea operatorilor == şi != este mai slabă decât a celorlalţi operatori relaţionali.

O expresie de forma `a<b == x>y` este echivalentă cu `(a<b) == (x>y)`.

2.2.3. Operatori logici (&& || !)

Operanzii sunt evaluaţi cu *adevărat* (diferit de 0) sau *fals* (egal cu 0) iar valoarea expresiei este totdeauna 0 sau 1. Operatorul ! (negaţie logică) transformă o valoare 0 în 1 şi o valoare diferită de 0 (*adevărat*) în 0 (*fals*).

Astfel, !0=1, !1=0, !7=0, !!7=1, !(2>3)=1.

La operatorii && (şi logic) şi || (sau logic), se garantează ordinea de evaluare a operanzilor după următoarele reguli:

- la &&, se evaluează întâi primul operand; dacă acesta este diferit de 0, se evaluează şi al doilea operand iar valoarea expresiei este dată de al doilea; dacă primul operand este 0, nu se mai evaluează al doilea, deoarece orice valoare ar avea, în final se obţine 0.
- la || , se evaluează, de asemenea, primul operand; dacă valoarea logică este 1 (adevărat) nu se mai evaluează al doilea, deoarece orice valoare ar avea, în final se obţine 1 (adevărat).

Acest mod de evaluare este eficient, deoarece se reduce la minimum posibil numărul de operaţii, ceea ce înseamnă economie de timp.

Prioritatea operatorului unar ! este mai mare decât a tuturor operatorilor aritmetici, logici şi relaţionali iar a operatorilor && şi || este inferioară operatorilor relaţionali, adică o expresie de forma:

```
a>b&&c==d&&e!=f
```

se evaluează ca şi cum ar fi scrisă:

```
(a>b)&&(c==d)&&(e!=f)
```

Prioritatea operatorului || (sau) este mai scăzută decât a lui && (şi) iar asociativitatea este de la stânga la dreapta.

2.2.4. Opratori de incrementare / decrementare (++ --)

Au ca efect creşterea valorii cu 1 (incrementare) sau scăderea valorii cu 1 (decrementare). Fiecare operator are două forme: forma prefixată, adică ++x, --y, sau forma postfixată x++ , y--. Valoarea expresiei x++ este valoarea lui **x înainte** de incrementare iar valoarea expresiei ++x este valoarea lui **x după** incrementare. Aceşti operatori nu se aplică unor expresii aritmetice şi nici constantelor. Exemple:

```
n=7; m=n++; //se atribuie lui m valoarea 7
n=7; m=++n; //se atribuie lui m valoarea 8
```

ambele variante modifică în final valoarea lui n în n+1.

Operatorii de incrementare / decrementare contribuie la scrierea unor programe mai eficiente şi mai scurte.

2.2.5. Operatorul unar *sizeof*

Returnează dimensiunea în octeţi (numărul de octeţi ocupaţi) a unei variabile, a unui tip de date, sau a unui tip structurat. De exemplu, `sizeof(int)` dă numărul de octeţi pe care este memorat un întreg iar dacă `tab` este un tablou, expresia:

```
sizeof(tab) / sizeof(tab[0])
```

dă numărul de elemente ale tabloului, oricare ar fi tipul de date, pentru că se împarte numărul total de octeţi ocupat de tablou la numărul de octeţi ocupaţi de un element.

În diverse implementări mai vechi, **sizeof** întorcea tipuri diferite de întregi, de exemplu **int**, **unsigned int** sau **unsigned long.** Standardul ANSI introduce un tip predefinit, numit **size_t**, garantând că **sizeof** întoarce acest tip de întreg. Astfel, programele care folosesc acest tip, capătă o portabilitate sporită. Tipul **size_t** este definit în fişierul header **stddef.h**, ca şi în **stdio.h** şi în alte fişiere header uzuale.

2.2.6. Operatori de atribuire (= *= += /= %= -= &= ^= |= <<= >>=)

În limbajul C, pe lângă operatorul de atribuire normală = (*se atribuie valoarea expresiei din dreapta, variabilei din partea stângă a semnului* =), există şi operatori de atribuire combinată cu o operaţie, pentru a face textul mai concis, dat fiind că foarte frecvent atribuirea este însoţită de o operaţie. Astfel, avem scrieri echivalente:

```
x=x+3;        echivalent cu    x += 3;
y=y*6.5;      echivalent cu    y *= 6.5;
u=u/(v-2);    echivalent cu    u /= v-2;
a=a % b ;     echivalent cu    a %= b;
```

Tipul expresiilor formate cu operatorii de atribuire este tipul expresiei din stânga. Valoarea unei expresii de atribuire de forma a=b este valoarea membrului drept, după o eventuală conversie la tipul membrului stâng. Asociativitatea este de la dreapta la stânga; de exemplu, în atribirea multiplă de mai jos:

```
x = y = z = 0 ;
```

ordinea operațiilor este:

```
x = (y = (z = 0));
```

faptul că z = 0 este o expresie cu valoarea 0 (a membrului drept), face posibilă atribuirea următoare y = 0 şi apoi x = 0; aşadar, în final toate cele trei variabile primesc valoarea 0.

Evaluarea expresiilor cu operatori compuşi de atribuire se face pe baza definiției echivalente a acestora. De exemplu, expresia:

```
a = b += c + 1;
```

se evaluează (de la dreapta la stânga) ca:

```
a = (b = b + (c+1));
```

2.2.7. Operatorul condițional (? :)

În situația în care unei variabile trebuie să i se atribuie o valoare sau alta în funcție de o condiție logică, se poate folosi instrucțiunea if dar în C există, ca variantă, operatorul condițional (? :), cu sintaxa:

```
(expr) ? expr_1: expr_2;
```

unde (expr) este evaluată logic; dacă este adevărată (diferită de 0), se evaluează expr_1 care devine valoare finală pentru expresia condițională; dacă (expr) este falsă (egală cu 0), atunci este evaluată expr_2, care devine valoare finală. Se garantează că una şi numai una dintre expr_1 şi expr_2 este evaluată.

Exemplu: tipărirea unui vector numeric, cu un anumit număr de elemente pe linie (de exemplu, 10), separate prin blank:

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

Se tipăreşte întregul a[i] şi apoi un singur caracter (cu formatul %c), acesta fiind '\n' sau ' ' după cum s-a tipărit tot al 10-lea întreg sau ultimul. Orice altă variantă de program ar fi dus la mai multe instrucțiuni.

2.2.8. Operatorul virgulă (,)

O expresie în care apare operatorul virgulă are forma:

```
expr_1, expr_2
```

Evaluarea se face în ordinea (garantată) expr_1 şi apoi expr_2. Valoarea finală este dată de expr_2. Se aplică regulile privind conversia de tip (dacă este cazul).

O asemenea construcție este utilă acolo unde sintaxa impune o singură expresie. Prin folosirea operatorului virgulă, putem realiza mai multe acțiuni.

2.3. Tipuri structurate: tablouri şi structuri

2.3.1. Tablouri

Un tablou este o colecție de date de acelaşi tip - numit *tip de bază* - la care accesul se realizează prin intermediul indicilor. Elementele unui tablou ocupă, în memorie, locații succesive iar ordinea în care sunt memorate elementele, este dată de indici.

O caracteristică importantă a unui tablou este dimensiunea, adică numărul de indici. Astfel, există tablouri:

- unidimensionale - elementele au un singur indice a[0], a[1], . . .;
- bidimensionale - elementele au doi indici b[0][0], b[0][1], . . . ;
- multidimensionale - elementele au mai mulți indici.

Declarația de tablou, în forma cea mai simplă, conține *tipul* comun al elementelor sale, *numele* tabloului şi *limitele* superioare pentru fiecare indice (valorile maxime ale fiecărui indice):

```
<tip>  <nume> [dim_1][dim_2]. . . [dim_n];
```

- **tip** este cuvânt cheie pentru tipurile predefinite;
- **nume** este un identificator pentru tablou;
- **dim_k** este limita superioară pentru indicele k ; acesta va lua valorile 0, 1, 2, . . . k-1, în total k valori.

Limitele `dim_k` sunt expresii constante, adică expresii care pot fi evaluate la compilare. Numele unui tablou este un simbol care are ca valoare adresa primului element din tablou.

La întâlnirea unei declarații de tablou, compilatorul alocă o zonă de memorie pentru păstrarea valorilor elementelor sale. Numele tabloului respectiv poate fi utilizat în diferite expresii, în program, valoarea lui fiind chiar adresa de început a zonei de memorie care i-a fost alocată.

Exemple:

```
int a[20];
char sir[80];
float x[10][50];
```

Declarațiile de mai sus arată că **a** este un tablou de 20 de variabile de tip întreg, **sir** este un tablou de 80 de caractere iar **x** este un tablou bidimensional care conține 10 x 50 = 500 elemente de tip real.

Elementele tabloului pot fi accesate prin numele tabloului urmat de o expresie întreagă pusă între paranteze drepte. Expresia trebuie să ia valori în domeniul 0, . . . , dim-1. De exemplu, referiri corecte la elementele tabloului **a** de mai sus sunt: `a[0], a[1], . . . , a[19]` iar la elementele tabloului **x** sunt: `x[0][0], x[0][1],..., x[9][0], x[9][1],..., x[9][49]`.

Prelucrările de tablori se implementează de obicei prin cicluri for, deoarece numărul de repetări este cunoscut, fiind dependent de valorile indicilor.

Tablourile pot fi declarate în interiorul funcțiilor, caz în care sunt implicit în clasa **auto**, sau în exteriorul funcțiilor, caz în care sunt implicit în clasa **extern**.

Tablourile nu pot fi declarate în clasa **register**.

Tablourile pot fi inițializate cu valori. Inițializarea respectă regulile care derivă din clasa de alocare: un tablou în clasa **auto** (inițializat explicit) va fi inițializat la fiecare intrare în funcția respectivă iar un tablou în clasa **static** sau **extern** (inițializat explicit) se inițializează o singură dată, la încărcarea programului în memorie. Tablorile în clasa **static** sau **extern**, care nu sunt inițializate explicit, se inițializează automat cu 0 iar cele în clasa **auto** vor primi valori aleatoare.

Inițializarea se face cu semnul egal şi o listă de valori:

```
int a[4]={2, 4, 6, 8};
char s[4]={'x', 'y', 'z', '\0'};
```

Dacă lista de valori conține mai puține elemente decât dimensiunea declarată a tabloului, celelalte valori se inițializează cu 0, indiferent de clasa de alocare. De exemplu, după declarația:

```
int a[7]={1, 5, 11, -3};
```

elementele a[4], a[5], a[6] vor conține valoarea 0.

În cazul unui tablou inițializat, se poate omite dimensiunea, ea fiind calculată de compilator, după numărul de elemente din listă.

De exemplu, declarația :

```
float x[ ]={1.0, 2.1, 3.2};
```

arată că x este un tablou cu trei elemente reale, specificate în listă.

Un caz special îl reprezintă tablourile de caractere, care pot fi inițializate şi cu şiruri constante de caractere. Astfel, declarația:

```
char s[ ]="abcdef" ;
```

este perfect echivalentă cu

```
char s[ ]={'a','b','c','d','e','f','\0'};
```

observând prezența caracterului special '\0' (octetul nul, care are rol de terminator de şir).

Transmiterea tablourilor cu o dimensiune la funcții se face prin declararea parametrului formal al funcției ca fiind tablou, ceea ce se realizează punând paranteze drepte. Nu este necesară prezența explicită a dimensiunii tabloului între paranteze, deoarece ceea ce se transmite efectiv este adresa de început a tabloului.

Dacă totuşi, este necesară dimensiunea tabloului, aceasta se poate preciza într-un parametru separat. În exemplul de mai jos, este prezentată o funcție care calculează produsul scalar a doi vectori de lungime n, reprezentați prin două tablouri de numere reale:

```
float prod_scalar(float x[], float y[], int n)
{
    float prod=0.0;
    int i;
    for (i=0; i<n; i++)
        prod +=x[i]*y[i];
    return prod;
}
```

Dacă sunt declarate două asemenea tablouri, de 3 elemente:

```
float a[3]={-1.0, 2.0, 3.14};
float b[3]={0.9, -1.02, 0.0};
float ab;
```

atunci produsul scalar al vectorilor a şi b se poate obține prin:

```
ab=prod_scalar(a, b, 3);
```

Tablourile de caractere se prelucrează diferit, deoarece ele nu au de obicei specificată dimensiunea; aceasta se deduce din prezența terminatorului '\0' care este ultimul caracter din şir.

O funcție care calculează numărul de caractere dintr-un asemenea tablou este:

```
int nr_car(char s[])
{
     int i;
     for (i=0; s[i] != '\0'; i++);
     return i;
}
```

Se observă că terminatorul '\0' nu se consideră caracter util, deci nu este inclus în rezultat. Funcția de mai sus este echivalentă cu funcția de bibliotecă **strlen()**.

Pentru tablouri uni- sau multi-dimensionale, alocarea memoriei se face după reguli stricte, cuprinse în **funcția de alocare a memoriei**.

Modul de alocare a memoriei se poate ține minte prin regula sintetică: *"tabloul este alocat la adrese succesive de memorie, ultimul indice având variația cea mai rapidă"*. De exemplu, considerând declarația:

char c[2][2][3]

imaginea în memorie a elementelor tabloului c[] va fi:

```
c[0][0][0]
c[0][0][1]
c[0][0][2]
c[0][1][0]
c[0][1][1]
c[0][1][2]
c[1][0][0]
c[1][0][1]
c[1][0][2]
c[1][1][0]
c[1][1][1]
c[1][1][2]
```

Adresa lui c[1][0][1] se calculează astfel:

adr(c[1][0][1]) = baza(c) + 1*2*3 + 0*3 + 1 = 7,

elementul c[1][0][1] găsindu-se într-adevăr la 7 octeți distanță de c[0][0][0]. Adesa se calculează astfel:

baza(tab) + nr.octeți/element * [indice_1 * dim_2 * dim_3 +
indice_2 * dim_3 +
indice_3].

În cazul tablourilor cu două dimensiuni, regula de alocare este dată de: *"matricele se memorează pe linii"*; într-adevăr, un tablou x[2][3] va fi memorat astfel:

```
x[0][0]
```

```
x[0][1]
x[0][2]
x[1][0]
x[1][1]
x[1][2]
```

În limbajul C, un tablou cu mai multe dimensiuni este tratat ca un tablou cu o dimensiune (şi anume, prima), care are ca elemente tablouri cu restul de dimensiuni; astfel, nu există limitări sintactice ale numărului de dimensiuni. De exemplu tabloul:

```
int a[2][3];
```

este interpretat ca un tablou uni-dimensional cu 2 elemente, fiecare fiind tablou 1-dimensional cu 3 elemente: a[0] este tablou cu 3 elemente, a[1] este tablou cu 3 elemente.

Transmiterea unui tablou cu mai multe dimensiuni unei funcții

În toate situațiile, în care lucrăm cu tablouri cu mai multe dimensiuni, trebuie să ținem seama de funcția de alocare a tabloului şi să stabilim *dacă la compilare se calculează corect adresa elementelor*. Este nevoie de toate dimensiunile, mai puțin prima.

Să considerăm o funcție care tipăreşte elementele unui tablou. Ceea ce se transmite funcției este adresa de început dar sunt necesare şi informațiile pentru calculul adresei, adică dimensiunea a doua. Definim tipul de date DATA şi formatul corespunzător de tipărire:

```
typedef int DATA;
#define FORMAT  "%7d"
```

Este incorectă forma:

```
void matprint(DATA a[][], int m, int n)
{    int i, j;
     for (i=0; i<m; i++) {
          for (j=0; j<n; j++)
               printf(FORMAT, a[i][j]);
          printf("\n");
        }
}
```

deoarece nu se poate calcula adresa lui a[i][j].

În momentul compilării, nu este cunoscută a doua dimensiune a lui **a**, deci apare eroare, chiar la compilare.

Pentru tipărirea unei matrice declarate cu:

```
DATA x[3][4];
```

prima linie din definiția funcției trebuie să fie:

```
void matprint(DATA a[][4], int m, int n )
```

iar apelul va fi:

```
matprint(x, 3, 4);
```

Acum compilatorul "ştie" să calculeze adresa lui a[i][j], dar funcţia astfel construită poate fi folosită numai cu matrice cu 4 coloane. O soluţie mai eficientă este folosirea unei forme echivalente pentru a[i][j], anume aceea care simulează funcţia de alocare memorie, pe baza celei de-a doua dimensiuni, care este n.

```
void matprint(DATA (*a)[], int m, int n)
{
     int i, j;
for (i=0; i<m; i++) {
     for (j=0; j<n; j++)
          printf(FORMAT, ((DATA *)a)[i*n+j];
     printf("\n");
     }
}
```

Cea ce se transmite este adresa de început a tabloului **a**, văzută ca pointer la un tablou cu o dimensiune de tipul DATA (nu mai apar erori la compilare). Conversia explicită de tip ((DATA *)a) face ca **a** să fie văzut ca un pointer la DATA, deci ca un tablou uni-dimensional. Indicele i*n+j face ca să fie accesat corect elementul a[i][j], indiferent de dimensiuni.

În concluzie, *se poate omite numai prima dimensiune a tabloului*, celelalte trebuie cunoscute, pentru a se cunoaşte modul de organizare; dimensiunea omisă se poate transmite ca parametru separat, ea trebuie cunoscută la execuţia funcţiei; programul următor calculează suma elementelor unei matrice.

```
#include <stdio.h>
#include <conio.h>
#define MAX 3
int suma(int tab[][MAX], int r);
void afis(int tab[][MAX],int r);
void main()
{
int mat[3][MAX]={{1,2,3},{4,5,6},{7,8,9}};
//initializare matrice
int rind=3;
afis(mat,rind);
printf("\n Suma este %d",suma(mat,rind));
getch();
}
int suma(int tab[][MAX],int r)
{
int i, j, s=0;
```

```
for(i=0;i<r;i++)
     for(j=0;j<MAX;j++)
       s+=tab[i][j];
        return s;
}
void afis(int tab[][MAX], int r)
{
int i,j;
printf("\n");
for(i=0;i<r;i++)    {
      for(j=0;j<MAX;j++)
     printf("  %2d",tab[i][j]);
          printf("\n");    }
}
```

2.3.2. Structuri

În multe aplicaţii, este necesară gruparea datelor sub forma unei mulţimi de elemente care să poată fi prelucrate atât element cu element cât şi global, ca mulţime. Dacă toate elementele sunt de acelaşi tip, mulţimea de elemente poate fi definită ca tablou; dacă elementele sunt diferite ca tip, atunci mulţimea lor se defineşte ca *structură*.

Structura este o colecţie de date de tipuri diferite, grupate sub acelaşi nume. Un exemplu simplu de structură este *data calendaristică*; ea conţine 3 componente: **zi** (1, 2, . . . , 31) - de tip întreg, **luna** - de tip şir de caractere şi **anul** - de tip întreg.

Pentru a utiliza structuri în programe, acestea trebuie definite.

Sintaxa definirii unei structuri este:

```
struct <nume>{
             declaratii de variabile ;
             }
```

unde `<nume>` este un *identificator de tip de date* definite de utilizator, `struct` este cuvânt cheie iar declaraţiile de variabile sunt cele obişnuite.

După o asemenea definiţie, se pot declara variabile sau funcţii de tipul `<nume>` . Exemple:

```
struct complex{float re; float im};
struct student{char nume[25];int gr;float
medie_an};
```

Aceste declaraţii definesc numai structurile, fără a aloca memorie, deoarece nu au fost încă declarate variabile - structură.

Pentru a defini obiecte concrete de tip structură, cu rezervare de memorie, se scrie:

```
struct complex z1, z2, zeta;
struct student std[99];
```

Variabilele **z1, z2, zeta** sunt de tip "complex", deci fiecare va avea două componente reale, **re** şi **im**. Cele 100 de variabile **std[k]** sunt de tip "student", fiecare cu 3 componente: **nume, gr, medie_an.**

Definirea structurii şi declararea variabilelor de tip structură pot fi combinate ca mai jos:

```
struct complex{float re; float im} z1, z2, zeta;
```

situație în care numele structurii ("complex") ar putea lipsi, dacă definirea conține şi declarații de variabile:

```
struct {float re; float im} z1, z2, zeta;
```

dar acest mod de lucru nu se recomandă, mai ales când se lucrează cu mai multe tipuri de structuri. Este recomandabil ca structurile să aibă asociate nume pentru identificarea tipurilor lor şi să se folosească declarația **typedef** pentru a lucra mai comod:

```
struct complex{float re; float im};
typedef struct complex COMPLEX;
COMPLEX z1, z2, zeta;
```

sau prin combinarea lui **typedef** cu definiția structurii:

```
typedef struct {float re; float im} COMPLEX;
COMPLEX z1, z2, zeta;
```

În concluzie, o structură asociază diverse tipuri de variabile, simple sau structurate, sub un singur tip de date. Întregul grup poate fi, apoi, tratat global (prin atribuiri, comparări, transmitere la funcții etc.).

Inițializarea structurilor. Se poate face la declarare, prin precizarea unor valori pentru variabilele componente, de exemplu:

```
COMPLEX  zeta = {1.0, -2.33};
struct student sx = {"Tanase Ion", 8310, 9.33};
```

sau, dacă structura conține date mai complexe (tablouri, alte structuri), se pot folosi mai multe niveluri de acolade:

```
struct tty {int a[10]; int b[3]; char *s};
struct tty  x =   {
```

```
                    {2, 2, 3, 3, 10, 20},
                    {22, 44, 66},
                    "Cheia de criptare, codata"
                    };
```

Se inițializează primele 6 elemente ale tabloului **a** (celelalte se vor inițializa automat cu 0), toate cele 3 elemente ale tabloului **b** şi şirul de caractere **s**.

Accesul la componentele unei structuri. Se face prin operatorul "**.**" aplicat structurii, conform sintaxei:

```
<nume variabila>.<nume componenta>
```

Exemple:

```
COMPLEX z, zeta;
struct student sx;
z.re=2.0;
z.im=7.2;
zeta=z;//echivalent cu zeta.re=z.re; zeta.im=z.im;
sx.nume="Niculescu D. R. Liviu";
sx.gr= 8315;
sx.medie_an= 8.99;
```

Este posibil ca o structură să conțină o altă structură drept membru; în exemplul următor, se defineşte structura `PERSOANA`, care are o componentă `adr` (adresa persoanei), aceasta fiind la rândul său o structură cu trei componente: `oras, strada, cod`.

```
typedef struct  { char *oras;
                  char *strada;
                  long cod; } ADRESA;
typedef struct  { char *nume; ADRESA adr;}
PERSOANA;
PERSOANA zz;
zz.nume="Gheorghe Hagi";
zz.adr.oras="Constanta";
zz.adr.strada="B-dul Tomis 44";
zz.adr.cod=5567;
```

Dacă elementele sunt tipuri structurate (tablouri, şiruri etc.), accesul se face în mod natural: `x.nume` este un pointer la caracter iar `x.nume[i]` reprezintă elementul *i* al şirului `x.nume`.

Pointeri la structuri şi accesul prin pointeri. Un pointer către o structură se declară similar cu pointerii către variabile simple:

```
COMPLEX  z, *pz;
```

unde *pz este un pointer către o structură de tipul complex; se poate face acum atribuirea:

```
pz=&z;
```

prin care se dă lui pz adresa structurii z. În operațiile cu pointeri la structuri (comparații, atribuiri), aceştia trebuie să fie definiți către acelaşi tip de structură (acelaşi nume de structură). Chiar dacă două structuri sunt identice ca număr şi tip de componente dar au nume diferite, pointerii corespunzători nu sunt compatibili. Exemplu:

```
struct tip_a {int x; float y} s1, *p1;
struct tip_b {int x; float y} s2, *p2;
p1=&s2; // eroare, adresele sunt incompatibile
```

Dacă p este un pointer către o structură, atunci *p este structura iar (*p).nume este un membru al structurii. Să considerăm din nou structura student, anterior definită, variabila sx de tip student şi un pointer ps către o structură student, inițializat cu adresa lu sx; se pot face atribuirile de mai jos:

```
struct student sx, *ps=&sx;
(*ps).nume="Ionescu Octavian";
(*ps).gr=442;
(*ps).medie_an= 9.73;
```

Limbajul C permite o formă echivalentă pentru aceste construcții: dacă p este un pointer către o structură iar ms este un membru al acestei structuri, atunci **(*p).ms** se poate scrie echivalent **p->ms**, unde simbolul **->** este format din caracterul - (minus) urmat de caracterul > (mai mare). Cele trei atribuiri de mai sus, se pot scrie, aşadar:

```
ps->nume="Ionescu Octavian";
ps->gr=442;
ps->medie_an= 9.73;
```

Se pot folosi operatorii ++ / -- pentru p sau ms:

```
     (++p)->ms; //incrementeaza p si apoi acceseaza pe
ms
     (p++)->ms; //acceseaza ms si apoi incrementeaza p
     (p->ms)++; // incrementeaza (p->ms), dupa acces
     ++(p->ms); // incrementeaza (p->ms),inainte de
acces
```

Toate regulile referitoare la pointeri către variabile simple rămân valabile şi la pointeri către structuri.

3 Instrucțiuni

Instrucțiuni simple	Instrucțiuni structurate	
♦ de atribuire	♦ instr. compusă	♦ { . . . }
♦ break	♦ alternative	♦ if
♦ continue		♦ switch
♦ goto	♦ repetitive	♦ while
♦ return		♦ do while
		♦ for

Descrierea unui proces de calcul se realizează prin instrucțiuni.

Instrucțiunile simple sunt auxiliare, fiind utilizate în componența celor structurate.

Orice proces, poate fi descris prin trei structuri de control fundamentale: secvențială, alternativă şi repetitivă.

Structura secvențială elementară este reprezentată de instrucțiunea compusă.

3.1. Instrucțiunea compusă

Instrucțiunea compusă este o succesiune de instrucțiuni incluse între paranteze acolade, care poate fi precedată de declarații.

Dacă sunt prezente declarații, atunci variabilele definite sunt valabile doar în timpul execuției instrucțiunii compuse.

Instrucțiunea compusă este tratată sintactic ca o singură instrucțiune; de aceea, se utilizează când sintaxa permite existența unei singure instrucțiuni: în structura if, for etc.

Exemplu: schimbarea reciprocă a valorilor între două variabile, a, b:

```
{       // instructiune compusa
        int t ;         // variabila ajutatoare
        t=a  ;
        a=b ;
        b=t  ;
}
```

După execuția secvenței de instrucțiuni, variabila t nu mai este stocată în memorie şi deci devine inexistentă.

Partea executabilă a unei unei funcţii (corpul funcţiei) este de fapt o instrucţiune compusă.

3.2. Instrucţiunea if

Corespunde structurii alternative cu două ramuri, sau unui bloc de decizie cu două ieşiri. **Sintaxa** instrucţiunii if este:

```
if  (expresie)
    instructiune_1 ;
[else
    instructiune_2 ; ]
```

Ramura else, inclusă între [ ], este opţională; instructiune_1 şi instructiune_2 pot fi oricare instrucţiuni ale limbajului, simple sau structurate, inclusiv if. Dacă este necesar, una sau ambele pot fi înlocuite cu instrucţiuni compuse.

Efectul:

- se evaluează expresia (se calculează valoarea expresiei) ;
- dacă este adevărată (valoare diferită de zero), se execută numai instructiune_1 (instructiune_2 nu este luată în consideraţie);
- dacă este falsă (valoare = 0), se execută numai instrucţiune_2; dacă aceasta lipseşte, se trece la instrucţiunea următoare din program.

Observaţii:

- În C, valoarea logică "adevărat" este asociată cu "$\neq 0$" iar valoarea logică "fals" este asociată cu "= 0".

  Valoarea expresiei poate fi de tip întreg:

```
if (k) x=k+3 ; // pentru k<>0, se executa x=k+3
if (k=5) x=k+3 ; // se executa totdeauna x=k+3, pentru ca expr.=5
if (k==5) x=k+5 ; // se executa x=k+3, numai pentru k=5
```

- Dacă una sau ambele instrucţiuni din if, sunt tot instrucţiuni if, else se asociază cu cea mai apropiată instrucţiune if anterioară, din acelaşi bloc, care nu are ramură else. Exemplu:

```
if (y==1)
      if (a==2)
      x=0;
      else x=2;
```

Ramura else aparţine lui if (a==2), cea mai apropiată instrucţiune if anterioară, care nu are ramură else. Dacă se doreşte altă aparteneţă, se pote utiliza o instrucţiune compusă:

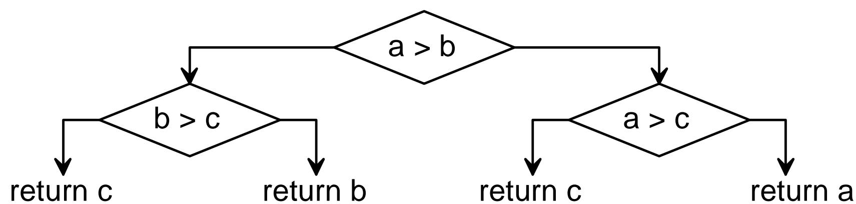
```
if (y==1)
{     if (a==2)
      x=0;
}
else x=2; // else apartine lui if (y==1)
```

1. Se consideră funcția $f: \boldsymbol{R} \to \boldsymbol{R}$, $f(x) = \begin{cases} \frac{\sqrt{3x^2+1}-1}{x^2+1} & , x \neq 0 \\ 3/2 & , x = 0 \end{cases}$.

Să se afişeze valoarea funcției, pentru o valoare a lui x introdusă de la tastatură.

```
#include <stdio.h>
#include <math.h>
int main(void) {
double x, f ;
printf("Introduceti valoarea lui x: ");
scanf("%lf ", &x); //totdeauna, argumentul lui scanf este adresa: &x
if (x !=0)
      f=(sqrt(3*x*x+1)-1)/(x*x+1);     // instr_1
else
      f=3.0/2.0 ;                      // instr_2
printf("Valoarea functiei este: %lf \n", f);
return 0;
}
```

2. Se consideră trei valori întregi; să se determine cea mai mare. Se utilizează instrucțiunea if pe mai multe niveluri.




```
if (a>b)
      if (a>c)  return a ; // intoarce ca rezultat valoarea a
      else      return c ;
else
      if (b>c)  return b ;
      else      return c ;
```

Se observă alinierea fiecărui else cu if -ul căruia îi aparține.

3.3. Instrucțiunea switch

Permite realizarea structurii selective cu mai multe ramuri. Este o generalizare a structurii alternative de tip if şi ca urmare poate fi realizată cu mai multe instrucțiuni if incluse unele în altele, pe mai multe niveluri.

Utilizarea instrucțiunii switch, face ca programul să fie mai clar decât dacă se utilizează if pe mai multe niveluri.

Sintaxa instrucțiunii switch:

```
switch (expresie)      {
case const_1:  instructiune_1;
               instructiune_2;
               . . . . . . . . . . ;
               break ;
case const_2:  instructiune_1;
               instructiune_2;
               . . . . . . . . . . ;
               break ;
 . . . . . . . . . . . . . . . . . . . . . . . .
case const_n:  instructiune_1;
               instructiune_2;
               . . . . . . . . . . ;
               break ;
[ default:     instructiune_1; // ramura optionala
               instructiune_2;
               . . . . . . . . . . ; ]
}
```

Efectul instrucțiunii switch este descris mai jos:

- Se evaluează expresie (de tip întreg) şi se compară cu valorile constante (care trebuie să fie distincte), din lista de etichete case;
- Dacă valoarea expresiei coincide cu una din constantele const_k, se execută <u>numai</u> secvența de instrucțiuni corespunzătoare acestei etichete, până la break, goto sau return şi se trece la instrucțiunea următoare din program, aflată după paranteza

acoladă }; în cazul în care secvenţele de instrucţiuni nu conţin break, goto sau return, se execută secvenţa cazului, precum şi toate secvenţele care urmează până la sfârşitul lui switch;

- Dacă valoarea expresiei nu coincide cu nici una din constantele notate const_1, const_2, . . . , const_n, se execută succesiunea de instrucţiuni aflată după default (dacă există) şi se trece la instrucţiunea următoare din program, aflată după paranteza acoladă }.

- Un caz (*case*) poate conţine una, mai multe sau nici o instrucţiune, adică pot exista mai multe cazuri asociate la aceeaşi secvenţă de instrucţiuni.

Exemple:

```
switch (c=getch( )) {
      case 'd':
      case 'D':   del_file;  break;
      case 'l':
      case 'L':   list_file;  break;
      default:    error( );  break;
}
```

Se citeşte un caracter de la consolă (cu getch()) şi se execută una din funcţile del_file(), list_file() sau error(), după cum caracterul este 'd' sau 'D', respectiv 'l' sau 'L', respectiv diferit de cele patru caractere.

Comanda break asigură ieşirea din switch după execuţia unei funcţii.

Comanda break de la default nu este necesară, dar este o bună măsură de protecţie, în cazul în care se adaugă ulterior alte cazuri, după default.

```
double calcul (doble op1, char operator, double op2)
{
switch (operator)
      {
      case '+' :  return op1+op2;
      case '-' :  return op1-op2;
      case '*':   return op1*op2;
      case '/':   return op1/op2;
      default:    printf("Operator necunoscut\n");
      exit(1);
      }
```

}

Funcția calcul(), de mai sus, se apelează cu trei argumente: op1, op2 şi un operator (+, -, *, /); ea returnează rezultatul operației, în funcție de operatorul de tip caracter, care a fost specificat în lista de argumente.

Funcția exit(1) produce terminarea execuției programului; dacă valoarea specificată ca parametru este zero, arată terminare normală, în caz contrar defineşte o terminare anormală, deci prezența unei erori. În cazul de mai sus, se produce o terminare forțată, din cauza unei erori de operator. Funcția calcul(), în mod normal, returnează o valoare reală de tip double. Din cauza operatorului necunoscut, funcția nu poate returna o valoare normală, fapt ce necesită terminarea forțată a programului, altfel eroarea va genera un lanț de alte erori, foarte greu de depistat.

Funcția exit() este o funcție de bibliotecă.

Pentru utilizarea în programe trebuie specificate fişierele header <stdlib.h> sau < process.h>.

3.4. Instrucțiunea while

Corespunde structurii repetitive condiționate anterior. Sintaxa:

```
while (expresie)
      instrucțiune ;
```

Efectul instrucțiunii while:

- **1**. Se evaluează expresia;
- **2**. Dacă are valoarea "adevărat" (este diferită de zero), se execută instrucțiunea şi se repetă pasul **1**.
- **3**. Dacă are valoarea "fals" (este egală cu zero), execuția instrucțiunii while se încheie şi se trece la instrucțiunea următoare din program.

Pe poziția "instrucțiune" este acceptată o singură instrucțiune C; dacă sunt necesare mai multe, acestea se includ între paranteze acolade şi formează astfel o singură instrucțiune, compusă.

Observații:

♦ Dacă la prima evaluare, expresie=0, while se încheie fără să se execute "instrucțiune".
♦ Dacă în partea executabilă apar instrucțiuni ca break, goto sau return, se schimbă acțiunea normală a lui while, prin încheierea ei înainte de termen.
♦ Instrucțiunea while se încheie dacă după expresie se pune ';'

```
        while (k<=100)        ; // eroare: corpul lui while este vid
              k=k+1      ; // se considera "instructiune urmatoare"
```

♦ Execuția instrucțiunii while trebuie să producă modificarea valorii "expresie" după un număr finit de repetări; în caz contrar, se produce un *ciclu infinit*.

Exemplul 1: Tabelarea valorilor unui polinom:

```
# include <stdio.h>
main()     /*afiseaza valorile lui p(x), pentru x=1, 2, . . .,10
            p(x)=x*x - 3*x+2 */
{
int x=1;
while (x<=10) {
      printf("x = %d \t p(x) = %d \n",x, x*x - 3*x +2);
      x++ ;
      }
}
```

Exemplul 2: Tabelarea valorilor funcției *sinus()*. Funcția sinus este o funcție standard, se află în fişierul *math.h* şi are prototipul

```
double sin(double x);
```

Argumentul x este interpretat în radiani; pentru a utiliza în program valori ale lui x în grade, acestea trebuie înmulțite cu factorul f=PI/180.

```
# include <stdio.h>
# include <math.h>
# define PI 3.14159265358979
main()
{
int x=0 ;
double f=PI/180.0 ;
while (x<=360) {
    printf("sin(%d)= %.16f\n",x,sin(x*f));
    x++ ;
```

```
        }
    }
```

Valorile se afişează câte una pe linie, deci programul afişează 360 de linii. Pe ecran pot fi analizate doar ultimele 25 de linii. În asemenea situații se pune problema de a întrerupe temporar execuția programului, după ce s-au afişat 25 de linii. Oprirea programului se poate face în mai multe moduri; un mod simplu de oprire este apelarea funcției getch() în momentul în care ecranul este plin. Apelarea funcției getch() opreşte execuția, deoarece se aşteaptă acționarea unei taste şi se afişează fereastra utilizator. La acționarea unei taste, programul intră din nou în execuție.

În exemplul următor se apelează funcția getch() după afişarea a 23 de valori (linii); pe linia 24 se va afişa textul:

Daca doriti sa continuati, apasati o tasta!

Se utilizează valorile lui x =0, 1, . . . , 360, pentru a apela getch, când x ia valorile 23, 46, 69, . . . , deci multiplu de 23. Aceasta se exprimă prin expresia x % 23 ==0.

```
# include <stdio.h>
# include <math.h>
# define PI 3.14159265358979
main()
{
int x=0 ;
double f=PI/180.0 ;
while (x<=360) {
    printf("sin(%d)= %.16f\n",x,sin(x*f));
    x++ ;
    if(x % 23==0){
        printf("Daca doriti sa continuati,
        apasati o tasta! \n");
        getch() ;
        }
    }
}
```

3.5. Instrucțiunea do - while

Corespunde structurii repetitive condiționate posterior. Efectul ei poate fi obținut cu celelalte instrucțiuni repetitive. Prezența ei în setul de instrucțiuni asigură o mai mare flexibilitate şi claritate programelor.

Sintaxa:

```
do
      instructiune
while (expresie) ;
```

Efectul instrucțiunii do - while:

- **1**. Se execută "instrucțiune";
- **2**. Se evaluează expresia;
- **3**. Dacă are valoarea "adevărat" (diferită de zero), se reia pasul **1**.
- **4**. Dacă are valoarea "fals" (zero), execuția instrucțiunii do-while se încheie şi se trece la instrucțiunea următoare din program.

Pe poziția "instrucțiune" este acceptată o singură instrucțiune C; dacă sunt necesare mai multe, acestea se includ între paranteze acolade şi formează astfel o singură instrucțiune, compusă.

Observații:

- Blocul "instrucțiune" se execută cel puțin o dată, chiar dacă la prima evaluare, expresie=0.
- Dacă în partea executabilă apar instrucțiuni ca break, goto sau return, se schimbă acțiunea normală a lui do-while, prin încheierea ei înainte de termen.
- Pentru a nu confunda, în textul programului, un while din do-while cu un ciclu while care conține instrucțiune vidă, se recomandă folosirea acoladelor chiar şi atunci când do-while conține o singură instrucțiune.

```
do   {
      instructiune
}     while (expresie) ;
```

- Execuția instrucțiunii do-while trebuie să producă modificarea valorii "expresie" după un număr finit de repetări; în caz contrar, se produce un *ciclu infinit*.

Exemple:

```
# include <stdio.h>
# define PUNCT '.'
int main(void)
{
      char car ;
      int nr_sp=0 ;
      printf("Introduceti o fraza terminata cu punct\n");
      do   {
```

```
                car = getch();
                if (car == ' ')  nr_sp++;
                }
        while (car != PUNCT);
        printf("S-au gasit %d spatii albe \n", nr_sp);
        return 0 ;
}
```

Programul de mai sus contorizează numărul de spații albe dintr-o frază, introdusă de la tastatură. La citirea caracterului punct ".", ciclul do-while se încheie şi se afişează numărul de spații.

Analog, se poate contoriza tastarea unui caracter oarecare, prin modificarea condiției din instrucțiunea `if(car == ' ') nr_sp++;` exemplu:

```
if (car == 'd') nr_sp++ ;
```

Exemplul următor este un program care calculează rădăcina pătrată dintr-un număr real pozitiv, prin metoda iterativă, a lui Newton. Se utilizează şirul $(x_n)_{n\in \boldsymbol{N}}$, definit prin recurență:

$$x_{n+1} = \frac{x_n^2+a}{2\cdot x_n}, \quad x_0 = 1.$$

Acest şir este convergent şi are limita $L = \sqrt{a}$.

Termenii şirului se calculează succesiv: $x_0, x_1, \ldots, x_N$, până când diferența dintre doi termeni consecutivi este mai mică dacât o valoare impusă, de exemplu, $\varepsilon = 10^{-10}$:

$$|x_N - x_{N-1}| \le \varepsilon$$

Valorea lui x_N este limita şirului şi deci $x_N = \sqrt{a}$.

```
#include <stdio.h>
#include <stdlib.h>
#define  EPS 1e-10
main( )    //se calculeaza radical din a = numar real pozitiv
{
double a, x1, x2, y ;
printf("Introduceti valoarea lui a>0, a= ");
scanf(" %lf ", &a);
x2=1;
do   {
      x1=x2;
      x2=0.5*(x1+a/x1);
      if ((y=x1-x2)<0)  y=-y ; // valoarea absoluta a lui y
      }     while (y>EPS);
printf("a=% .11g \t radical(a)=% .11g\n", a, x2);
```

```
}
```

În program, x1 şi x2 sunt două valori consecutive ale termenilor şirului. Ele sunt necesare pentru estimarea erorii şi oprirea procesului de calcul, când eroarea este mai mică sau egală cu EPS.

Variabila y este utilizată pentru diferența a doi termeni consecutivi, valoarea absolută a acesteia şi compararea cu EPS.

Specificatorul .11g arată că valoarea se tipăreşte pe minim 11 spații după virgulă, în formatul cel mai economic (normal sau cu exponent).

Numărul de iterații (de repetări ale buclei do-while) nu este anterior cunoscut; el depinde de valorea EPS impusă: cu cât această valoare este mai mică, volumul de calcul (numărul de repetări) creşte. Pentru aflarea numărului de repetări, se poate introduce un contor, inițializat cu zero, care este incrementat la fiecare trecere. În final, valoarea contorului arată numărul de treceri prin do-while.

3.6. Instrucțiunea for

Corespunde structurii repetitive condiționate anterior, ca şi în cazul instrucțiunii while; spre deosebire de while, instrucțiunea for utilizează trei expresii, fiecare având un rol propriu. Sintaxa:

```
for (expresie_1; expresie_2; expresie_3)
    instructiune ;
```

Prin definiție, construcția de mai sus este absolut echivalentă cu secvența:

```
expresie_1;
while (expresie_2)  {
      instructiune ;
      expresie_3 ;
}
```

Din construcția echivalentă cu while, rezultă că:

- expresie_1 este interpretată ca o instrucțiune care se execută o singură dată (inițializare);
- expresie_2 este condiția logică de execuție a ciclului;
- expresie_3 este interpretată ca instrucțiune de actualizare;

Instrucţiunea for este, în esenţă, un while dotat cu o iniţializare şi o actualizare la fiecare iteraţie, actualizare care se face după execuţia instrucţiunii din corpul while.

Pe poziţia "instrucţiune" este acceptată o singură instrucţiune C; dacă sunt necesare mai multe, acestea se includ între paranteze acolade şi formează astfel o singură instrucţiune, compusă.

Observaţii:

- Atât expresie_1 cât şi expresie_3 pot lipsi:

```
for ( ;expresie_2; )
     instructiune;
```

Ceea ce se obţine este while, care este preferabilă lui for din punct de vedere al clarităţii. Dacă lipseşte şi expresie_2, condiţia este tot timpul adevărată, ceea ce duce la repetare infinită:

```
for( ; ; ) instructiune;
```

determină un ciclu infinit, din care se poate ieşi numai cu break, goto sau return.

- Sub influenţa practicii din alte limbaje de programare, se consideră şi se prezintă uneori instrucţiunea for limitată la un ciclu cu contor. Evident, cu for se pot implementa şi cicluri cu contor dar este greşit să limităm posibilităţile lui for la cicluri cu contor. Faţă de for din alte limbaje, în C instrucţiunea este mai complexă. Cele trei expresii

pot conţine diferite instrucţiuni ale limbajului şi se pot utiliza variabile diferite:

```
#include <stdio.h>
int main(void)
{
float depunere=1000.0;
an=2000
for(printf("Depozit initial:\n");an++<=2010;
printf("Depozit in %d este %.2f\n",an,depunere))
 depunere=depunere * 1.19;
return 0;
}
```

Programul de mai sus calculează suma dintr-un depozit bancar, care beneficiază de o dobândă anuală de 19%, pentru o periadă de 10 ani. Dobânda se varsă în contul de depozit după fiecare an iar dobânda anului următor se calculează pentru suma majorată.

Prima expresie din for este un mesaj care se scrie o singură dată.

A doua expresie conţine condiţia de repetare şi incrementarea variabilei "an".

A treia expresie conține mesajul privind suma din depozit după fiecare an din perioada stabilită: 2000 - 2010.

- ♦ O eroare frecventă este for (i=1; i=n; i++), care atribuie lui i valoarea n. Dacă n este diferit de zero, acest for va cicla la infinit. O altă eroare este for (i=0; i==n; i++), în care se testează dacă i este identic cu n, uitând că exp_2 este condiție de continuare, nu de terminare.

Exemple:

1. Repetări cu număr cunoscut de paşi:

```
double putere(double nr, int n)
{
    double x_la_n;
    int k ;
    x_la_n=1.0 ;
for (k=1 ; k <=n ; k++)
    x_la_n *= x ;
    return x_la_n ;
}
```

Funcția putere() primeşte ca argumente numerele x şi n şi întoarce valoarea x^n, prin înmulțire cu x, repetată de n ori.

2. Calcul de sume cu număr cunoscut de termeni:

```
float suma(int n)
{
    float sum=0;
    int k;
for (k=0 ; k <=n ; k++)
    sum += 1.0/((k+1.)*(k+2.));
    return sum ;
}
```

Funcția sum() calculează suma $\sum_{k=0}^{n} \frac{1}{(k+1)(k+2)}$, pentru valoarea lui n specificată de argument.

3.7. Instrucțiunea continue

Se utilizează numai în corpul unei instrucțiuni repetitive (while, do-while, for). Sintaxa:

```
continue;
```

Permite revenirea la începutul blocului care se execută repetat, înainte de încheierea sa. La apariția instrucțiunii continue, se renunță la executarea instrucțiunilor următoare şi se execută un salt la începutul blocului.

Prezența instrucțiunii măreşte flexibilitatea programelor şi contribuie la rezolvarea mai simplă a unor ramificații.

Nu se utilizează în cazul instrucțiunii switch.

3.8. Instrucțiunea break

Produce ieşirea forțată din instrucțiunile while, do-while, for, switch. Dacă sunt mai multe asemenea instrucțiuni incluse unele în altele, se realizează numai ieşirea din instrucțiunea curentă (în care este break).

Sintaxa:

```
break;
```

În cazul instrucțiunii for, ieşirea se realizează fără reactualizarea expresiei exp_3.

În instrucțiunea switch, prezența lui break este strict necesară pentru a produce ieşirea, după tratarea cazului selectat; în celelalte instrucțiuni, se poate evita break, prin includerea unor condiții suplimentare în câmpul "expresie", care condiționează repetarea.

3.9. Instrucțiunea goto

Este o instrucțiune de salt necondiționat la o instrucțiune etichetată, care se află în aceeaşi funcție. Etichetele se definesc printr-un nume, urmat de două puncte ":".

Deşi principiile programării structurate recomandă evitarea folosirii lui goto, instrucțiunea se dovedeşte uneori utilă. De exemplu, ieşirea forțată dintr-un ciclu inclus în altul (cu break se iese numai din ciclul interior). În secvența următoare, se caută o valoare comună în două tablouri de întregi, a şi b:

```
for (i=0; i<n; i++)
      for (j=0; j<m; j++)
      if(a[i]==b[j])  goto ok;
```

```
pritf("Nu sunt elemente comune\n");
. . . . . . . . . . . . . . . . . . . . .
ok:  printf("S-a gasit a[%d] = b[%d]=%d\n", i, j, a[i]);
. . . . . . . . . . . . . . . . . . . . .
```

O construcție cu goto se poate înlocui întotdeauna cu una structurată introducând eventual variabile suplimentare. Secvența anterioară se poate scrie:

```
int gasit = 0;
for (i=0; i<n && ! gasit; i++)
     for (j=0; j<m && ! gasit; j++)
     gasit = (a[i]==b[j]) ;
if (gasit)
     printf("S-a gasit a[%d] = b[%d]=%d\n", i-1, j-1, a[i-1]);
else pritf("Nu sunt elemente comune\n");
. . . . . . . . . . . . . . . . . . . . .
```

Instrucțiunile break şi continue permit construirea unor cicluri while, do - while, for, cu mai multe puncte de ieşire sau ramificații interne, evitând folosirea instrucțiunii goto.

Din sintaxa instrucțiunilor while, do-while, for, rezultă că punctele lor de ieşire pot fi doar la început (când blocul executabil nu se parcurge nici măcar o dată) sau la sfârşit. Sunt situații, când ieşirea este necesară undeva "la mijloc", caz în care se foloseşte un ciclu infinit şi un break într-o instrucțiune if.

4 Funcții

4.1. Definirea şi apelul funcțiilor

Funcțiile sunt elementele constructive ale limbajului C şi sunt de două categorii:

- funcții standard (de bibliotecă) definite în fişiere *.h: **printf(), scanf(), sin(), strlen(), getchar(), putchar(),** . . . ; acestea pot fi utilizate în programe, prin specificarea fişierului header în care se află.
- funcții definite de utilizator, altele decât cele standard.

În programul sursă, o funcție definită de utilizator, are două părți: *antetul* şi *corpul funcției*. O funcție poate avea un număr de parametri (variabile) dar o singură valoare - *valoarea funcției*. Antetul

conţine informaţii despre tipul valorii funcţiei, numărul şi tipul parametrilor şi conţine identificatorul (numele) funcţiei.

Structura unei funcţii:

<tip> <nume> (<lista declaraţiilor parametrilor formali>)
{
<declaraţii>
<instrucţiuni>
}

Primul rând este antetul; acesta apare obligatoriu înaite de corpul funcţiei dar şi la începutul programului (înainte de main()) situaţie în care se numeşte *prototip*.

În cazul tipurilor standard, <tip> din antet este un cuvânt cheie care defineşte tipul valorii returnate de funcţie.

În C există două categorii de funcţii:

- funcţii cu tip, care returnează o valoare, de tipul <tip>;
- funcţii fără tip, care nu returnează nici o valoare după execuţie, dar efectuează anumite prelucrări de date; pentru acestea se va folosi cuvântul rezervat ***void*** în calitate de <tip>.

O funcţie poate avea zero, unul sau mai mulţi parametri. Lista declaraţiilor parametrilor (formali) este vidă când funcţia nu are parametri.

În corpul funcţiei pot să apară una sau mai multe instrucţiuni return, care au ca efect revenirea în programul apelant. Dacă nu apare nici o instrucţiune return, revenirea în programul apelant se face după execuţia ultimei instrucţiuni din corpul funcţiei.

Valoarea returnată de o funcţie. Instrucţiunea return

Forma generală: return <expresie> ;
unde "expresie" trebuie să aibă o valoare compatibilă cu tipul declarat al funcţiei. Efectul constă în încheierea execuţiei funcţiei şi revenirea în programul apelant; se transmite valoarea calculată <expresie> către programul apelant, reprezentând **valoarea funcţiei**, ce va înlocui **numele funcţiei**. Dacă funcţia este de tip void, expresia lipseşte. Exemplu:

```
void fact(int n); //calculeaza n!
{
int k; long fct;
if (n<0)     {
```

```
        printf("Valoarea arg. negativa !"\n);
        return;//nu se calculeaza n!
              }
    for (fct=1, k=1; k<=n; k++)
        fct *=k;
    printf(Valoarea %d!=%ld\n", n, fct);
    return; // s-a calculat n! si s-a afisat
    }
```

- Dacă într-o funcție sunt mai multe comenzi return, prima ce se execută va produce revenirea în programul apelant.
- O funcție poate transmite prin return o singură valoare, de orice tip (inclusiv structură) dar fără tipul tablou.
- Valoarea returnată de o funcție cu tip trebuie să fie compatibilă cu tipul declarat al funcției; de exemplu, dacă o funcție este definită prin int f(), valoarea returnată poate fi de orice tip aritmetic, deoarece aceasta poate fi convertită imediat la tipul întreg:

```
int f()
{
int i, float a; char c;
. . . . . . . . . . . . . . . .
return i;    //valoarea coincide cu tipul funcției
. . . . . . . . . . . . . . . .
return a;    //valoarea "a" este convertita la int
. . . . . . . . . . . . . . . .
return c;    //valoarea "c" este convertita la int
. . . . . . . . . . . . . . . .
}
```

- Dacă o funcție returnează o valoare de tip adresă (pointer), cerințele sunt mai restrictive: **tipul valorii** returnate trebuie să fie exact **tipul funcției**.

```
char *f()
{
char c, *adr_c, tablou_c[5], **adr_adr_c;
int *adr_i;
. . . . . . . . . . . .
c='A' ;
adr_c=&c;
return adr_c ; //corect, adr_c = tipul functiei
. . . . . . . . . . . .
*adr_adr_c=adr_c;
return          *adr_adr_c;//corect,*adr_adr_c=tipul
functiei
adr_c=tablou_c;
return tablou_c;  //corect, numele tabloului e un
                  //pointer de acelasi tip
                  //cu tipul functiei
. . . . . . . . . . . .
```

```
return *adr_c;//gresit, se returneaza un caracter
              //in loc de adresa
. . . . . . . . . . . . .
return adr_i; //gresit, tipul pointerului (int) nu
              //corespunde cu tipul functiei
. . . . . . . . . . . . .
return ;      //valoarea returnata nu este definita
```

În cazul în care funcția are mai mulți parametri, declarațiile de tip pentru parametri se separă prin virgulă.

Parametrii se utilizează pentru a permite transferul de date, către o funcție, în momentul utilizării ei în program. La construirea unei funcții, se face abstracție de valorile concrete. Acestea vor fi prezente numai la execuția programului.

Exemple:

```
void f(int a, int b, char c)  /*corect, se specifica tipul fiecarui param.*/
void f(int a, b, char c)      /* incorect, nu se specifica tipul lui b */
```

Parametrii unei funcții, ca şi variabilele definite în corpul funcției sunt *variabile locale*, adică sunt valabile numai în interiorul funcției; ele nu sunt recunoscute în exterior, nici ca nume, nici ca valoare.

În momentul compilării, este necesară doar cunoaşterea tipurilor valorilor pe care le vor primi parametrii la execuție. Aceste declarații de tip sunt indicate în antetul funcției şi vor servi compilatorului să rezerve memoria necesară pentru fiecare parametru.

Parametrii declarați în antetul unei funcții se numesc *formali,* pentru a evidenția faptul că ei nu reprezintă valori concrete, ci numai țin locul acestora, pentru a putea exprima procesul de calcul. Ei se concretizează la execuție prin valori ce rezultă din apelarea funcției, când sunt numiți *parametri efectivi*.

Observații:

- ♦ Dacă nu se specifică tipul unei funcții, se consideră automat că ea va returna o valoare de tip *int*.
- ♦ În limbajul C++, controlul cu privire la tipul valorii unei funcții este mai strict şi ca urmare se recomandă, ca tipul să fie efectiv specificat, în toate cazurile.
- ♦ Pentru funcția principală ***main***, se pot utiliza antetele:

```
int  main()
int  main(void)
void main()
void main(void)
     main()
     main(void)
```

Primele două antete arată că funcţia returnează o valoare întreagă; de asemenea ultimele două antete arată că se returnează o valoare întreagă, chiar dacă nu se specifică tipul *int* în antet.

Se utilizează foarte frecvent varianta fără tip main().

Funcţia *main* poate avea şi parametri.

Programul următor produce tabelarea valorilor unei funcţii de două variabile, în domeniul [0, 1]x[0, 1], cu pasul 0.1.

```
# include <stdio.h>
double fm(double, double);
void main(void)
{
double x,  y,  pas=0.1;
for(x=0.0; x<=1.0; x=x+pas)
    for(y=0.0; y<=1.0; y=y+pas)
    printf("x=%lf y=%lf f(x,y)=%lf\n",x,y,fm(x,y));
{
double fm(double x, double y)
{
    return (3.0*x*y + 1.0)/(1.0 + x + y + x*y);
}
```

Programul conţine două funcţii: main() şi fm(). Funcţia matematică fm() are doi parametri de tip real, doublă precizie şi anume x şi y iar tipul valorii funcţiei este tot real, dublă precizie. Numele funcţiei reprezintă valoarea întoarsă de aceasta, de exemplu, fm(0.0, 1.0)=0.5 este valoarea lui fm când argumentele sale iau valorile x=0 şi y=1.

Instrucţiunea return <expresie>, întoarce valoarea expresiei în programul apelant, această formă fiind obligatorie la funcţiile cu tip.

Declaraţia

double fm(double, double);

de la începutul programului este un *prototip* al funcţiei fm(), care descrie tipul funcţiei, numele şi tipul fiecărui parametru. În acest mod, funcţia fm() este recunoscută în main(), chiar dacă definiţia ei se află după main().

În general, este indicat să se scrie prototipurile tuturor funcţiilor, atât ale celor definite în modulul curent de program, cât şi ale celor definite în alte module dar folosite în modulul curent.

Funcţiile de bibliotecă sunt complet definite (inclusiv prototipurile) în fişiere header, deci includerea acestor fişiere, asigură şi prezenţa prototipurilor. De exemplu, prototipul lui printf() este în fişierul stdio.h.

Apelul funcţiilor se face diferenţiat, după cum sunt cu tip sau fără tip. O funcţie fără tip se apelează prin:

```
nume(listă de parametri actuali); /*cand funcţia are parametri
nume();                           /* cand functia nu are parametri
```

O funcţie cu tip se apelează în general prin:

```
variabilă = nume(listă de parametri actuali);
```

sau, mai general, prin specificarea numelui funcţiei într-o expresie în care este permis tipul valorii funcţiei. Uneori nu ne interesează valoarea pe care o returnează o funcţie şi atunci funcţia se apelează exact ca o funcţie fără tip. Un exemplu uzual este funcţia printf(), care întoarce numărul de caractere scrise la consolă, dar apelul uzual al acestei funcţii se face ca şi cum ar fi de tip *void*.

Funcţiile au clasa de alocare implicită external, deci sunt vizibile în toate modulele care compun aplicaţia. Ca şi la variabilele externe, o funcţie este vizibilă din locul unde este declarată, până la sfârşitul fişierului sursă. Din acest motiv se foloseşte un prototip al funcţiei care se plasează la începutul textului sursă, în afara tuturor definiţiilor de funcţii. O altă posibilitate este includerea prototipurilor în fişiere **header**. Fişierele header predefinite conţin, între altele, prototipuri ale funcţiilor de bibliotecă. Prezenţa prototipurilor face posibilă verificarea la compilare a corespondenţei dintre numărul şi tipul parametrilor formali cu numărul şi tipul parametrilor actuali (la apelare). Exemple de prototipuri:

1. float fmat (float x, float y, int n)
2. int fdet (float, int, int, char)

Declaraţiile parametrilor formali pot conţine numele acestora (ca în exemplul 1) sau pot fi anonime, deci fără a specifica un nume pentru fiecare parametru (ca în exemplul 2).

În limbajul C, ordinea evaluării parametrilor actuali la apelul unei funcţii, nu este garantată. În fapt, această ordine este în general de la dreapta la stânga, dar programele trebuie asfel construite încât să funcţioneze corect indiferent de această ordine. De exemplu, secvenţa:

```
int n=7;
printf (" %d %d\n", n++, n);
```

va tipări (la Borland C): 7 7 şi nu 7 8 cum ar fi de aşteptat. Asemenea construcţii trebuie evitate, prin separarea în mai multe apeluri, de exemplu:

```
printf ("%d ", n++) ;
```

```
printf ("%d\n", n) ;
```

Acelaşi lucru este valabil şi la expresii în care intervin apeluri de funcţii. De exemplu, în secvenţa următoare nu se garantează care funcţie va fi apelată prima:

y = f( ) - g( ) ;

Dacă f() este apelată prima, şi calculul valorii lui f() are efecte asupra lui g() nu se obţine acelaşi rezultat ca în cazul apelării mai întâi a lui g(). Dacă ordinea este importantă, aceasta se poate impune prin:

y = -g( ) ; y = y + f( );

Exemplul 1 - (varianta 1, corectă)

Programul ilustrează necesitatea folosirii prototipului. Se calculează valoarea maximă dintr-un tablou de până la 20 de numere reale, introduse de utilizator.

```
#include <stdio.h>
#include <conio.h>
#define MAX 20
double   max(double   *tab,   int   n);//prototipul
functiei
void main()
{
double x[MAX];
int i;
printf("\n Tastati numerele,terminati cu 0:\n");
for(i=0; i<MAX; i++)
     {
     scanf("%lf",&x[i]);
     if(x[i]==.0) break;
     } // la iesire din for, i=nr.elemente introduse
printf("\n Maximul este %lf ", max(x,i));
getch();
}
double max(double *tab,int n)
/* functia max (),de tip double; parametrii sunt:
adresa tabloului si numarul de elemente */
{
int i; double vmax=*tab;
for(i=1;i<n;i++)
     if(vmax<*(tab+i))
          vmax=*(tab+i);
          return vmax;
}
```

Exemplul 1 - (varianta 2, incorectă)

```
#include <stdio.h>
#include <conio.h>
#define MAX 20
```

```
void main()
{
double x[MAX];
int i;
printf("\n Introd. numerele,terminati cu 0:\n");
for(i=0;i<MAX;i++)
     {
     scanf("%lf",&x[i]);
     if(x[i]==.0) break;
     } // la iesire din for, i = nr.elemente
printf("\n Maximul este %lf ", max(x,i));
/* functia max e considerata de tip int deoarece nu
     exista prototip */
getch();
}
double max(double *tab,int n)
/* functia max (),de tip double; parametrii sunt:
adresa tabloului si numarul de elemente */
{
int i; double vmax=*tab;
for(i=1;i<n;i++)
     if(vmax<*(tab+i))
          vmax=*(tab+i);
          return vmax;
}
```

La compilare, apare mesajul de eroare:

"Type mismatch in redeclaration of max*"*, datorită nepotrivirii tipului din definiție cu cel considerat implicit la apelul funcției; soluție:

- definirea funcției înaintea funcției *main* (incomod);
- folosirea prototipului.

Exemplul 1 - varianta 3 - corect----------

```
#include <stdio.h>
#include <conio.h>
#define MAX 20
double   max(double   tab[],int   n);//prototipul
functiei
/* tablou unidimensional  pentru care nu trebuie
precizat numarul de elemente;el trebuie cunoscut la
prelucrare si se transfera ca parametru separat */
void main()
{
double x[MAX];
int i;
printf("\n Introd. numerele,terminati cu 0:\n");
for(i=0;i<MAX;i++)
     {
```

```
        scanf("%lf",&x[i]);
        if(x[i]==.0) break;
        } // la iesire din for, i = nr.elemente
    printf("\n Maximul este %lf ", max(x,i));
    getch();
    }
    double max(double tab[],int n)
    /* functia max (),de tip double; parametrii sunt:
    tabloul tab[] si numarul de elemente */
    {
    int i; double vmax=tab[0];
    for(i=1;i<n;i++)
        if(vmax<tab[i])
            vmax=tab[i];
            return vmax;
    }
```

Exemplul 2:

Programul sursă este alcătuit din două fişiere, grupate într-un proiect. Se calculează puterea întreagă a unui număr real.

```
// princ.c
#include <stdio.h>
#include <conio.h>
double power(double, int);   // prototip functie
void main()
{
    int i;  double x;
    printf("\n Numarul real: ");
    scanf("%lf",&x);
    printf(" Puterea intreaga: ");
    scanf("%d",&i);
    printf("\n Rezultatul este %lf\n", power(x,i));
    getch();
}
// func.c
#include <math.h>
double power(double a, int b)
{
    int i, n; double p=1.; n=abs(b);
    for(i=1; i<=n; i++)
    p*=a;
    if(b<0) return (1/p);
    else return(p);
}
```

Cele două fişiere se pot compila separat, apoi se grupează într-un proiect, folosindu-se meniul Project al mediului.

Se deschide un fişier proiect (extensia .prj) care va conține numele celor două fişiere; dacă la nume nu se adaugă extensia se

consideră că e vorba de fişierele sursa (cu extensia .c); se pot include în proiect direct fişierele *.obj rezultate după compilare.

Pentru a se crea modulul executabil, se selectează "Make EXE file" din meniul Compile, şi are loc compilarea (dacă în proiect sunt date fişiere sursă), editarea de legături şi crearea fişierului EXE; modulul EXE va avea acelaşi nume cu fişierul proiect; se va selecta direct comanda Run pentru crearea fişierului EXE şi execuție.

Exemplul 3 - (varianta 1, incorectă)

Programul ilustrează conversia automată de tip de date, la apelul funcției după regulile de conversie cunoscute.

Programul calculează cel mai apropiat număr întreg, pentru un număr real.

```
#include <stdio.h>
#include <conio.h>
void main()
{
     float x;  printf("\n Introduceti numarul ");
     scanf("%f",&x);
     printf("\n Cel mai apropiat intreg este %d",
n_int(x));
     getch();
}
/* deoarece nu exista prototip, functia n_int este
considerata de tip int iar parametrul i este
convertit automat la double in definitia functiei,
parametrul fiind tip float, apare eroare la
     compilare*/

int n_int(float num)
{
if(num>0){
if(num-(int)num<0.5)
return((int)num); /* se returneaza o valoare
intreaga prin conversie  explicita */
else return((int)num+1);
      }
else  {
if((int)num-num<0.5)
     return((int)num);
else
     return((int)num-1);
     }
}
```

Sunt mai multe puncte de ieşire diferite, din funcția n_int dar toate cu acelaşi tip de date de ieşire.

Exemplul 3 - (varianta 2, corectă)

```
#include <stdio.h>
#include <conio.h>
void main()
{
  float x;
printf("\n Introduceti numarul ");
scanf("%f",&x);
printf("\n   Cel   mai   apropiat   int   este   %d",
n_int(x));
// se face conversia la double a lui x
}
int n_int(double num)
{
if(num>0)
     {
     if(num-(int)num<0.5) return((int)num);
     else return((int)num+1);
     }
else
     {
     if((int)num-num<0.5) return((int)num);
     else return((int)num-1);
     }
}
```

Exemplul 3 - (varianta 3, corectă)

```
#include <stdio.h>
int n_int(float);
// nu e obligatorie prezenta identificatorului de
//parametru, este suficient tipul;
void main()
{
float x;
printf("\n Introduceti numarul ");
scanf("%f",&x);
printf("\n   Cel   mai   apropiat   int   este   %d",
n_int(x));
}
int n_int(float num)
{
if(num>0)
     { if(num-(int)num<0.5) return((int)num);
        else return((int)num+1);
     }
else{
       if((int)num-num<0.5) return((int)num);
       else return((int)num-1);
     }
}
```

TEMĂ

1. Să se scrie un program care să citeasca numere reale introduse de la tastatură pâna la citirea numarului 0 şi să conţină funcţii care să calculeze numărul minim, numărul maxim şi media numerelor citite.

2. Să se scrie un program care să numere cuvintele dintr-un text de maxim 1000 de caractere de la tastatură. Cuvintele sunt separate prin spaţiu, caracterul *Tab* sau *Enter*. Încheierea citirii textului se face prin apasarea tastei *Esc* ('\0x1b').

3. Să se scrie un program care să calculeze şi să afişeze inversa unei matrice de 3 linii şi 3 coloane cu elemente întregi citite de la tastatură.

4.2. Transmiterea parametrilor la funcţii

Transmiterea parametrilor actuali, către o funcţie, se face:

- **prin valoare;**
- **prin referinţă;**

La transmiterea prin valoare, **se copiază** valoarea fiecărui parametru actual în spaţiul rezervat pentru parametrul formal corespunzător. Acest principiu de transfer face ca modificările făcute asupra parametrilor formali, în corpul funcţiei, să nu se reflecte în afara ei, adică **o funcţie nu poate modifica parametrii actuali cu care este apelată**, sau altfel spus, parametrii actuali sunt pentru funcţie nişte constante predefinite. Acest tip de transfer este predefinit în C şi nu poate fi modificat !

Transmiterea parametrilor actuali **prin referinţă,** se bazează pe tipul de date pointer (adresă). Parametrii actuali ai funcţiei sunt declaraţi în mod explicit, **pointeri**. În acest tip de transfer, o funcţie **primeşte indirect valorile actuale** (prin adresele lor) şi **le poate modifica** tot în mod indirect, prin intermediul pointerilor de acces, de care dispune.

Exemplu:

Pentru modificarea unor variabile, acestea nu se transferă direct ci prin pointeri, o funcţie neputând modifica direct valorile variabilelor din funcţia care o apelează.

```
#include <stdio.h>
#include <conio.h>
void main()
{
 void schimba(int *,int *); /* prototipul folosit
in main, local */
int x=2003, y=0;
printf("\n Valori initiale x=%d y=%d", x, y);
```

```
schimba(&x, &y);
printf("\n Valori finale   x=%d y=%d", x, y);
getch();
}
void schimba(int *a,int *b)
{
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}
```

Programul produce pe ecran:

```
Valori initiale: x=2003  y=0
Valori finale:   x=0  y=2003
```

În general, modul în care se transmit parametrii la funcții este dependent de implementare. Unele principii generale, sunt însă valabile la toate implementările. Parametrii sunt transmişi prin stivă. Ordinea de plasare a parametrilor actuali în stivă este, în general, de la dreapta la stânga, pentru lista de parametri din apelul funcției. Standardul nu garantează însă acest lucru, şi nici ordinea de evaluare a parametrilor. Un program corect conceput, nu trebuie să depindă de această ordine. Descărcarea stivei se face, de regulă, la majoritatea implementărilor de către funcția apelantă.

Regulile de mai sus sunt determinate de faptul că în C pot exista funcții cu număr variabil de parametri (aceeaşi funcție poate fi apelată o dată cu *m* parametri şi altă dată cu *n* parametri etc.).

Un exemplu clasic este funcția printf(), care are în mod obligatoriu numai primul parametru şi anume adresa şirului care descrie formatul, restul parametrior fiind în număr variabil şi pot chiar lipsi. Modul de plasare asigură că, totdeauna, funcția printf() va găsi în vârful stivei adresa şirului de caractere care descrie formatul. Analizând specificatorii de format din acest şir, funcția printf() ştie câți parametri să-şi extragă din stivă şi ce dimensiune are fiecare parametru (2 octeți, 4 octeți etc.).

De exemplu, la un apel de forma:

```
printf("%d %s\n", i, s);
```

în vârful stivei se va găsi adresa şirului constant `"%d %s\n"`.

Funcția preia acest parametru din vârful stivei, citeşte şirul şi extrage din stivă încă doi parametri: un întreg şi adresa unui şir de caractere. Evident, dacă parametrii actuali nu concordă (ca număr şi tip) cu specificatorii de format, rezultatele vor fi eronate. Este clar

acum, de ce este necesar ca stiva să fie descărcată de către programul apelant: acesta ştie câţi octeţi a încărcat în stivă la apelarea unei funcţii. De exemplu, un apel de forma:

```
printf("%d %d", n);
```

va produce o a doua tipărire, fără sens (se va tipări, ca întreg, conţinutul vârfului stivei înainte de apelul funcţiei), dar nu va duce la situaţii de blocare sau pierdere a controlului; revenirea în programul apelant se face corect, iar acesta descarcă stiva de asemenea corect.

Un apel de forma:

```
printf("%d", m, n);
```

va tipări numai variabila *m*, ignorând pe *n* iar restul acţiunilor se execută corect (revenire, descărcare stivă).

4.3. Biblioteci standard

Oferă o mare varietate de funcţii, grupate după tipul operaţiilor efectuate (familii de funcţii). Pentru fiecare familie, există un fişier **header** (***.h**), care conţine prototipurile funcţiilor.

Principalele fişiere header sunt:

stdio.h	- conţine funcţii de intrare / ieşire: printf(), scanf()
ctype.h	- conţine funcţii pentru testarea apartenenţei la clase de caractere: litere mari, litere mici, cifre, caractere tip spaţiu, de control etc.
string.h	- conţine funcţii pentru operaţii cu şiruri de caractere.
math.h	- conţine funcţii matematice: sin(), cos(), exp(), log(), pow()
stdlib.h	- conţine funcţii utilitare: de conversie, de generare a numerelor aleatoare, alocare eliberare memorie etc.
stdarg.h	- conţine macroinstrucţiuni pentru crearea de funcţii cu număr variabil de argumente
time.h	- conţine funcţii pentru gestionarea timpului (data şi ora): numele zilei, lunii, anul, minutul, secunda, reprezentarea datei;
limits.h float.h	- în aceste fişiere sunt definite constante care dau valorile minime şi maxime pentru tipurile de bază; aceste valori depind de implementare: Borland C, Turbo C, Turbo C++ etc.
conio.h	- conţine funcţii standard pentru gestionarea ecranului în modul text (25 linii şi 40 sau 80 de coloane)

graphics.h - conține funcții (peste 60) pentru gestionarea ecranului în modul grafic (de ex. VGA: 480 x 640 pixeli); numărul de pixeli depinde de tipul de monitor şi de adaptorul grafic.

Pentru a utiliza o funcție, aceasta trebuie apelată. Forma generală a apelării este:

```
nume_funcție(lista de parametri efectivi);
```

Cu excepția funcțiilor de tip *void*, orice funcție returnează o valoare care înlocuieşte numele funcției.

Funcțiile de tip *void* se numesc procedurale, deoarece realizează anumite prelucrări (operații) dar nu returnează o anumită valoare.

4.4. Intrări / ieşiri standard: <stdio.h>

Conceptul de bază pentru intrări / ieşiri standard este cel de pointer la fişier. În orice implementare este definit (cu typedef) un tip de structură, numit FILE.

În general, prin **fişier** înțelegem o colecție ordonată de elemente de acelaşi tip, numite înregistrări, organizate secvențial. Accesul la date se poate face numai prin intermediul unei variabile care schimbă date cu o singură înregistrare, la un moment dat. Orice fişier are un marcator *"început de fişier"* şi unul *"sfârşit de fişier"*.

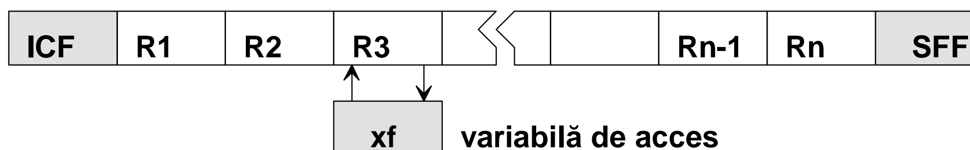


Fig.1 Structura secvențială a unui fişier

În C există o mare varietate de funcții pentru lucrul cu fişiere. Acestea necesită informații privind: numele fişierului, starea (deschis, închis), poziția curentă în fişier (componenta accesibilă) etc. Aceste informații sunt organizate într-o structură cu tip predefinit: FILE.

Orice operație cu fişiere, se face prin intermediul unei variabile de tip pointer la o structură de tip FILE. Pentru definirea acestei variabile se utilizează declarația:

```
FILE  *adr_fisier;
```

4.4.1. Deschiderea unui fişier

Pentru deschiderea unui fişier se utilizează funcţia fopen(). Ea returnează un pointer spre tipul FILE (ce indică adresa informaţiilor asociate fişierului) sau pointerul nul în caz de eroare. Prototipul este:

```
FILE *fopen(const char *nume_fisier, const char *mod);
```

unde:

- ♦ *nume_fisier* este un şir de caractere - numele fişierului, care depinde ca formă de sistemul de operare; de exemplu, la MS-DOS se scrie
  "c:\\user\\lista.txt"
- ♦ *mod* este un şir de caractere care descrie modul de acces:

"r"	deschide un fişier text pentru citire; fişierul trebuie să existe
"w"	deschide un fişier text pentru scriere; dacă fişierul există, conţinutul lui se pierde; dacă nu există un fişier cu numele specificat, el este creat.
"a"	deschide un fişier text pentru scriere prin adăugare la sfârşit; dacă fişierul nu există, el este creat.
"r+"	deschide un fişier text pentru actualizare, adică citire şi scriere
"w+"	deschide un fişier text pentru actualizare; dacă fişierul există, conţinutul lui se pierde; dacă nu există un fişier cu numele specificat, el este creat.
"a+"	deschide un fişier text pentru actualizare; fişierul poate fi citit dar scrierea se face prin adăugare la sfârşit;

Exemplu:

```
FILE  *pf ;
pf=fopen("c:\\user\\lista.txt", "w");
```

Funcţia fopen() creează un fişier pe discul c:\ cu numele lista.txt, pregătit pentru scriere date şi returnează un pointer de acces la fişier. Toate operaţiile cu fişierul creat se realizează prin intermediul variabilei de tip pointer pf, care este un identificator de fişier. Dacă operaţia de deschidere fişier nu se poate realiza (disc protejat la scriere, disc plin), funcţia fopen() returnează valoarea zero = NULL care are semnificaţia "nici o adresă". De aceea, se recomandă, ca la deschiderea unui fişier să se testeze dacă operaţia a decurs normal. Se

poate combina deschiderea cu testul de realizare, ca în exemplul următor:

```
FILE  *pf ;
if((pf=fopen("c:\\user\\lista.txt","w"))==NULL)
     {
     printf("Fisierul nu poate fi deschis");
     exit(1);
     }
```

Constanta NULL este predefinită în fişierul stddef.h şi corespunde unui pointer care nu indică o adresă (pointerul nul).

Funcţia exit() determină închiderea tuturor fişierelor deschise, încheierea execuţiei programului şi transferul controlului către sistemul de operare.

În tabelul de mai sus, modurile care conţin + (actualizare), permit scrierea şi citirea din acelaşi fişier, mai precis prin acelaşi pointer.

Dacă se pune sufixul b, se indică modul binar:
rb, wb, ab, r+b, w+b, a+b ;

Se poate pune explicit şi sufixul t, indicând explicit modul text:
rt, wt, at, r+t, w+t, a+t ;

Distincţia dintre modul binar şi text, depinde de sistemul de operare (la MS-DOS există diferenţe). Modul text este implicit.

Diferenţele se referă la două aspecte: tratarea caracterelor "\n" şi tratarea sfârşitului de fişier.

1. În modul text, gupul "\n" este convertit în două caractere CR (0xD) şi LF (0xA). La citire, are loc conversia inversă.

Dacă se scrie în modul text şi se citeşte în mod binar, nu se obţine aceeaşi informaţie. Se recomandă ca scrierea şi citirea să se realizeze cu acelaşi tip de funcţii: fprintf / fscanf, fputs / fgets sau fwrite / fread etc.

2. Tratarea sfârşitului de fişier. În ambele moduri (text şi binar), funcţiile de intrare/ieşire gestionează numărul de octeţi din fişier. De exemplu, funcţia fgetc(), la terminarea citirii fişierului, întoarce o valoare întreagă predefinită EOF. Acest întreg nu există fizic în fişier ci este generat de funcţia fgetc(). În modul text însă, există un caracter special, 0x1A, ca ultim caracter al fişierului (există fizic în fişier). La întâlnirea acestui caracter, fgetc() va întoarce EOF. Dacă un fişier conţine numere întregi sau reale, poate să apară un octet cu valoarea 26 (care se reprezintă intern prin 0x1A). La citirea în modul text, nu se

poate trece de acest octet, el fiind interpretat ca sfârşit fizic de fişier. Citirea trebuie făcută în acest caz în modul binar.

4.4.2. Închiderea unui fişier

Toate fişierele deschise trebuie închise înainte de terminarea programului. Dacă se omite închiderea fişierelor pot să apară incidente nedorite (distrugerea fişierelor, erori diverse). Funcţia care închide ceea ce s-a deschis cu fopen() este fclose(), cu prototipul:

```
int fclose(FILE *pf);
```

unde pf este pointerul de identificare a fişierului. Dacă valoarea pe care o returnează este zero, operaţia a decurs normal iar dacă nu este zero, operaţia de închidere nu s-a putut realiza (fişierul este inexistent sau este inaccesibil, când discul pe care se afla a fost scos din unitate).

4.4.3. Citirea şi scrierea unor caractere

- Funcţia putc() face scrierea unui caracter într-un fişier deschis cu fopen():

  ```
  int putc(int caracter,  FILE *pf);
  ```

- Funcţia getc() face citirea unui caracter dintr-u fişier deschis cu fopen():

  ```
  int getc(FILE *pf);
  ```

Pentru a citi tot fişierul se poate folosi secvenţa:

```
do
    caracter=getc(pf)
while (caracter!=EOF);
```

Funcţiile **fopen(),fclose(),putc(),getc()** reprezintă un nucleu minim de funcţii pentru lucrul cu fişiere.

5 Variabile globale şi locale
Domeniu de valabilitate

La dezvoltarea unui program este util ca problema să fie descompusă în părți mai simple care să fie tratate separat (pe module). Dezvoltarea modulară a programelor este avantajoasă şi conduce la programe structurate. Instrumentul prin care se poate aborda programarea structurată este funcția. Funcțiile care compun programul se pot afla în unul sau mai multe fişiere sursă, care pot fi compilate separat şi încărcate împreună. Este strict necesar săe transmită corect informațiile între modulele de program.

- ♦ Anumite informații (valori de variabile) sunt utilizate numai în interiorul funcției; ele se numesc **variabile locale** sau **interne** şi sunt definite şi vizibile în interiorul funcției.
- ♦ Alte informații sunt necesare pentru mai multe funcții şi se numesc **globale**; ele se definesc în afara oricărei funcții iar funcțiile au acces la acestea prin intermediul numelui variabilei. O variabilă globală poate fi accesată într-un singur fişier (în care este definită) sau în mai multe fişiere.
- ♦ Variabilele globale, fiind accesibile mai multor funcții, pot transmite informații între funcții, fiind utilizate ca parametri actuali ai funcțiilor.

5.1. Variabile locale

Sunt definite în interiorul unei funcții, pot fi utilizate şi sunt vizibile numai în interiorul funcției. Nici o altă funcție nu are acces la variabilele locale, deoarece la ieşirea din funcție, valorile lor sunt distruse (se eliberează memoria ocupată de acestea). Nu există nici o legătură între variabilele cu acelaşi nume din funcții diferite.

Limbajul C nu permite să se definească o funcție în interiorul altei funcții.

Cea mai mică unitate de program, în care se poate declara o variabilă locală, este blocul = o instrucțiune compusă, adică o secvență de instrucțiuni delimitată de acolade. Limbajul C permite ca într-o instrucțiune compusă să se introducă declarații. O funcție poate conține mai multe blocuri.

Durata de viață a unei variabile este intervalul de timp în care este păstrată în memorie.

- Unele variabile pot rămâne în memorie pe toată durata execuției programului, la aceeaşi adresă; acestea sunt definite cu cuvântul cheie STATIC - **variabile statice sau permanente**.
- Alte variabile sunt definite şi primesc spațiu de memorie numai când se execută blocul sau modulul în care au fost definite. La încheierea execuției blocului sau modulului, se eliberează automat memoria alocată şi variabilele dispar. Dacă se revine în blocul respectiv, variabilele primesc din nou spațiu de memorie, eventual la alte adrese. Variabilele de acest tip, sunt numite **variabile cu alocare automată** a memoriei. Exemplu:

```
float functie()
{
int k ;
static int a[]={1, 2, 3, 4};
. . . . . . . . . . . . . .
}
```

Variabilele k şi a[] sunt locale; domeniul lor de valabilitate este blocul în care au fost definite.

Tabloul a[] păstrează adresa de memorie pe toată durata execuției programului.

Variabila k este cu alocare automată a memoriei.

5.1.1. Inițializarea variabilelor locale

La fiecare apel, variabilele cu alocare automată trebuie inițializate. Dacă nu se face inițializarea, acestea vor avea valori întâmplătoare, care se află în spațiul de memorie alocat. Variabilele statice se inițializează o singură dată. Exemple:

```
void incrementare(void)
{
int i=1;
static int k=1;
i++ ; k++ ;
printf("i=%d \t k=%d \n", i, k);
}
int main(void)
incrementare();
incrementare();
incrementare();
}
```

Rezultatul execuției programului este:

```
i=2      k=2
i=2      k=3
```

```
i=2       k=4
```

Variabila i este cu alocare automată; ea este inițializată la fiecare apel al funcției cu valoarea i=1.

Variabila k este de tip **static**, permanentă; se inițializează o singură dată iar valoarea curentă se păstrează la aceeaşi adresă. Variabilele statice dacă nu sunt inițializate, primesc valoarea zero.

Variabilele statice interne (locale) oferă posibilitatea păstrării unor infoemații despre funcție (de exemplu, de câte ori a fost apelată).

5.1.2. Variabile globale

După domeniul de valabilitate, variabilele pot fi ierarhizate astfel:

- variabile cu acțiune în bloc (locale);
- variabile cu acțiune în funcție (locale);
- variabile cu acțiune în fişierul sursă;
- variabile cu acțiune extinsă la întregul program.

Pentru ca o variabilă să fie valabilă în tot fişierul, trebuie declarată în afara oricărei funcții, precedată de cuvântul STATIC. Dacă fişierul conține mai multe funcții, variabila este vizibilă în toate funcțiile care urmează declarației.

Prin convenție, numele unei variabile globale se scrie cu literă mare.

Sfera de acțiune cea mai mare pentru o variabilă este întregul program; o asemenea variabilă este vizibilă în toate funcțiile aflate atât din fişierul în care a fost definită, cât şi din alte fişiere.

Pentru a crea o variabilă globală ea se declară în afara oricărei funcții, fără cuvântul cheie STATIC.

```
float x ;     //var. globala in tot programul
static float y ; //var. globala in fisierul curent
int main(void)
{
. . . . . . .
}
```

Variabilele globale măresc complexitatea unui program deoarece ele pot fi modificate de orice funcție. Se recomandă utilizarea lor numai când este strict necesar.

6 Structuri de date: LISTE

6.1. 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 acelaşi tip, între care există o relație de ordine determinată de poziția relativă a elementelor; un element din listă conține o informație propriu-zisă şi o informație 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 înscrişi 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 conține informații cu privire la poziția sa în listă (predecesorul şi 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	Maraşescu Gh. Liviu Costin	6/M	7.5	10	9.56	8.95
	. . . . .	. . .	. . .	. . .	. . .	. . . .

Fiecare informație 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 sfârşitul listei;

- inserarea de noi elemente în interiorul listei;
- ştergerea unui element din listă;
- schimbarea poziției unui element (modificarea unei înregistrări greşite);
- 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 îndeplineşte 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.

6.2. 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 şi se depun în locații succesive de memorie.

Avantaje: timpul de acces este acelaşi 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 şi ştergerea unui element în/din interiorul listei implică deplasarea tuturor elementelor pe alte poziții decât cele inițiale.

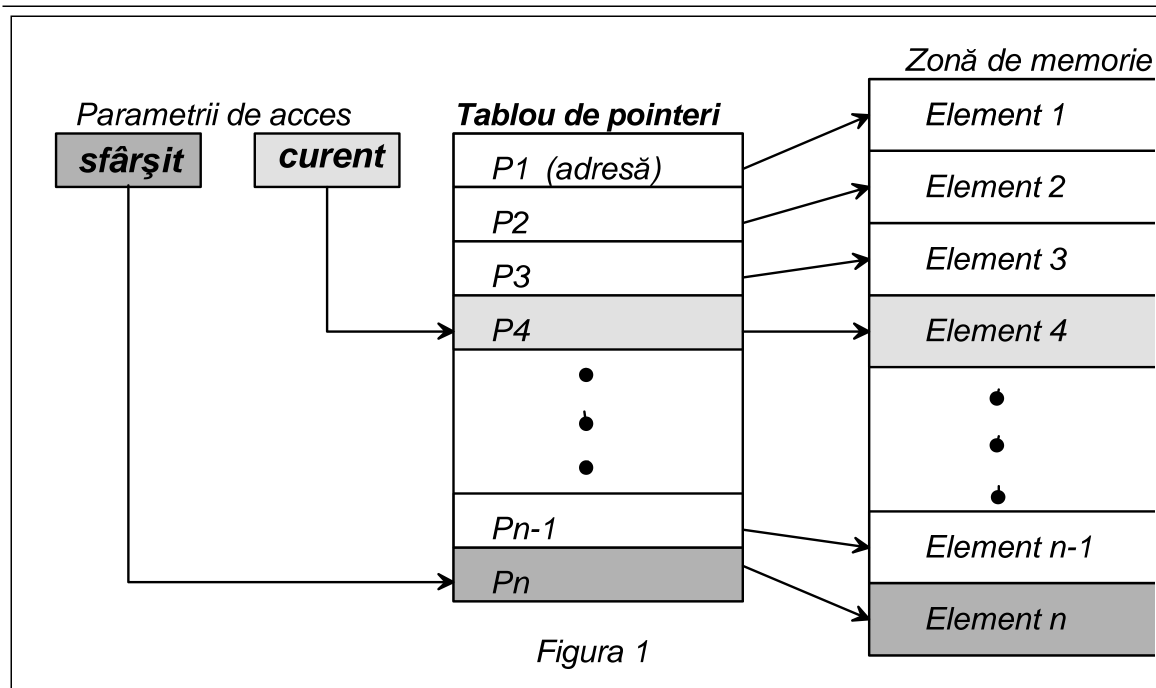


Figura 1

În cazul unei alocări statice (mai eficiente) a unei liste (fig.1), se construieşte un tablou de pointeri care permit accesul la informaţia utilă (elementele listei). Accesul la elemente se realizează prin doi parametri: *curent* (indică poziţia în tablou a adresei elementului curent) şi *sfârşit* (indică poziţia în tablou a adresei ultimului element).

În cazul **reprezentării dinamice,** prin liste înlănţuite, 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 informaţiei utile din fiecare element.

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

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

O structură dinamică de listă conţine 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 (*sfârşit*), de tip ponter.

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

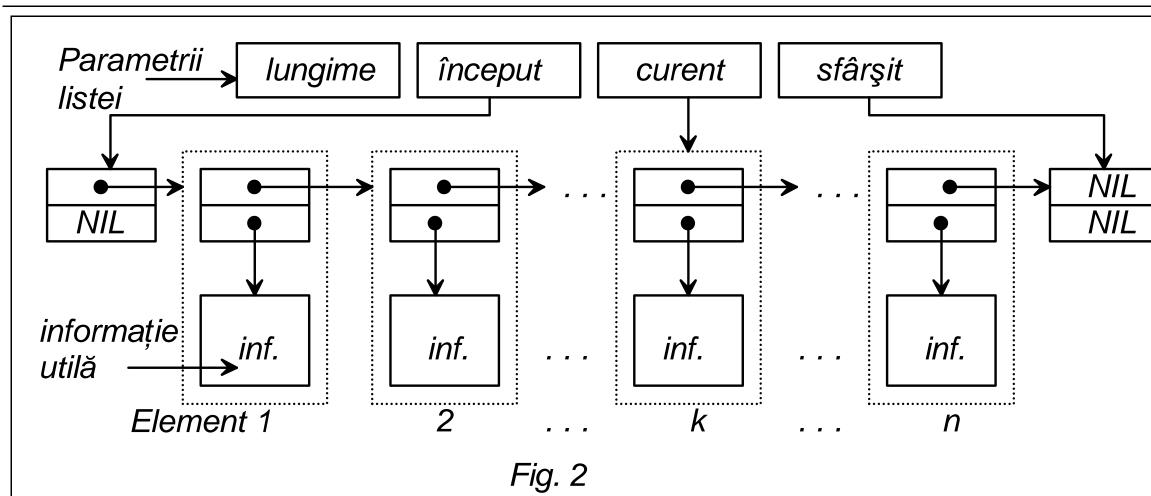


Fig. 2

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

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

Căutarea unui element se face prin parcurgerea secvenţială a listei, până când pointerul curent ia valoarea dorită sau o componentă a informaţiei corespunde unei condiţii.

Lista din fig.2 este o listă simplu înlănţuită - 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 obţine astfel o listă dublu înlănţuită (fig.3), care prezintă avantaje suplimentare privind operaţiile curente prin micşorarea 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 dublu înlănţuite se conferă listei o structură circulară şi se pot elimina elementele "false" de marcare a capetelor (santinele).

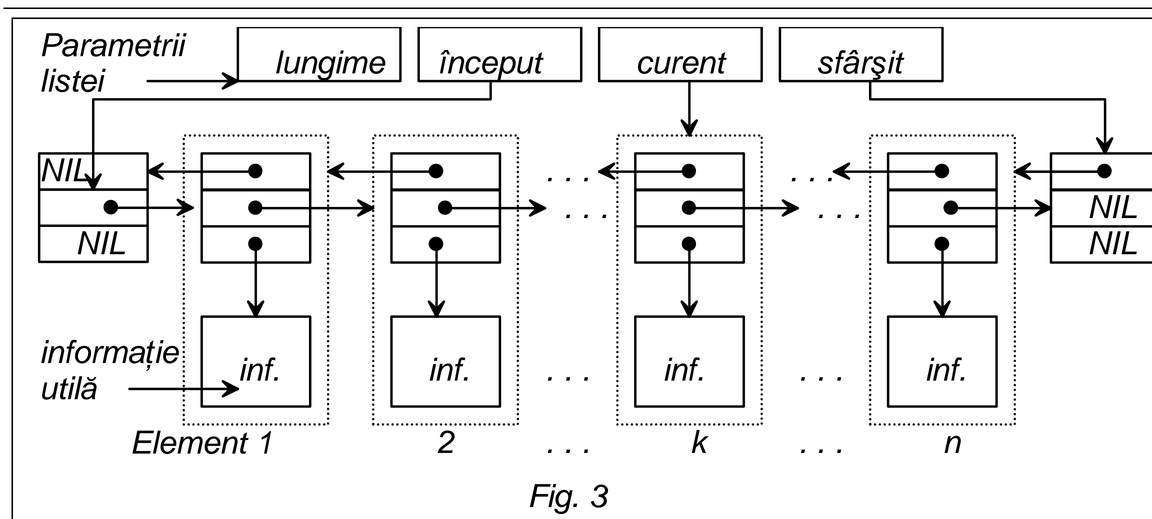


Fig. 3

6.3. Implementarea operaţiilor de bază pentru liste simplu înlănţuite

Presupunem că s-a definit un tip de date DATA, care este specific informaţiei dintr-un element al listei şi depinde de aplicaţie (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 conţine NULL, indicând că nu mai urmează nici un element.

Se observă că o listă simplu înlănţuită 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 adiţionale de tip stivă.

Totuşi, 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 tipăreşte şirul primit şi 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` şi `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 şi conversia la tipul LINK şi 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 şi se memorează în **t** şi **p**, apoi se crează succesiv elemente, incrementând pointerul **tab** şi avansând pointerul **p** la următorul element. În final se întoarce **t**.

O a doua variantă, mai clară, **arr_to_list_r()**, foloseşte 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) şi un pointer întors de aceeaşi funcție **arr_to_list_r()**, apelată cu parametrii **tab+1** şi **--n**. Ar fi o greşeală 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));
	}
```

6.4. Tipărirea unei liste

Se utilizează o funcție recursivă pentru tipărire, care nu este dependentă de tipul DATA (acest tip depinde de aplicație). Funcția tipăreşte 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 funcţiei 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 funcţiilor de parcurgere a listelor liniare în variantă iterativă şi 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 funcţie `prelucrare()`, care realizează operaţiile dorite asupra datelor din fiecare element; funcţia depinde de tipul concret al datelor şi deci este dependentă de aplicaţie.

6.5. Operaţii asupra listelor liniare

Pentru o listă liniară nu se poate implementa un algoritm de căutare binară (prin divizarea listei şi căutarea în secţiuni) ci numai un algoritm de căutare secvenţială. Funcţia de mai jos caută un element dat într-o listă, întorcând un pointer la primul element găsit, sau NULL dacă lista nu conţine elementul dat. Se utilizează funcţia cmp() de

comparare a două elemente, întorcând un întreg negativ, zero sau pozitiv, la fel ca strcmp(). O asemenea funcţie este strict necesară, deoarece în C, structurile nu se pot compara.

```
int cmp(DATA a, DATA b);
LINK list_src_i(LINK t, DATA x)
{
	while(t!=NULL && cmp(t->data, x)!=0)
		t=t->next;
	return t;
}

LINK list_src_r(LINK t, DATA x)
{
	if(t==NULL || cmp(t->data, x)==0)
		return t;
	else
		return list_src_r(t->next,x);
}
```

Ordinea în care apar condiţiile conectate prin || sau && este esenţială: dacă t este NULL, comparaţia dintre t->data (care nu există) şi x nu trebuie făcută.

Avantajul esenţial al structurilor înlănţuite faţă de cele secvenţiale (tablouri) apare la operaţiile de inserare şi ştergere de elemente.

La un tablou unidimensional, dacă se şterge un element, toate elementele care urmează trebuie deplasate cu o poziţie spre stânga, pentru a păstra forma continuă a tabloului (fără "goluri"). Similar, dacă se inserează un element, toate elementele care urmează trebuie deplasate spre dreapta cu o poziţie, pentru a se realiza un spaţiu pentru noul element.

La structurile înlănţuite, operaţiile de inserare şi ştergere se realizează fără schimbarea poziţiei celorlalte elemente; se modifică doar câmpurile de adresă care asigură "înlănţuirea".

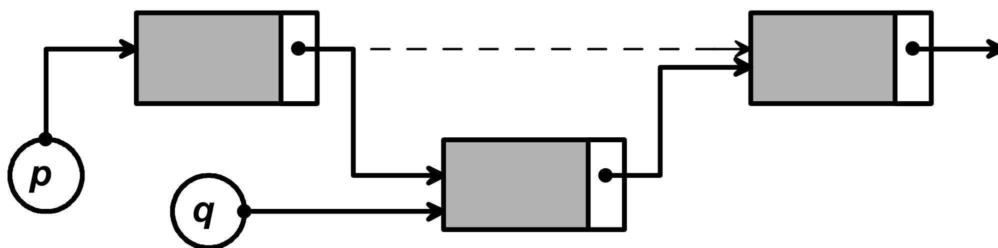


Fig.4 Inserare element

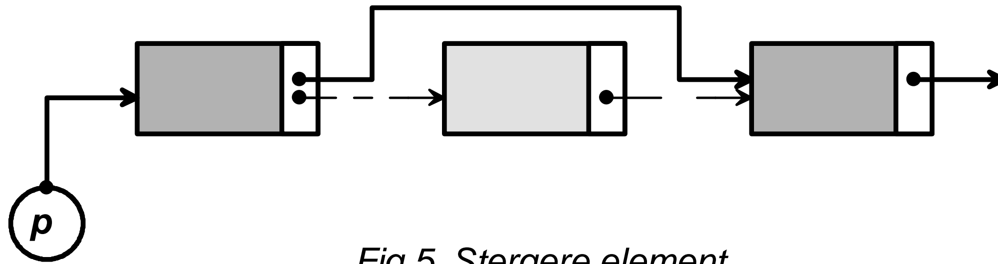


Fig.5 Ştergere element

6.5.1. Inserarea unui element după un element dat

Se consideră că pointerul **p** adresează elementul din listă **după care** se va face inserarea iar **q** adresează noul element, care **se inserează**.

Dacă p=NULL nu avem după cine insera iar dacă avem q=NULL, nu avem ce insera. Dacă p este adresa ultimului element din listă (adică p->next = NULL), se conectează noul element (q) la acesta şi atunci q devine ultimul element.

```
void ins_dupa(LINK p, LINK q)
{
     if(q==NULL || p==NULL)
          return;
     q->next=p->next;
     p->next=q;
}
```

Inserarea se face simplu, prin două atribuiri de pointeri (fig. 4).

Trebuie observat un fapt esenţial, că în instrucţiunile de atribuire expresiile din stânga sunt toate obţinute cu operatorul ->, deci se modifică efectiv lista. O atribuire de forma p= . . . într-o asemenea funcţie, unde p este un parametru formal, nu are nici un efect în exteriorul funcţiei (transfer prin valoare).

6.5.2. Ştergerea unui element de după un element dat

```
void del_dupa(LINK p)
{
     LINK q
     if(p==NULL || p->next==NULL)
          return;
     q=p->next;
     p->next=q->next;
     free(q);
```

```
}
```

Dacă p este NULL (nu există element) sau p->next este NULL (adică nu există următorul element), nu avem nimic de făcut. Dacă elementul adresat cu p există şi nu este ultimul, se salvează adresa de legătură în p->next şi apoi se eliberează memoria ocupată de elementul care se şterge, adresat temporar cu q (fig.5).

6.5.3. Ştergerea tuturor elementelor unei liste

```
void del_list(LINK t)
{
     LINK p;
     while(t!=NULL) {
          p=t;
     t=t->next;
     free(p);       }
}
```

După eliberarea spaţiului indicat de un pointer, acel spaţiu nu mai poate fi folosit. Ca urmare, se salvează t într-o variabilă temporară p, se atribuie lui t adresa următorului element şi abia apoi se distruge elementul curent. Dacă s-ar scrie direct free(t), nu mai avem acces la următorul element.

7 Structuri de date: ARBORI

Organizarea liniară de tip listă este adecvată pentru aplicațiile în care datele (elementele din listă) formează o mulțime omogenă şi deci se află pe acelaşi nivel. În multe aplicații, este strict necesară *organizarea ierarhică* pentru studiul şi rezolvarea problemelor.

Exemple:

- planificarea meciurilor într-un turneu sportiv de tip cupă (fig.1);
- organizarea ierarhică a fişierelor în memoria calculatorului;
- structura de conducere a unei întreprinderi, minister etc.;
- organizarea administrativă a unei țări.

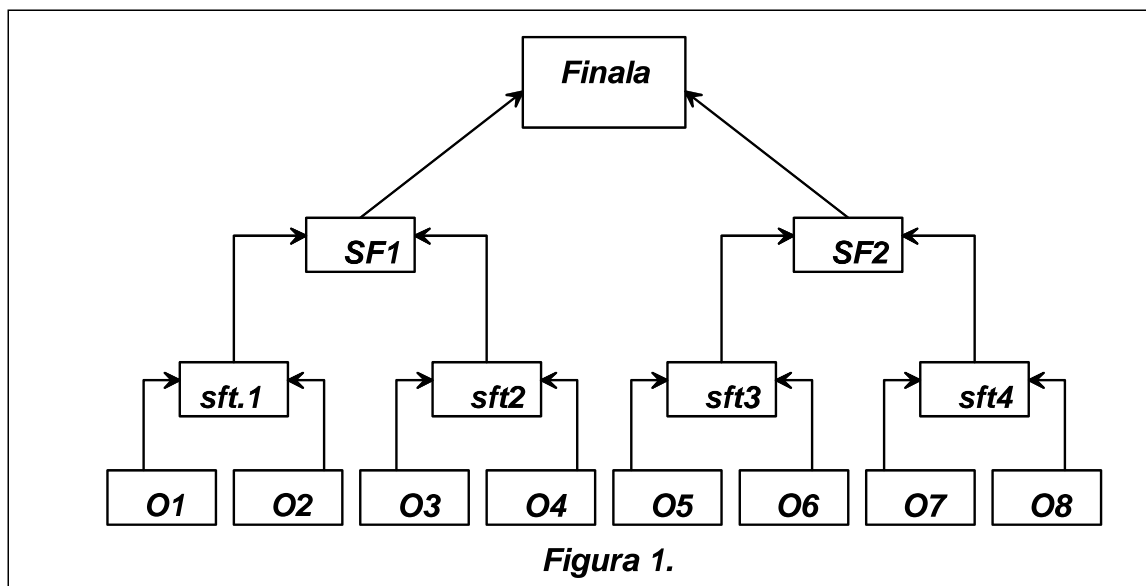


Figura 1.

În unele aplicații, implementarea structurilor ierarhice în programele de calculator este singura cale de rezolvare.

Un ***arbore*** este o structură ramificată formată dintr-un nod A (rădăcina) şi un număr finit de arbori (subarbori) ai lui A.

- orice nod din arbore este rădăcină pentru un subarbore iar orice arbore poate deveni subarbore;

- doi subarbori pot fi în relație *de incluziune*, când un subarbore este inclus în celălalt sau *de excluziune*, când nu au noduri comune.

Definiții:

nod = punct în care se întâlnesc doi subarbori;

nivel = numărul de noduri parcurse de la rădăcină până la un nod;

rădăcină = nivel 1;

descendent = primul nod al unui subarbore;

nod terminal = nod fără descendenți;

înălțimea unui nod = numărul maxim de niveluri;

arbore binar = arbore în care toate nodurile au 2 descendenţi;

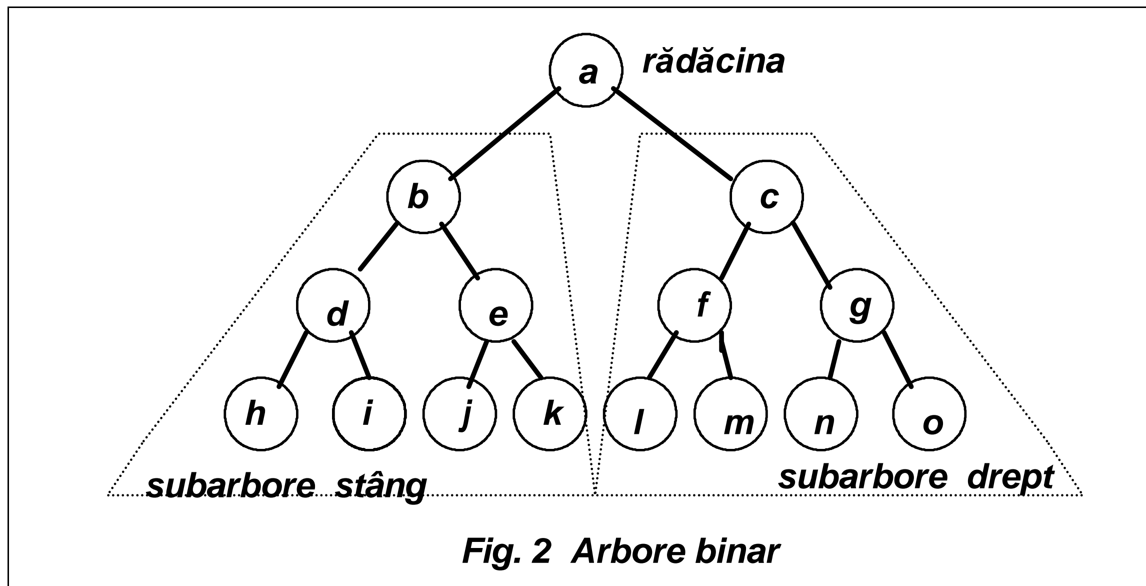


Fig. 2 Arbore binar

Orice arbore binar are doi subarbori: stâng şi drept. Arborii binari pot fi arbori de căutare şi arbori de selecţie; ei se caracterizează prin faptul că fiecare nod areo*"cheie"* reprezentată printr-o informaţie specifică de identificare a nodului. Cheia permite alegerea unuia din cei doi subarbori în funcţie de o decizie tip *"mai mic"*, *"mai mare", etc.*

Un arbore este *echilibrat* dacă pentru orice subarbore diferenţa dintre înalţimile subarborilor săi este cel mult 1.

7.1. Parcurgerea arborilor

Prin parcurgerea arborelui înţelegem inspectarea (vizitarea) fiecărui nod şi prelucrarea informaţiei specifice lui. Pentru un arbore dat, corespunzător unei anumite aplicaţii, se impune o anumită ordine de parcurgere.

În programe se utilizează algoritmi de parcurgere sistematică a arborilor implementaţi sub formă de proceduri. Procedurile eficiente de parcurgere generează liste dinamice cu nodurile ce urmează a fi examinate, care sunt reactualizate după examinarea fiecărui nod; când lista devine vidă operaţia de parcurgere se încheie (au fost examinate toate nodurile). Posibilităţile de parcurgere sunt:

1. ***În lăţime***, examinând nodurile pe niveluri, în acelaşi sens.
2. ***În adâncime***: de la rădăcină spre nodurile terminale sau invers;

În cazul parcurgerii *în lăţime* algoritmul conţine operaţiile:

- se examinează nodul rădăcină şi se formează lista (coada) descen- denților săi într-o anumită ordine (exemplu: de la stânga la dreapta) ;

- se examinează primul nod din coada de aşteptare şi se adaugă la coadă descendenții săi în ordinea stabilită;

- se repetă operația precedentă până când coada devine vidă.

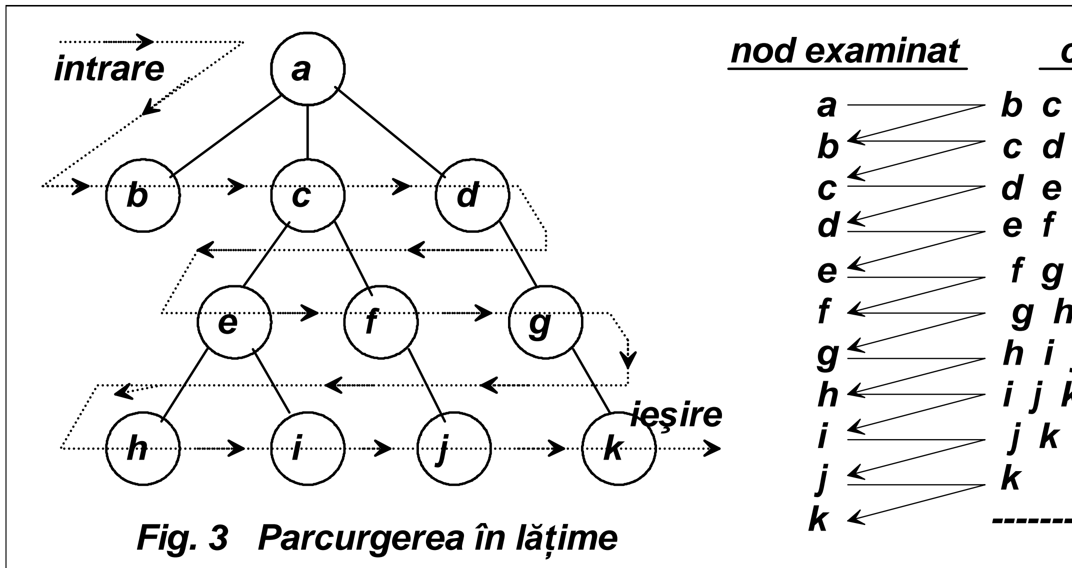


Fig. 3 Parcurgerea în lățime

În cazul parcurgerii în *adâncime*, algoritmul conține operațiile:

- se examinează nodul rădăcină şi se formează lista (stiva) descen- denților săi într-o anumită ordine (exemplu: de la stânga la dreapta) ;

- se examinează primul nod din stiva de aşteptare şi se introduc în stivă descendenții nodului curent până la cei terminali (întregul subarbore);

- se repetă operația precedentă până când stiva devine vidă.

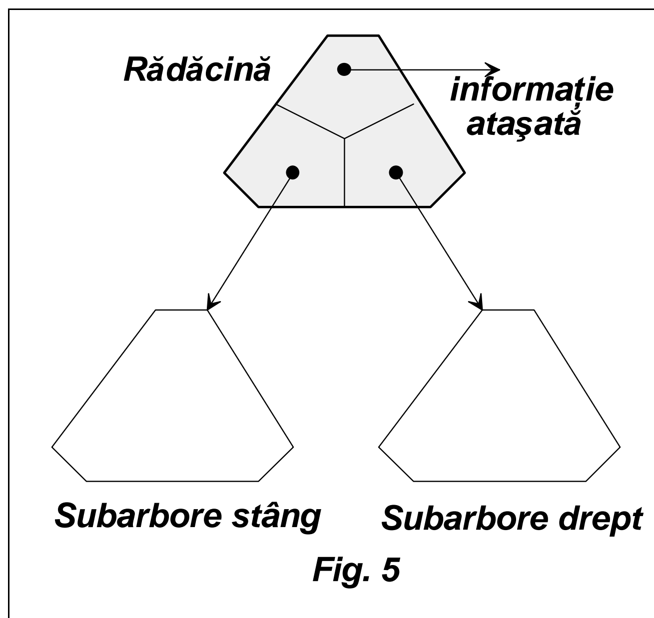


Fig. 5

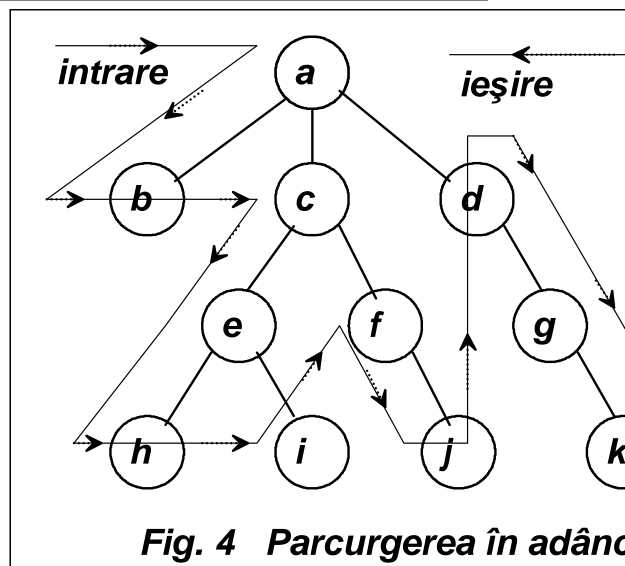


Fig. 4 Parcurgerea în adânc

7.2. Implementarea arborilor binari

Un arbore binar are în general 3 componente: nodul rădăcină, subarbore stâng, subarbore drept.

Cazul limită este un arbore vid, reprezentat printr-o constantă simbolică "*ArboreVid*". Tipul informației atașate nodurilor dintr-un arbore este specificat în fiecare aplicație.

De aceea vom considera că informația din fiecare nod este adresată indirect prin intermediul unui pointer (fig. 5). Cei doi subarbori sunt de asemenea adresați indirect, fie prin pointeri (fig. 5), fie prin intermediul indicilor de tablou în cazul implementării statice.

Operațiile fundamentale specifice arborilor binari sunt:

a) Operații care furnizează informații despre un arbore:

- ♦ testarea existenței unui arbore (dacă nu există, este vid);
- ♦ furnizarea adresei către un subarbore;
- ♦ evaluarea numărului de noduri;

b) Operații care modifică structura unui arbore:

- ♦ inserarea unui nod;
- ♦ ştergerea unui nod;

♦ modificarea informației dintr-un nod;
♦ ştergerea arborelui (eliberarea spațiului de memorie).

Observații:

1. Operația de căutare nu este inclusă în operațiile fundamentale pentru că aceasta poate avea diferite variante, în funcție de tipul arborelui prelucrat.

2. Arborii sunt utilizați (în mod frecvent) pentru a memora informații care se modifică. De aceea, pentru reprezentarea lor se preferă soluția dinamică, bazată pe utilizarea pointerilor. În implementarea arborilor se utilizează o metodă asemănătoare cu cea de la liste, prin construirea unui fişier cu operațiile fundamentale şi includerea acestuia în programele de aplicații.

3. În afara operațiilor prezentate anterior, pot fi definite şi altele, necesare în diferite aplicații.

Deoarece numărul de legături ale unui nod este cunoscut (cel mult 2), se utilizează doi pointeri pentru nodurile din stânga şi din dreapta, adică rădăcinile celor doi subarbori. Un subarbore vid, va fi reprezentat prin pointerul NULL.

Presupunând că s-a definit un tip de date DATA pentru câmpul de informație, următoarea structură C defineşte tipurile de date NODE şi BTREE (referință la NODE):

```
struct node {
    DATA d;
    struct node *left;
    struct node *right;
typedef struct node NODE, *BTREE;
```

O primă funcție este cea care construieşte un nod.

```
    BTREE new_node(DATA d)
    {
    BTREE t=(BTREE) malloc(sizeof(NOD));
    if(t==NULL) err_exit("new_node: Eroare de
alocare");
    t->d=d;
    t->left=t->right=NULL;
    return t;
    }
```

Este utilă în aplicații o funcție constructor de nod, care primeşte atât câmpul de date cât şi cele două câmpuri de legătură:

```
BTREE init_node(DATA d, BTREE left, BTREE right)
{
     BTREE t=new_node(d);
     t->left=left;
     t->right=right;
     return t;
}
```

Parcurgerea arborilor binari se face în trei moduri :

- ♦ În ***pre**ordine* (**R**SD - rădăcină, stânga, dreapta), adică se vizitează rădăcina, subarborele stâng şi apoi subarborele drept; cei doi subarbori vor fi tratați ca arbori în procesul de parcurgere, aplicând acelaşi algoritm - RSD.
- ♦ În ***in**ordine* (S**R**D - stânga, rădăcină, dreapta), adică se parcurge subarborele stâng, se vizitează rădăcina şi apoi subarborele drept; cei doi subarbori vor fi tratați ca arbori în procesul de parcurgere, aplicând acelaşi algoritm - SRD.
- ♦ În ***post**ordine* (SD**R** - stânga, dreapta, rădăcină), adică se parcurge subarborele drept, cel stâng şi apoi se vizitează rădăcina; cei doi subarbori vor fi tratați ca arbori în procesul de parcurgere, aplicând acelaşi algoritm - SDR.

Pentru arborele binar din fig. 2, aplicând metodele de parcurgere de mai sus, rezultă următoarele şiruri de noduri:

- preordine: **a b d h i e j k c f l m g n o** ;
- inordine: **h d i b j e k a l f m c n g o** ;
- postordine: **h i d j e k b l m f n o g c a** ;

Definițiile celor trei metode de parcurgere sunt recursive, adică se aplică la fel pentru orice subarbore; acest fapt sugerează o implementare recursivă a funcțiilor de parcurgere. Considerăm o funcție, visit() care examinează informația dintr-un nod, cu prototipul:

```
void visit(BTREE);
```

Cele trei metode de parcurgere, se implementează în mod natural, astfel:

```
void par_rsd(BTREE t)
{
     if(t!=NULL) {
          visit(t);
```

```
            par_rsd(t->left);
            par_rsd(t->right);
            }
    }
     ___________________________________________
        void par_srd(BTREE t)
    {
        if(t!=NULL) {
            par_srd(t->left);
            visit(t);
            par_srd(t->right);
            }
    }

    void par_sdr(BTREE t)
    {
        if(t!=NULL) {
            par_sdr(t->left);
            par_sdr(t->right);
            visit(t);
            }
    }
    ___________________________________________
```

Pentru afişarea nodurilor parcurse (prin nume nod), sub formă indentată, se poate folosi o funcţie `print_tree()`, recursivă, care scrie un număr variabil de spaţii, înainte de afişarea unui nod, în funcţie de nivelul nodului respectiv.

```
    void print_t(BTREE t, int n)// functie ajutatoare
    {
        int i;
        for(i=0; i<n; i++)
        printf(" ");
        if(t!=NULL) {
            print_node(t->d); // functie de afisare nod
            print_t(t->left, n+1);
            print_t(t->right, n+1);
        }
        else
            printf("Arbore vid!");
    }
    void print_tree(BTREE t)
    {
        print_t(t, 0);
        putchar('\n');
    }
```

7.3. Arbori de căutare

Arborii de căutare sunt arbori binari în care există o relație de ordine pe mulțimea elementelor de tip DATA, acestea fiind câmpurile de date din noduri. Categoria arborilor de căutare este caracterizată de următoarele proprietăți esențiale:

- ♦ câmpurile de date din noduri conțin valori distincte;
- ♦ un arbore vid este, prin convenție, arbore de căutare;
- ♦ rădăcina unui arbore de căutare conține o valoare mai mare decât toate valorile din subarborele stâng şi mai mică decât toate valorile din subarborele drept;
- ♦ subarborii stâng şi drept sunt arbori de căutare.

O importantă proprietate a arborilor de căutare, care decurge din definiție, este aceea că parcurgerea în inordine produce o listă de elemente sortate crescător, în raport cu câmpul DATA.

Această proprietate este utilizată de algoritmii de sortare internă.

A doua proprietate importantă a arborilor de căutare, care dă chiar denumirea lor, este posibilitatea implementării unui algoritm eficient de căutare a unui element, pe baza valorii din câmpul DATA: se compară elementul căutat cu data din rădăcină; apar trei posibilități:

- ♦ acestea coincid - elementul a fost găsit;
- ♦ elementul căutat este mai mic decât data din rădăcină - atunci căutarea continuă în subarborele stâng;
- ♦ elementul căutat este mai mare decât data din rădăcină - atunci căutarea continuă în subarborele drept;

Presupunem că s-a definit o funcție de comparație (care este dependentă de tipul DATA, deci de aplicație):

```
int cmp_data(DATA a, DATA b);
```

care returnează o valoare negativă dacă **a<b**, zero dacă **a=b** sau o valoare pozitivă, dacă **a>b**.

Funcţia de căutare, în variantă recursivă, se poate scrie astfel:

```
BTREE cauta (BTREE t, DATA x)
{
int y;
if(t==NULL || (y=cmp(x, t-.d))==0)
     return t;
t=(y<0) ? t->left: t->right;
return cauta(t,x);
}
```

Funcţia `cauta()` primeşte un pointer `t` la rădăcina arborelui şi o valoare `x` şi returnează pointerul la nodul găsit (care conține `x`) sau `NULL` dacă valoarea `x` nu există în arborele `t`.

Dacă arborele de căutare este echilibrat (adică fiecare subarbore are aproximativ acelaşi număr de noduri în stânga şi dreapta), numărul de comparaţii necesare pentru a găsi o anumită valoare este de ordinul lui $\log_2(n)$, unde n este numărul de noduri.

Dacă arborele de căutare este neechilibrat, atunci căutarea este asemănătoare celei dintr-o listă simplu înlănțuită, caz în care numărul de comparaţii necesare, poate ajunge la n.

Petru a elimina restricţia ca nodurile să aibă elemente distincte, se modifică tipul `DATA`, introducând, pe lângă valoarea propriu-zisă, un întreg care reprezintă numărul de repetări ale valorii respective. Definim tipul informaţiei din nod ca fiind `KEY` iar tipul `DATA` devine:

```
typedef struct {KEY key; int count;} DATA;
```

Definiţia tipurilor `NODE` şi `BTREE` nu se schimbă dar funcţia de construcţie a unui nod se modifică, primind acum o valoare de tip `KEY` şi inițializând cu 1 câmpul count al nodului creat.

Inserarea unui element într-un arbore de căutare

Din definiția arborelui de căutare, rezultă că adăugarea unui nod se face în funcţie de valoarea câmpului de informaţie; pentru adăugarea unui nod, este necesar aşadar să se determine locul în care se inserează. Se fac deci două operaţii: căutarea locului şi inserarea propriu-zisă.

Algoritmul de bază utilizat în construcţia arborilor de căutare este numit *căutare şi inserare*.

Se dă arborele t şi un element x. Dacă x este în t, se incrementează câmpul count corespunzător valorii x; dacă nu, x este inserat în t într-un nod terminal, astfel încât arborele să rămână *de căutare*.

Considerăm că s-a definit o funcţie de comparare, `cmp_key()` care returnează o valoare negativă, zero sau o valoare pozitivă, după cum relaţia dintre argumente este `a<b, a=b, a>b`.

```
int cmp_key(KEY a, KEY b);
```

O implementare recursivă a algoritmului de căuare şi inserare este:

```
BTREE cauta_si_insr(BTREE t, KEY x)
{
     int c;
     if(t==NULL) return new_node(x);
if((c=cmp_key(x, (t->d).key))<0)
     t->left=cauta_si_insr(t->left, x);
else
     if(c>0)
          t->right=cauta_si_insr(t->right, x);
     else  (t->d).count++;
return t;
}
```

Funcţia primeşte pointerul t la rădăcina arborelui şi cheia x. Dacă s-a ajuns la un nod terminal sau arborele este vid, rezultă că x nu este în arbore şi ca urmare se construieşte un nod nou cu valoarea x.

Dacă `(t!=NULL)`, se aplică algoritmul de căutare, apelând recursiv funcţia cu subarborele stâng, respectiv drept. Dacă se identifică valoarea x în arbore, atunci se incrementează câmpul count al nodului în care s-a găsit x. În final se returnează pointerul la arborele modificat.

7.3.1. Sortarea unui sir, bazată pe arbore de căutare

Se construieşte un arbore de căutare cu elementele şirului de date, considerate ca elemente ale tabloului tab[]; apoi se parcurge în inordine arborele creat returnând elementele sale, în ordine, în acelaşi tablou.

```
void sort(KEY tab[], int n)
{
     BTREE t=NULL;
     int i
     for(i=0; i<n; i++)
t=cauta_si_insr(t, tab[i]);
index=0;
parcurge(t, tab);
delete(t);
}
```

Funcţia parcurge() preia elementele din arborele t şi le plasează în tabloul v[]:

```
static int index;
void parcurge(BTREE t, KEY v[])
{
     int j;
     if(t!=NULL) {
     parcurge(t->left, v);
     for(j=0; j<(t->d).count; j++)
          v[index++]=(t->d).key;
     parcurge(t->right, v);
     }
}
```

Se parcurge t în inordine, copiind în v cheia din rădăcină, de atâtea ori cât este valoarea lui count. Este esenţială declararea indicelui de tablou, index, în clasa `static external` şi iniţializarea lui cu 0, înainte de a apela funcţia `parcurge()` în cadrul funcţiei `sort()`; se asigură astfel incrementarea sa corectă pe parcursul apelurilor recursive ale funcţiei `parcurge()`.

Funcţia `delete()` (utilizată în `sort()`), eliberează spaţiul de memorie alocat arborelui t, după copierea datelor sortate, fiind şi un exemplu de parcurgere în postordine:

```
void delete(BTREE t)
{
     if(t!=NULL) {
          delete(t->left);
          delete(t->right);
          free(t);
     }
}
```

Apelul funcţiei free() trebuie să fie după apelurile recursive, deoarece, după execuţia lui free(t), variabila t nu mai există.

Acest algoritm de sortare, combinat cu implementarea arborelui prin alocare secvenţială şi anume chiar în tabloul care trebuie sortat, duce la o metodă eficientă de sortare internă.

8 Tehnici de sortare

8.1. Introducere, definiţii

Prin **sortare** se înţelege ordonarea unei mulţimi de obiecte de acelaşi tip, pe baza unei componente de tip numeric, ce caracterizează fiecare obiect.

De regulă, obiectele sunt organizate sub formă de fişier sau tablou, în ambele cazuri ele formează un şir. Lungimea şirului este dată de numărul de obiecte (n).

Teoretic, obiectele sunt considerate înregistrări (în C sunt structuri):

$$R_0, R_1, \ldots, R_{n-1};$$

Fiecare înregistrare R_k, are asociată o *"cheie"* C_k, în funcție de care se face sortarea. Pe mulțimea cheilor se defineşte o relație de ordine, cu proprietățile:

- oricare ar fi două chei C_i, C_j are loc numai una din relațiile:

$$C_i < C_j, \quad C_i = C_j, \quad C_i > C_j \,;$$

- relația de ordine este tranzitivă.

Un şir este **sortat** dacă oricare ar fi $i, j \in \{0, 1, \ldots, n-1\}$, $i < j$, avem $C_i \leq C_j$. Această definiție implică o sortare în sens crescător pe mulțimea cheilor. În cele ce urmează, algoritmii de sortare vor fi implementați pentru sortare crescătoare, fapt ce nu micşorează gradul de generalitate; dacă dorim ca şirul final să fie sortat descrescător, el va fi citit şi înregistrat de la componenta n-1 către 0.

Un algoritm de sortare se numeşte **stabil**, dacă înregistrările cu chei egale rămân, în şirul sortat în aceeaşi ordine relativă în care se găseau în şirul inițial (nu se schimbă ordinea înregistrărilor egale).

Metodele de sortare se pot clasifica în două mari categorii:

- **metode de sortare internă**, în care înregistrările se află în memoria internă şi sunt rapid accesibile;
- **metode de sortare externă**, în care înregistrările se află pe suport extern (disc) iar timpul de acces (mare) face inacceptabil accesul repetat la înregistrări.

Dacă înregistrările ocupă spațiu mare în memorie (multe locații de memorie / înregistr.), se preferă construirea unui tablou de adrese ale înregistrărilor. În loc de muta înregistrări, se mută adrese în tabel, cu o importantă economie de timp. Dacă privim adresele înregistrărilor ca indici, tabloul de adrese conține în final permutarea indicilor, care permite accesul la înregistrări în ordinea crescătoare a cheilor.

Această metodă se numeşte **sortare prin tablou de adrese**.

În cazul sortării interne, dacă înregistrările nu ocupă volum mare de memorie, ele vor fi mutate efectiv iar metoda se numeşte **sortare de tablouri**.

8.2. Funcții polimorfice

Aceste funcții sunt construite pentru a se putea aplica oricăror tipuri de date. Evident că prelucrările concrete vor fi diferite de la un tip de date la altul dar aceste aspecte concrete se pot transfera funcțiilor specifice de prelucrare. Transmitând funcției polimorfice un pointer la funcția de prelucrare şi utilizând peste tot tipul de pointer universal void *, se poate realiza uşor o funcție polimorfică.

De exemplu, funcția standard, de bibliotecă qsort() implementează algoritmul de sortare internă Quicksort, având prototipul:

```
void qsort(void *tab, size_t n, size_t dim,
          int (*cmp)(const void *, const void *));
```

Se sortează tabloul tab, cu dimensiunea n, fiecare element din tablou având dimensiunea dim iar cmp este un pointer la o funcție care primeşte doi pointeri la două elemente ale tabloului, returnând valoare<0, 0 sau valoare>0 după cum este mai mare al doilea, sunt egale sau este mai mare primul element.

Exemple:

1. Pentru sortarea unui tablou de întregi, funcția de comparație trebuie să fie:

```
int cmp_int(const void *a, const void *b)
{
    return *((int*)a - *((int*)b;
}
```

Se convertesc pointerii (void *) la (int*), se ia apoi conținutul fiecăruia (deci doi întregi), se face diferența lor, care se returnează ca rezultat. Apelul va fi în acest caz:

```
int tab_int[100];
qsort(tab_int, 100, sizeof(int), cmp_int);
```

2. Pentru sortarea unui tablou de 100 de şiruri de caractere, fiecare cu lungimea maximă de 50 de octeți, declarat prin:

```
char a[100][50];
```

în condițiile în care se doreşte să se modifica efectiv poziția din memorie a celor 100 de şiruri, funcția de comparație va fi:

```
int cmp_sir(const void *a, const void *b)
{
    return strcmp(a, b);
}
```

Se utilizează funcția standard de comparație a șirurilor **strcmp()**; apelul funcției de sortare va fi:

```
qsort(a, 100, 50, cmp_sir);
```

ceea ce pune în evidență că un element al tabloului este de 50 de octeți.

3. Definim o structură şi un tablou de structuri:

```
typedef struct {int key; char *sir} STRUCT_1;
STRUCT_1 tab[100];
```

Funcția de comparație pentru căutarea unui element în tablou, după cheia numerică **key,** se defineşte acum astfel:

```
int cmp_str_1(const void *key, const void *elem)
{
    return *((int *) key) - ((STRUCT_1 *) elem) ->
key;
}
```

Dacă valoarea returnată este zero, elementul căutat a fost găsit. Aici key este adresa unui întreg (cheia numerică) iar elem este adresa unui element al tabloului, deci adresa unei structuri de tipul STRUCT_1, deci este vorba de tipuri diferite. Se converteşte pointerul key la (int *) şi apoi se ia conținutul, apoi se converteşte pointerul elem la (STRUCT_1 *) şi se ia câmpul key al structurii indicate de elem. Se întoarece diferența celor doi întregi.

Dacă dorim acum să sortăm tabloul de structuri de mai sus, cu funcția polimorfică qsort(), după cheia key, trebuie modificată funcția de comparație, deoarece se primesc de data asta adresele a două elemente de acelaşi tip (elementele tabloului).

```
int cmp_str_2(const void *a, const void *b)
{
return ((STRUCT_1 *)a)->key - ((STRUCT_1 *)b) - >key;
}
```

Pentru generalitate, toți algoritmii de sortare vor fi implementați prin **funcții polimorfice,** încât să se poată sorta orice tip de tablou, fără modificări în program. Pentru aceasta, vom considera o funcție externă care compară două înregistrări, pe baza adreselor lor din memorie (de tip void * pentru generalitate) şi care întoarce o valoare întreagă negativă, zero sau o valoare pozitivă, în funcție de relația dintre cele două înregistrări (<, =, >).

Definim, pentru aceasta, tipul de date:

```
typedef int (*PFCMP) (const void *, const void *) ;
```

adică un pointer la funcția de comparație. Toți algoritmii vor fi implementați după prototipul:

```
void sort(void *v, size_t n, size_t size, PFCMP cmp);
```

în care **v** este adresa de început a tabloului de înregistrări, **n** este numărul de înregistrări, **size** este dimensiunea în octeți a unei înregistrări iar **cmp** este pointerul la funcția care compară două înregistrări.

După apelul funcției de sortare, tabloul **v** va conține înregistrările în ordinea crescătoare a cheilor.

Procedând în acest mod, partea specifică a problemei (structura înregistrării, poziția şi natura cheii, tipul relației de ordine) este preluată de funcția de comparație şi detaşată de algoritmul propriu-zis, ceea ce conferă algoritmului un mare grad de generalitate.

Pentru transferul înregistrărilor dintr-o zonă de memorie în alta, considerăm două funcții externe, `swap()` - care schimbă între ele pozițiile din memorie a două înregistrări şi `copy()` - care copiază o înregistrare dată la o adresă de salvare.

```
void swap(void *v, size_t size, int i, int j);
void copy(void *a, void *b, size_t size);
```

Prima funcție schimbă între ele înregistrările `v[i]` şi `v[j]` iar a doua copiază înregistrarea de la adresa **b** (sursă) la adresa **a** (destinație).

Dimensiunea fiecărei înregistrări este de `size` octeți. Calculul adresei se face prin relația `(BYTE *)v + i*size`, unde tipul `BYTE` este definit de:

```
typedef unsigned char BYTE;
```

Metodele de sortare internă se clasifică în trei categorii:

- sortare prin interschimbare;
- sortare prin inserție;
- sortare prin selecție.

Fiecare din aceste categorii cuprinde atât metode elementare, în general neeficiente pentru blocuri mari de date, cât şi metode evoluate, de mare eficiență. Metodele elementare au avantajul că sunt mai uşor de înțeles şi au implementare simplă. Ele se pot folosi cu succes pentru tablouri de dimensiuni relativ reduse (sute, mii de înregistrări).

8.3. Eficiența algoritmilor de sortare

Este naturală problema comparării algoritmilor. Ce este un algoritm eficient şi de ce unul este superior altuia ? Analiza eficienței unui algoritm se face, de regulă în trei situații: şir inițial sortat, şir aleator şi pentru şir inițial sortat în ordine inversă.

Pentru algoritmii care nu necesită spațiu suplimentar de memorie, comparația se face după numărul de operații efectuate, care în esență sunt: **comparații de chei** şi **transferuri de înregistrări.** Transferurile se pot detalia în interschimbări şi copieri (o interschimbare necesită 3 copieri).

Evident, numărul de operații depinde de dimensiunea **n** a tabloului care este supus sortării. Natura acestei dependențe este cea care diferențiază algoritmii de sortare. Se defineşte o mărime care depinde de **n**, numită ordinul algoritmului **O(n)**.

Dacă numărul de operații se exprimă printr-o funcție polinomială de grad k, se spune că ordinul este $O(n^k)$. De exemplu, dacă numărul de operații este n(n-1)/2, ordinul este $O(n^2)$.

Algoritmii care apar în domeniul prelucrării datelor sunt de ordin polinomial, logaritmic sau combinații ale acestora.

În algoritmii de sortare, apar ordinele n^2 şi n*log(n).

Un algoritm bun de sortare este O(n*log(n)); algoritmii elementari sunt de ordin $O(n^2)$.

8.4. Sortare prin interschimbare: Bubblesort

Cea mai simplă este interschimbarea directă sau metoda bulelor.

Pentru v[i] fixat, se compară v[j] cu v[j-1] pentru j=n-1, n-2, . . . , i. şi se face un schimb reciproc de locuri în tablou, atunci când nu se respectă condiția de sortare crescătoare v[j]>=v[j-1]. La sfârşitul primei etape, pe locul v[0] ajunge, evident cel mai mic element.

După a două trecere, pe locul v[1] va ajunge , cel mai mic element din cele rămase, s.a.m.d.

Se procedează analog până la ultimul element, care nu se mai compară cu nimic, fiind cel mai mare.

Numărul de interschimbări este n(n-1)/2, în cazul cel mai defavorabil, când şirul inițial este sortat invers (dar noi nu ştim !).

O creştere a eficienţei se poate realiza astfel: dacă la o trecere prin bucla interioară (de comparări), nu s-a făcut nici o inversare de elemente, înseamnă că şirul este sortat şi algoritmul trebuie oprit. Acest lucru se realizează prin introducerea unei variabile logice `sortat` care este iniţializată cu 1 şi pusă în 0 la orice interschimbare. Se reduce numărul de operaţii în cazul unui tablou iniţial sortat (la n-1) şi cresc performanţele în cazul unui şir aleator.

Denumirea metodei provine din faptul că elementele v[i], de valoare mare, se deplasează către poziţia lor finală (urcă în şir), pas cu pas, asemănător bulelor de gaz într-un lichid.

```
void bubble_sort(void *v, size_t size, PFCMP cmp)
{
     int i, j, sortat;
     BYTE *a=(BYTE *) v;
     sortat=0;
     for(i=1; i<n && !sortat; i++) {
          sortat=1;
          for(j=n-1; j>=i; j--)
               if(cmp(a+(j-1)*size, a+j*size)>0)
               {  swap(a, size, j-1, j);
                   sortat=0;
               }
     }
}
```

8.5. Sortarea prin partiţionare şi interschimbare: Quicksort

Se poate încadra într-o tehnică mai generală, "*Divide et impera*" (împarte şi stăpâneşte), deoarece se bazează pe divizarea şirului iniţial în şiruri din ce în ce mai scurte.

Algoritmul Quicksort, fundamemtat în 1962 de C. A. R. Hoare, este un exemplu de algoritm performant de sortare, fiind de ordinul *n*log(n)*.

Este un algoritm recursiv, uşor de implementat în C.

Funcţia care realizaează sortarea primeşte ca date de intrare o parte a tabloului ce trebuie sortat, prin adresa de început şi doi indici *left* şi *right*. Iniţial funcţia se apelează cu indicii 0 şi n-1.

Se alege un element arbitrar al tabloului **v**, numit **pivot**, care se notează cu mark, uzual `mark = v[(left+right)/2]`. Se divide tabloul în două părţi în raport cu **mark**: toate elementele mai mici decât **mark** trec în stânga iar cele mai mari decât **mark** trec în dreapta.

În acest moment elementul **mark** se află pe poziția finală iar tabloul este partiționat în două tablouri de dimensiuni mai mici.

Dacă notăm cu k indicele pivotului, putem apela aceeaşi funcţie de sortare cu limitele `left şi k-1` pentru partea stângă şi cu limitele `k+1 şi right` pentru partea dreaptă.

Când se realizează condiția left>=right, algoritmul se încheie.

Există mai multe variante de implementare; în exemplul de mai jos, se utilizează doi indici, i şi j, care sunt inițializați cu left, respectiv cu right. Cât timp v[i]<mark, incrementăm i, apoi cât timp v[j]>mark, decrementăm j. Acum, dacă i<=j, se face interschimbarea v[i] cu v[j], actualizând similar indicii i şi j. Procesul continuă până când i>j.

S-a obținut următoarea partiționare:

- toate elementele `v[left],v[left+1],...,v[i-1]` sunt mai mici ca mark;

- toate elementele `v[j+1],v[j+2],...,v[right]` sunt mai mari ca mark;

- toate elementele `v[i],...,v[j]` sunt egale cu mark;

Acum se apelează aceeaşi funcţie cu indicii `left,j`, respectiv `i, right` (dacă left<j şi i<right).

Pentru eficiență, variabilele intens folosite în etapa de partiționare (i şi j) sunt declarate în clasa register.

```
void quick_sort(BYTE*v,size_t size,int left,
                 int right,PFCMPcmp)
{
     register int i, j;
     BYTE *mark;
     i=left; j=right;
     switch(j-i) {
          case 0: return;
          case 1: if(cmp(v+i*size, v+j*size)>0)
                   swap(v, size, i, j);
                  return;
          default: break;
     }
     mark=(BYTE*)malloc(size);
     copy(mark, v+((left+right)/2)*size, size);
     do {
          while(cmp(v+i*size, mark)<0)  i++;
          while(cmp(v+j*size, mark)>0)  j--;
          if(i<=j) swap(v, size, i++, j--);
           } while (i<=j);
     if(left<j)   quick_sort(v, size, left, j, cmp);
     if(i<right)  quick_sort(v, size, i, right, cmp);
     free(mark);
     }
```

```
    void Quick(void *v, size_t n, size_t size, PFCMP
cmp)
    {
        quick_sort(v, size, 0, (int) n-1, cmp);
    }
```

Analiza algoritmului Quicksort arată că, în cazul datelor aleatoare, el este de ordin O(nlog(n)).