Florian Moraru

PROGRAMAREA CALCULATOARELOR în limbajul C

2008

PROGRAMAREA CALCULATOARELOR
IN LIMBAJUL C

1. Introducere în programare. Limbajul C

2. Date si prelucrări

3. Operatii de intrare-iesire în C

4. Prelucrări conditionate

5. Prelucrări repetitive în C

1. Introducere în programare. Limbajul C.

Algoritmi si programe

Un algoritm este o metodã de rezolvare a unei probleme printr-o succesiune de operatii simple, cum ar fi operatii aritmetice, comparatii, s.a.. Numãrul de operatii este de obicei foarte mare, dar finit. Spre deosebire de aplicarea unor formule de calcul, un algoritm contine operatii executate conditionat, numai pentru anumite date, si operatii repetate de un numãr de ori, în functie de datele problemei. Un program este o descriere precisã si concisã a unui algoritm într-un limbaj de programare.

Algoritmii mai simpli pot fi exprimati direct într-un limbaj de programare, dar pentru un algoritm mai complex se practicã descrierea algoritmului fie sub formã graficã (organigrame sau scheme logice), fie folosind un "pseudocod", ca un text intermediar între limbajul natural si un limbaj de programare. Un pseudocod are reguli mai putine si descrie numai operatiile de prelucrare (nu si variabilele folosite). Nu existã un pseudocod standardizat sau unanim acceptat. Descrierea unor prelucrãri în pseudocod se poate face la diferite niveluri de detaliere.

Exemplu de pseudocod pentru algoritmul lui Euclid care determinã cel mai mare divizor comun a doi întregi, prin împãrtiri repetate:

```
repetã cat timp restul împãrtirii lui a prin b este nenul
    a primeste valoarea lui b
    b primeste valoarea restului împãrtirii a prin b
cmmdc este ultimul împãrtitor b care a produs rest 0
```

Tipic pentru un algoritm este faptul cã anumite operatii se executã conditionat (în functie de valorile datelor initiale), iar alte operatii se executã în mod repetat (iar numãrul de repetãri poate depinde de datele initiale). Altfel spus, anumite operatii dintr-un program pot sã nu fie executate de loc sau sã fie executate de un numãr de ori, functie de datele initiale.

La descrierea anterioarã pentru algoritmul lui Euclid trebuie sã adãugãm citirea datelor initiale (numerele întregi a si b) si afisarea rezultatului (cmmdc). Variantã de descriere a algoritmului lui Euclid în pseudocod:

```
citeste a si b
r = rest împãrtire a prin b
repeta cât timp r > 0
    a=b
    b=r
    r=rest impartire a prin b
scrie b
```

Prin deplasarea spre dreapta a celor trei linii s-a arãtat cã acele operatii fac obiectul repetãrii (nu însã si operatia de scriere).

Urmează un program C care implementează algoritmul lui Euclid descris anterior si care contine în plus declaratii pentru variabilele folosite:

```
int main () {
 int a=15, b=9, r ;        // declaratii cu initializarea variabilelor
 r= a%b;                   // restul impartirii lui a prin b
 while ( r > 0) {          // repeta cat timp rest nenul
   a=b;                         // noul deimpartit va fi a
   b=r;                      // noul impartitor va fi r
   r=a%b;                    // noul rest
 }
 printf ("%d \n",b);       // afiseaza ultimul impartitor
}
```

Exemplu de algoritm pentru afisarea numerelor perfecte mai mici ca un număr n dat, descris într-un pseudocod:

```
repetã pentru fiecare întreg m între 2 si n
   calcul sumã s a divizorilor lui m
   dacã m = s atunci
       scrie m
```

sau, la un nivel de detaliere mai aproape de un program în C:

```
repetã pentru fiecare întreg m între 2 si n
   s=0
   repeta pentru fiecare întreg d între 1 si m
      daca d este divizor al lui m atunci
         aduna d la s
   dacã m = s atunci
       scrie m
```

Prin alinierea spre dreapta s-a pus în evidentã structura de blocuri, adicã ce operatii fac obiectul unei comenzi de repetare (“repetã”) sau de selectie (“dacã”). Aceastã conventie nu este suficient de precisã si poate fi înlocuitã cu caractere delimitator pentru operatiile dintr-un bloc ce face obiectul unei repetări sau unei conditionări. Exemplu:

```
repetã pentru fiecare întreg m între 2 si n
{ s=0
  repeta pentru fiecare întreg d între 1 si m
    { daca d este divizor al lui m atunci
        aduna d la s
    }
  dacã m = s atunci
       scrie m
}
```

Un program are un caracter general si de aceea are nevoie de date initiale (diferite de la o utilizare la alta a programului), date care particularizează programul pentru o situatie concretã. De exemplu, un program pentru afisarea numerelor perfecte mai mici ca un numãr dat n are ca date initiale numãrul n si ca rezultate numerele perfecte între 2 si n. Exemplu:

```
#include <stdio.h>                  // necesara ptr functiile scanf si printf
void main () {
  int n,m,s,d ;                     // declaratii de variabile
  printf("n="); scanf("%d",&n);     // citire date
  for (m=2; m<=n ;m++) {                 // repeta ptr fiecare m
    s=0;                                 // suma divizorilor lui m
    for (d=1; d<m ; d++) {          // repeta ptr fiecare posibil divizor d
      if ( m % d==0)                // daca restul împãrtirii m/d este zero
        s=s+d;                      // aduna un divizor la suma
    }
    if (m==s)                       // daca m este numar perfect
      printf ("\n %d", m);          // afisare m singur pe o linie
  }
}
```

Rezultatele produse de un program pe baza datelor initiale sunt de obicei afisate pe ecran. Datele se introduc manual de la tastaturã sau se citesc din fisiere disc.

Operatiile uzuale din limbajele de programare sunt operatii de prelucrare (calcule, comparatii etc) si operatii de intrare-iesire (de citire-scriere). Aceste operatii sunt exprimate prin instructiuni ale limbajului sau prin apelarea unor functii standard predefinite (de bibliotecã).

Desfãsurarea în timp a instructiunilor de prelucrare si de intrare-iesire este controlatã prin instructiuni de repetere (de ciclare) si de selectie (de comparatie).

Fiecare limbaj de programare are reguli gramaticale precise, a cãror respectare este verificatã de programul compilator (translator).

Dezvoltarea de programe

Scrierea unui program într-un limbaj de programare este doar primul pas dintr-un proces care mai cuprinde si alti pasi. Mai corect ar fi sã spunem scrierea unei versiuni initiale a programului, pentru cã întotdeauna aceastã formã initialã este corectatã, modificatã sau extinsã pentru eliminarea unor erori, pentru satisfacerea unor noi cerinte sau pentru îmbunãtãtirea performantelor în executie.

Un program scris într-un limbaj independent de masinã (C, Pascal, s.a.) trebuie mai întâi tradus de cãtre un program translator sau compilator. Compilatorul citeste si analizează un text sursã (de exemplu în limbajul C) si produce un modul obiect (scris într-un fisier), dacã nu s-au gãsit erori în textul sursã. Pentru programele mari este uzual ca textul sursã sã fie format din mai multe fisiere sursã, care sã poatã fi scrise, compilate, verificate si modificate separat de celelalte fisiere sursã.

Mai multe module obiect, rezultate din compilări separate sunt legate împreunã si cu alte module extrase din biblioteci de functii standard într-un program executabil de cãtre un program numit editor de legãturi (“Linker” sau “Builder”). Executia unui program poate pune în evidentã erori de logicã sau chiar erori de programare care au trecut de compilare (mai ales în limbajul C).

Cauzele erorilor la executie sau unor rezultate gresite nu sunt de obicei evidente din cauzã cã ele sunt efectul unui numãr mare de operatii efectuate de calculator. Pentru descoperirea cauzelor erorilor se poate folosi un program depanator (“Debugger”) sau se pot insera intructiuni de afisare a unor rezultate intermediare în programul sursã, pentru trasarea evolutiei programului.

Fazele de modificare (editare) a textului sursã, de compilare, linkeditare si executie sunt repetate de câte ori este necesar pentru a obtine un program corect. De fapt, testarea unui program cu diverse date initiale poate arãta prezenta unor erori si nu absenta erorilor, iar efectuarea tuturor testelor necesare nu este posibilã pentru programe mai complexe (pentru un compilator sau un editor de texte, de exemplu).

Programele compilator si linkeditor pot fi apelate în mod linie de comandã sau prin selectarea unor optiuni din cadrul unui mediu integrat de dezvoltare a programelor (IDE = Integrated Development Environment).

Alte programe utilizate în procesul de dezvoltare a unor aplicatii mari sunt:

- program bibliotecar pentru crearea si modificarea unor biblioteci de subprograme pe baza unor module obiect rezultate din compilare.
- program pentru executia unor fisiere de comenzi necesare pentru compilarea selectivã si re-crearea programului executabil, dupã modificarea unor fisiere sursã sau obiect (“make”).
- program de control al versiunilor succesive de fisiere sursã.

Limbajul de programare C

Limbajul C s-a impus în principal datoritã existentei unui standard care contine toate facilitãtile necesare unui limbaj pentru a putea fi folosit într-o mare diversitate de aplicatii, fãrã a fi necesare abateri sau extinderi fatã de standard (ca în cazul limbajelor Basic si Pascal). Un exemplu este recunoasterea posibilitãtii ca un program sã fie format din mai multe fisiere sursã si a compilãrii lor separate, inclusiv referiri dintr-un fisier în altul. In plus, existã un numãr relativ mare de functii uzuale care fac parte din standardul limbajului si care contribuie la portabilitatea programelor C pe diferite platforme (sisteme de operare).

Unii programatori apreciazã faptul cã limbajul C permite un control total asupra operatiilor realizate de procesor si asupra functiilor sistemului de operare gazdã, aproape la fel ca si limbajele de asamblare. Astfel se explicã de ce majoritatea programelor de sistem si utilitare sunt scrise de mai multi ani în limbajul C, pe lângã multe programe de aplicatii.

Comparativ cu limbajele mai noi (Java, C#, s.a) limbajul C permite generarea unui cod masinã foarte eficient, aspect important pentru programele executate frecvent (compilatoare, editoare de texte, diverse utilitare, s.a.).

Limbajul C permite scrierea unor programe foarte compacte, ceea ce poate fi un avantaj dar si un dezavantaj, atunci când programele devin criptice si greu de înteles. Scurtarea programelor C s-a obtinut prin reducerea numãrului de cuvinte cheie, prin existenta unui numãr mare de operatori exprimati prin unul sau prin douã caractere speciale dar si prin posibilitatea de a combina mai multi operatori si expresii într-o singurã instructiune (acolo unde alte limbaje folosesc mai multe instructiuni pentru a obtine acelasi efect).

Din perspectiva timpului se poate spune cã instructiunile C sunt o reusitã a limbajului (si au fost preluate fãrã modificari de multe alte limbaje : C++, Java s.a.) dar functiile de intrare-iesire (printf, scanf) nu au fost un succes (si au fost înlocuite în limbaje ulterioare). Un alt neajuns s-a dovedit a fi necesitatea argumentelor de tip pointer pentru functiile care trebuie sã modifice o parte din argumentele primite si a fost corectat prin argumente de tip referintã.

Utilizarea directã de pointeri (adrese de memorie) de cãtre programatorii C corespunde lucrului cu adrese de memorie din limbajele de asamblare si permite operatii imposibile în alte limbaje, dar în timp s-a dovedit a fi o sursã importantã de erori la executie, greu de depistat.

Au mai fost preluate în limbajele post-C si anumite conventii, cum ar fi diferenta dintre litere mici si litere mari, diferenta dintre caractere individuale si siruri de caractere, operatorii, comentariile s.a.

Programarea în C este mai putin sigurã ca în alte limbaje (Pascal, Java) si necesitã mai multã atentie. Limbajul C permite o mare diversitate de constructii corecte sintactic (care trec de compilare), dar multe din ele trãdeazã intentiile programatorului si produc erori greu de gãsit la executie. Poate cel mai bun exemplu este utilizarea gresitã a operatorului de atribuire ‘=‘ în locul operatorului de comparare la egalitate ‘==‘. Exemplu:

```
if ( a=b) printf (" a = b" \n");        // gresit
if ( a==b) printf (" a = b" \n");       // corect
```

Elementele componente ale unui program

Orice limbaj de programare trebuie sã continã:

- Instructiuni imperative, prin care se comandã executarea anumitor actiuni (prelucrãri);
- Declaratii de variabile, de functii s.a., necesare compilatorului dar fãrã efect la executie;
- Comentarii, ignorate de compilator, destinate oamenilor care citesc programe.

In plus, limbajul C mai contine si directive preprocesor, pentru compilator.

Instructiunile executabile sunt grupate în functii (subprograme). In C trebuie sã existe cel putin o functie cu numele "main", cu care începe executia unui program. Celelalte functii sunt apelate din functia "main" sau din alte functii activate direct sau indirect de "main".

Prin "program" întelegem uneori toate instructiunile necesare rezolvării unei probleme, deci o aplicatie completă, dar uneori se întelege prin "program" doar programul principal (functia "main"). Functia “main” poate fi de tip *int* sau *void* si poate avea sau nu argumente.
Exemplu de program C minimal, cu o functie "main" ce contine o singură instructiune (apelul functiei "printf") si nu contine declaratii:

```
#include <stdio.h>
int main ( ) {
   printf (" main ");
}
```

Cuvântul *int* reprezintă tipul functiei "main" si arată că această functie nu transmite nici un rezultat prin numele său. Parantezele care urmează cuvântului "main" arată că numele "main" este numele unei functii (si nu este numele unei variabile), dar o functie fără parametri. Sunt posibile si alte forme de definire a functiei "main".

Acoladele sunt necesare pentru a delimita definitia unei functii, care este un bloc de instructiuni si declaratii.

In acest text functia “main” va fi scrisă conform cerintelor compilatorului “gcc”, considerat ca o referintă pentru compilatoarele de C si utilizabil atât în sisteme de tip Linux (Unix) cât si în sisteme MS-Windows (direct, în linie de comandă sau prin intermediul unui mediu integrat IDE).

Un program descrie procedurile de obtinere a unor rezultate pe baza unor date initiale si foloseste rezultate intermediare. Toate acestea sunt memorate în variabile ale programului. Pot exista si date constante, ale căror valori nu se pot modifica în cursul executiei. Variabilele folosite într-un program trebuie definite sau declarate prin declaratii ale limbajului. Exemplu:

```
#include <stdio.h>
/* calculeaza si afiseaza media a doua numere */
int main ( ) {
  int a,b; float c;              /* declaratii de variabile */
  scanf ("%d%d", &a,&b);         /* citire date initiale */
  c= (a+b) / 2.0;                /* instructiune de calcul */
  printf ("%f\n", c);            /* afisare rezultat */
}
```

In programul anterior "scanf" si "printf" sunt functii de citire de la tastatură si respectiv de afisare pe ecran, iar liniile în care ele apar sunt instructiuni pentru apelarea acestor functii. Practic nu există program fără operatii de citire a unor date si de scriere a unor rezultate. Datele initiale asigură adaptarea unui program general la o problemă concretă iar rezultatele obtinute de program trebuie comunicate persoanei care are nevoie de ele.

Directiva #include este necesară pentru că se folosesc functiile “scanf” si “printf”

Un program este adresat unui calculator pentru a i se cere efectuarea unor operatii, dar programul trebuie citit si înteles si de cãtre oameni; de aceea se folosesc comentarii care explicã de ce se fac anumite operatii (comentariile din exemplul anterior nu costituie un bun exemplu de comentarii).

Initial în limbajul C a fost un singur tip de comentariu, care începea cu secventa "/*' si se termina cu secventa "*/". Ulterior s-au adoptat si comentariile din C++, care încep cu secventa "//" si se terminã la sfârsitul liniei care contine acest comentariu, fiind mai comode pentru programatori.

Conventii lexicale ale limbajului C

Instructiunile si declaratiile limbajului C sunt formate din cuvinte cheie ale limbajului, din nume simbolice alese de programator, din constante (numerice si nenumerice) si din operatori formati în general din unul sau douã caractere speciale.

Vocabularul limbajului contine litere mari si mici ale alfabetului englez, cifre zecimale si o serie de caractere speciale, care nu sunt nici litere, nici cifre. Printre caracterele speciale mult folosite sunt semne de punctuatie (',' ';'), operatori ('=','+','-','*','/'), paranteze ('(',')',[',']','{'}') s.a.

In C se face diferentã între litere mici si litere mari, iar cuvintele cheie ale limbajului trebuie scrise cu litere mici. Cuvintele cheie se folosesc în declaratii si instructiuni si nu pot fi folosite ca nume de variabile sau de functii (sunt cuvinte rezervate ale limbajului). Exemple de cuvinte cheie:

```
int, float, char, void, unsigned,
do, while, for, if, switch
struct, typedef, const, sizeof
```

Numele de functii standard (scanf, printf, sqrt, etc.) nu sunt cuvinte cheie, dar nu se recomandã utilizarea lor în alte scopuri, ceea ce ar produce schimbarea sensului initial, atribuit în toate versiunile limbajului.

Literele mari se folosesc în numele unor constante simbolice predefinite :

```
EOF, M_PI, INT_MAX, INT_MIN
```

Constantele simbolice sunt definite în fisiere de tip “h” : EOF în “stdio.h”, M_PI în “math.h”, INT_MAX si INT_MIN în “limits.h”.

Prin numele de "spatii albe" se înteleg în C mai multe caractere folosite cu rol de separator: blanc (‘ ‘), tab ('\t'), linie nouã ('\n'),

Acolo unde este permis un spatiu alb pot fi folosite oricâte spatii albe (de obicei blancuri). Spatii albe sunt necesare între nume simbolice succesive (în declaratii, între cuvinte cheie si/sau identificatori) dar pot fi folosite si între alti atomi lexicali succesivi. Exemple:

```
const int * p;  typedef unsigned char byte;
```

Atomii lexicali ("tokens" în englezã) sunt: cuvinte cheie, identificatori (nume simbolice alese de programatori), numere (constante numerice), constante sir (între ghilimele), operatori si separatori. Un atom lexical trebuie scris integral pe o linie si nu se poate extinde pe mai multe linii. In cadrul unui atom lexical nu se pot folosi spatii albe (exceptie fac spatiilor dintr-o constantã sir).

Respectarea acestei reguli poate fi mai dificilã în cazul unor siruri constante lungi, dar existã posibilitatea prelungirii unui sir constant de pe o linie pe alta folosind caracterul '\'. Exemple:

```
puts (" Inceput sir foarte foarte lung\
   sfârsit sir");                 // spatiile albe se vor afisa
// solutie alternativa
puts ("Inceput sir foarte foarte lung",
       "sfârsit sir");            // spatiile albe nu conteazã
```

Spatiile albe se folosesc între elementele unei expresii, pentru a usura citirea lor, si la început de linie pentru alinierea instructiunilor dintr-un bloc.

Structura programelor C

Un program C este compus în general din mai multe functii, dintre care functia "main" nu poate lipsi, deoarece cu ea începe executia programului. Functiile pot face parte dintr-un singur fisier sursã sau din mai multe fisiere sursã. Un fisier sursã C este un fisier text care contine o succesiune de declaratii: definitii de functii si, eventual, declaratii de variabile.

Functia “main” poate fi declaratã fãrã argumente sau cu argumente, prin care ea primeste date transmise de operator prin linia de comandã care lanseazã programul în executie. Dacã functia “main” are tipul explicit sau implicit *int*, atunci trebuie folositã instructiunea *return* pentru a preciza un cod de terminare (zero pentru terminare normalã, negativ pentru terminare cu eroare). Exemplu:

```
#include <stdio.h>
int main ( ) {
  printf (" main ");
  return 0;
}
```

Compilatorul gcc nu semnaleazã ca eroare absenta intructiunii return din functia “main” sau din alte functii cu rezultat (cu tip diferit de *void*), dar aceastã libertate este si o sursã importantã de erori pentru alte functii decât “main”.

Definitia unei functii C are un antet si un bloc de instructiuni încadrat de acolade. In interiorul unei functii existã de obicei si alte blocuri de instructiuni, încadrate de acolade, si care pot contine declaratii de variabile. Antetul contine tipul si numele functiei si o listã de argumente.

Exemplu de program cu două functii:

```
#include <stdio.h>
void clear () {            // sterge ecran prin defilare
   int i;                  // variabila locala functiei clear
   for (i=0;i<24;i++)
     putchar('\n');
}
void main ( ) {
   clear( );               // apel functie
}
```

Functia “clear” putea fi scrisă (definită) si după functia “main”, dar atunci era necesară declararea acestei functii înainte de “main”. Exemplu:

```
#include <stdio.h>
void clear();          // declaratie functie
void main ( ) {
  clear( );         // apel functie
}
void clear () {       // definitie functie
  int i;
  for (i=0;i<24;i++)
     putchar('\n');
}
```

Intr-un program cu mai multe functii putem avea două categorii de variabile:

- variabile definite în interiorul functiilor, numite si "locale".
- variabile definite în afara functiilor, numite si "externe" (globale).

Locul unde este definită o variabilă determină domeniul de valabilitate al variabilei respective: o variabilă definită într-un bloc poate fi folosită numai în blocul respectiv. Pot exista variabile cu acelasi nume în blocuri diferite; ele se memorează la adrese diferite si se referă la valori diferite.

In primele versiuni ale limbajului C era obligatoriu ca toate declaratiile de variabile dintr-o functie (sau dintr-un dintr-un bloc) să fie grupate la începutul functiei, înainte de prima instructiune executabilă. In C++ si în ultimele versiuni de C declaratiile pot apare oriunde, intercalate cu instructiuni. Exemplu (incorect sintactic în C, corect sintactic în C++):

```
#include <stdio.h>
void main () {             // calcul factorial
  int n;              // un întreg dat
  scanf ("%d", &n);       // citeste valoare n
  long nf=1;            // variabila rezultat
  for (int k=1;k<=n;k++)         // repeta de n ori
     nf=nf * k;          // o înmultire
  printf ("%ld\n", nf);         // afisare rezultat
}
```

Instructiunile nu pot fi scrise în C decât în cadrul definitiei unei functii.

Directive preprocesor

Un program C contine una sau mai multe linii initiale, care încep toate cu caracterul '#'. Acestea sunt directive pentru preprocesorul C si sunt interpretate înainte de a se analiza programul propriu-zis (compus din instructiuni si declaratii). Directivele fac parte din standardul limbajului C.

Cele mai folosite directive sunt "#include" si "#define".

Directiva #include cere includerea în compilare a unor fisiere sursã C, care sunt de obicei fisiere "antet" ("header"), ce reunesc declaratii de functii standard. Fisierele de tip ".h" nu sunt biblioteci de functii si nu contin definitii de functii, dar grupeazã declaratii de functii, constante si tipuri de date.

Pentru a permite compilatorului sã verifice utilizarea corectã a unei functii este necesar ca el sã afle declaratia functiei (sau definitia ei) înainte de prima utilizare. Pentru o functie de bibiotecã definitia functiei este deja compilatã si nu se mai transmite programului compilator, deci trebuie comunicate doar informatiile despre tipul functiei, numãrul si tipul argumentelor printr-o declaratie ("prototip" al functiei). Fisierele antet contin declaratii de functii.

Absenta declaratiei unei functii utilizate (datoritã absentei unei directive "include") este semnalatã ca avertisment în programele C si ca eroare ce nu permite executia în C++. Pentru anumite functii absenta declaratiei afecteazã rezultatul functiei (considerat implicit de tip *int*), dar pentru alte functii (de tip *void* sau *int*) rezultatul nu este afectat de absenta declaratiei.

Orice program trebuie sã citeascã anumite date initiale variabile si sã scrie (pe ecran sau la imprimantã) rezultatele obtinute. In C nu existã instructiuni de citire si de scriere, dar existã mai multe functii standard destinate acestor operatii. Declaratiile functiilor standard de I/E sunt reunite în fisierul antet "stdio.h" ("Standard Input-Output "), care trebuie inclus în compilare:

```
#include <stdio.h>          // sau #include <STDIO.H>
```

Numele fisierelor antet pot fi scrise cu litere mici sau cu litere mari în sistemele MS-DOS si MS-Windows, deoarece nu sunt nume proprii limbajului C, ci sunt nume specifice sistemului de operare gazdã care are alte conventii.

Parantezele unghiulare '<' si '>' sunt delimitatori ai sirului de caractere ce reprezintã numele fisierului si aratã cã acest nume trebuie cãutat într-un anumit director (grup de fisiere).

Fiecare directivã de compilare trebuie scrisã pe o linie separatã si nu trebuie terminatã cu caracterul ';' spre deosebire de instructiuni si declaratii.

De obicei toate directivele "include" si "define" se scriu la începutul unui fisier sursã, în afara functiilor, dar pot fi scrise si între functii dacã respectã regula generalã cã "definitia trebuie sã preceadã utilizarea".

In multe din exemplele care urmeazã vom considera implicite directivele de includere necesare pentru functiile folosite, fãrã a le mai scrie (dar ele sunt necesare pentru o compilare fãrã erori).

2. Date si prelucrări

Variabile si constante

Orice program prelucrează un număr de date initiale si produce o serie de rezultate. In plus, pot fi necesare date de lucru, pentru păstrarea unor valori folosite în prelucrare, care nu sunt nici date initiale nici rezultate finale.

Toate aceste date sunt memorate la anumite adrese, dar programatorul se referă la ele prin nume simbolice. Cu exceptia unor date constante, valorile asociate unor nume se modifică pe parcursul executiei programului. De aici denumirea de "variabile" pentru numele atribuite datelor memorate.

Numele unei variabile începe obligatoriu cu o literă si poate fi urmat de litere si cifre. Caracterul special '_' (subliniere) este considerat literă, fiind folosit în numele unor variabile sau constante predefinite (în fisiere de tip H).

Aplicatiile calculatoarelor sunt diverse, iar limbajele de programare reflectă această diversitate, prin existenta mai multor tipuri de date: tipuri numerice întregi si neîntregi, siruri de caractere de lungime variabilă s.a.

Pentru a preciza tipul unei variabile este necesară o definitie (o declaratie). Cuvintele "definitie" si "declaratie" se folosesc uneori cu acelasi sens, pentru variabile declarate în "main" sau în alte functii.

In limbajul C se face diferentă între notiunile de "definitie" si "declaratie", iar diferenta apare la variabile definite într-un fisier sursă si declarate (si folosite) într-un alt fisier sursă. O definitie de variabilă alocă memorie pentru acea variabilă (în functie de tipul ei), dar o declaratie anuntă doar tipul unei variabile definite în altă parte, pentru a permite compilatorului să verifice utilizarea corectă a variabilelor.

O declaratie trebuie să specifice numele variabilei (ales de programator), tipul variabilei si, eventual, alte atribute. In C o variabilă poate avea mai multe atribute, care au valori implicite atunci când nu sunt specificate explicit (cu exceptia tipului care trebuie declarat explicit).

O eroare frecventă este utilizarea unor variabile neinitializate, eroare care nu este detectată de compilatoarele C si care produce efecte la executie.

Tipuri de date în limbajul C

Principalele tipuri de date în C sunt:

- Tipuri numerice întregi si neîntregi, de diferite lungimi.
- Tipuri pointer (adrese de memorie)
- Tipuri structurate (derivate): vectori, structuri s.a.

Pentru functiile fără rezultat s-a introdus cuvântul *void*, cu sensul "fără tip".

Tipul unei variabile C poate fi un tip predefinit (recunoscut de compilator) si specificat printr-un cuvânt cheie (*int,char,float* etc) sau poate fi un nume de tip atribuit de programator (prin declaratii *typedef* sau *struct*). Exemple de declaratii de variabile:

```
int a,b;
```

```
float x,y,z; double d;        // tipuri standard
stiva s;                      // tip definit de utilizator
```

O definitie de variabilã poate fi însotitã de initializarea ei cu o valoare, valoare care poate fi ulterior modificatã.

```
int suma=0;       // declaratie cu initializare
```

Asemãnãtor cu tipul variabilelor se declarã si tipul functiilor. Exemple:

```
int cmmdc (int a, int b);          // functie cu 2 argumente int si rezultat int
double sqrt (double x);            // functie cu argument double si rezultat double
```

Orice declaratie si orice instructiune trebuie terminatã cu caracterul ';' dar un bloc nu trebuie terminat cu ';'. Exemplu de definire a unei functii simple:

```
double sqr (double x) { return x * x; }    // square = ridicare la patrat
```

Declaratiile de variabile si de functii pot include si alte atribute: ***static,const.***

Datoritã reprezentãrii interne complet diferite, limbajele de programare trateazã diferit numerele întregi de numerele reale, care pot avea si o parte fractionarã. Pentru a utiliza eficient memoria si a satisface necesitãtile unei multitudini de aplicatii existã în C mai multe tipuri de întregi si respectiv de reali, ce diferã prin memoria alocatã si deci prin numãrul de cifre ce pot fi memorate si prin domeniul de valori.

Implicit toate numerele întregi sunt numere cu semn (algebrice), dar prin folosirea cuvântului cheie ***unsigned*** la declararea lor se poate cere interpretarea ca numere fãrã semn.

Tipurile întregi sunt: ***char , short , int , long , long long***

Tipuri neîntregi: ***float , double , long double*** (numai cu semn).

Standardul C din 1999 prevede si tipul boolean _Bool (sau ***bool***) pe un octet.

Reprezentarea internã si numãrul de octeti necesari pentru fiecare tip nu sunt reglementate de standardul limbajului C, dar limitele fiecãrui tip pentru o anumitã implementare a limbajului pot fi aflate din fisierul antet "limits.h".

Toate variabilele numerice de un anumit tip se reprezintã pe acelasi numãr de octeti, iar acest numãr limiteazã domeniul de valori (pentru întregi si neîntregi) si precizia numerelor neîntregi. Numãrul de octeti ocupati de un tip de date sau de o variabilã se poate obtine cu operatorul ***sizeof***.

De obicei tipul ***int*** ocupã 4 octeti, iar valoarea maximã este de cca. 10 cifre zecimale pentru tipul int. Depãsirile la operatii cu întregi de orice lungime nu sunt semnalate desi rezultatele sunt incorecte în caz de depãsire. Exemplu:

```
short int a=15000, b=20000, c;
c=a+b;                    // depasire ! c > 32767
```

Reprezentarea numerelor reale în diferite versiuni ale limbajului C este mai uniformã deoarece urmeazã un standard IEEE de reprezentare în virgulã mobilã. Pentru tipul *float* domeniul de valori este între 10E-38 si 10E+38 iar precizia este de 6 cifre zecimale exacte. Pentru tipul *double* domeniul de valori este între 10E-308 si 10E+308 iar precizia este de 15 cifre zecimale.

Limitele de reprezentare pentru fiecare tip de date sunt specificate în fisierul antet "limits.h", care contine si nume simbolice pentru aceste limite. Exemplu:

```
#define INT_MAX  2147483647
#define INT_MIN (-INT_MAX - 1)
```

De observat cã, la afisarea valorilor unor variabile reale se pot cere mai multe cifre zecimale decât pot fi memorate, dar cifrele suplimentare nu sunt corecte. Se pot cere, prin formatul de afisare, si mai putine cifre zecimale decât sunt memorate în calculator.

Constante în limbajul C

Tipul constantelor C rezultã din forma lor de scriere, dupã cum urmeazã:

- Constantele întregi sunt siruri de cifre zecimale, eventual precedate de un semn ('-', +'). Exemple :

0 , 11 , -205 , 12345

- Constantele care contin, pe lângã cifre si semn, un punct zecimal si/sau litera 'E' (sau 'e') sunt de tipul *double*. Exemple:

7.0 , -2. , 0.5 , .25 , 3e10 , 0.12345678E-14

- Constantele care contin un exponent precedat de litera 'E' ('e') sau contin un punct zecimal dar sunt urmate de litera 'F' ('f') sunt de tipul *float*. Exemple:

1.0f, -2.F , 5e10f , 7.5 E-14F

- Constantele formate dintr-un caracter între apostrofuri sunt de tip *char*. Exemple:

'0', 'a' , 'A', '+', '-', '\n' , '\t', ' '

Constantele caracter se pot scrie si sub forma unei secvente ce începe cu '\', urmat de o literã ('\n' = new line , '\t' =tab , '\b' = backspace etc), sau de codul numeric al caracterului în octal sau în hexazecimal (\012 = \0x0a = 10 este codul pentru caracterul de trecere la linie nouã '\n').

- Constantele întregi în baza 16 trebuie precedate de precedate de sufixul "0x". Cifrele hexazecimale sunt 0..9,A,B,C,D,E,F sau 0..9,a,b,c,d,e,f. Exemple

0x0A, 0x7FFFF, 0x2c, 0xef

- Constantele formate din unul sau mai multe caractere între ghilimele sunt constante sir de caractere . Exemple:

"a" , "alfa" , "-1234" , "####"

Orice constantã poate primi un nume, devenind o constantã simbolicã.

Utilizarea de constante simbolice în programe are mai multe avantaje:

- Permit modificarea mai simplã si mai sigurã a unei constante care apare în mai multe locuri.
- Permite întelegerea mai usoarã a programelor, cu mai putine comentarii.

Exemplu de nume pentru constanta ce reprezintã dimensiunea maximã a unor vectori :

```
#define NMAX  1000   // dimensiune maxima
int main () {
   int n, x[NMAX], y[NMAX];
   printf ("n= "); scanf ("%d"" &n);
   assert ( n < NMAX);
   ... // citire elemente vectori
```

Declaratia ***enum*** permite definirea mai multor constante întregi cu valori succesive sau nu, simultan cu definirea unui nou tip de date. Exemple:

```
enum color {BLACK, BLUE, RED};     //BLACK=0, BLUE=1, RED=2
enum color {RED=5, WHITE=15, BLUE=1};
```

Operatori si expresii aritmetice în limbajul C

O expresie este formatã din operatori, operanzi si paranteze rotunde. Operanzii pot fi constante, variabile sau functii. Parantezele se folosesc pentru a delimita subexpresii, care se calculeazã înaintea altor subexpresii, deci pentru a impune ordinea de calcul. Exemple de expresii aritmetice:

5, x , k+1 ,a / b, a / (b * c), 2 * n-1 , 1./sqrt(x) , sqr(b)-4*a*c

Operatorii aritmetici '+','-','*', '/' se pot folosi cu operanzi numerici întregi sau reali. Operatorul '/' cu operanzi întregi are rezultat întreg (partea întreagã a câtului) si operatorul '%' are ca rezultat restul împãrtirii întregi a doi întregi. Semnul restului este acelasi cu semnul deîmpãrtitului; restul poate fi negativ.

In general, rezultatul unei (sub)expresii cu operanzi întregi este întreg.

Dacã cei doi operanzi diferã ca tip atunci tipul “inferior” este automat promovat la tipul “superior” înainte de efectuarea operatiei. Un tip T1 este superior unui tip T2 dacã toate valorile de tipul T2 pot fi reprezentate în tipul T1 fãrã trunchiere sau pierdere de precizie. Ierarhia tipurilor aritmetice din C este urmãtoarea:

char < short < int < long < long long < float < double < long double

Subexpresiile cu operanzi întregi dintr-o expresie care contine si reali au rezultat întreg, deoarece evaluarea subexpresiilor se face în etape. Exemple:

```
float x = 9.8, y = 1/2 * x;      // y=0.
y= x / 2;                // y=4.9;
```

In limbajul C existã operator de atribuire ‘=‘, iar rezultatul expresiei de atribuire este valoarea atribuitã (copiatã). In partea stângã a unei atribuiri se poate afla o variabilã sau o expresie de indirectare printr-un pointer; în partea dreaptã a operatorului de atribuire poate sta orice expresie. Exemple:

```
k=1;  i=j=k=0;  d = b*b-4*a*c;  x1=(-b +sqrt(d))/(2*a);
```

La atribuire, dacã tipul pãrtii stânga diferã de tipul pãrtii dreapta atunci se face automat conversia de tip (la tipul din stânga), chiar dacã ea necesitã trunchiere sau pierdere de precizie. Exemplu:

```
int a;  a= sqrt(3.);   // a=1
```

Exemplu de conversii dorite de programator:

```
float rad,grd,min;          // radiani, grade, minute
int g,m;                    // nr intreg de grade, minute
grd = 180*rad/M_PI;         // nr de grade (neintreg)
g=grd;                      // sau g= (int)grd;
min=60*(grd-(float)g);      // min=60*(grd-g)
m=min;                      // sau m= (int)min;
```

Conversiile automate pot fi o sursã de erori (la executie) si de aceea se preferã conversii explicite prin operatorul de conversie (“cast”= fortare tip), care are forma (tip) si se aplica unei expresii. Exemple:

```
float x; int a,b;
x= (float)a/b;        // câtul exact
float x; int k;
k= (int)(x+0.5);      // rotunjire x la intregul apropiat
```

Conversia prin operatorul (tip) se poate face între orice tipuri aritmetice sau între tipuri pointer. Pentru tipurile aritmetice se poate folosi si atribuirea pentru modificarea tipului (si valorii) unor variabile sau functii. Exemplu:

```
float x; int k;
x= x+0.5; k=x;        // rotunjire x
```

In limbajul C există mai multi operatori care reunesc un calcul sau altă prelucrare cu o atribuire:

+= -= *= /= %=

Urmãtoarele expresii sunt echivalente ('v' = o variabilã, 'e' = o expresie):

```
v += e        v = v + e
```

Operatorii unari de incrementare (++) si decrementare (--) au ca efect mărirea si respectiv micsorarea cu 1 a valorii operandului numeric:

++x adunã 1 la x înainte de se folosi valoarea variabilei x

x++ adunã 1 la x dupã ce se foloseste valoarea variabilei x

Operatorii ++ si -- se pot aplica oricărei expresii numerice (întregi sau reale) si variabilelor pointer. In general acesti operatori realizează o prescurtare a atribuirilor de forma x=x+1 sau x=x-1, dar pot exista si diferente între cele două forme de mărire sau diminuare a unei valori.

Expresiile urmãtoare au rezultat diferit:

```
a= ++b    a=b++
```

Urmãtoarele expresii au acelasi efect dacã x este o variabilã :

```
x=x+1  x += 1  ++x    x++
```

Utilizarea postincrementării (sau postdecrementării) trebuie făcută cu precautie în cazul apelării unor functii, deoarece "adunarea după folosirea variabilei" poate însemna că incrementarea nu are nici un efect real în program. Exemplu de functie recursivă care vrea să determine lungimea unui sir de caractere terminat cu zero:

```
int length (char * s) {              // s= adresa sir
  if (*s ==0) return 0;              // un sir vid are lungime zero
  else return 1+ length(s++);        // lungime subsir urmator plus 1
}
```

Mărirea variabilei pointer s după apelul functiei "length" nu are nici un efect, iar functia este apelată mereu (la infinit) cu aceeasi adresă s. Variantele corecte sunt cu preincrementare sau cu adunare:

```
int length (char * s) {              // s= adresa sir
```

```
    if (*s ==0) return 0;                // un sir vid are lungime zero
    else return 1+ length(s+1);      // lungime subsir urmator plus 1
}
```

De asemenea trebuie retinut cã operatorii ++ si – sunt operatori unari, iar operatorii unari se aplicã de la dreapta la stânga. Deci expresia *p++ diferã de expresia (*p)++ în care '*' este operator unar de indirectare printr-un pointer.

Prelucrãri la nivel de bit

O variabilã este un nume pentru o zonã de memorie, care contine un sir de cifre binare (biti). Operatorii aritmetici interpretează sirurile de biti ca numere binare cu semn. Anumite aplicatii dau alte interpretări sirurilor de biti si necesitã operatii la nivel de bit sau grupuri de biti care nu sunt multiplii de 8.

Operatorii la nivel de bit din C sunt aplicabili numai unor operanzi de tip întreg (char, int, long). Putem deosebi douã categorii de operatori pe biti:

- Operatori logici bit cu bit
- Operatori pentru deplasare cu un numãr de biti

Operatorul unar '~' face o inversare logicã bit cu bit a operandului si poate fi util în crearea unor configuratii binare cu multi biti egali cu 1, pe orice lungime. Exemplu:

```
~0x8000       // este 0x7FFF
```

Operatorul pentru produs logic bit cu bit '&' se foloseste pentru fortarea pe zero a unor biti selectati printr-o mascã si pentru extragerea unor grupuri de biti dintr-un sir de biti. Pentru a extrage cei 4 biti din dreapta (mai putini semnificativi) dintr-un octet memorat în variabila 'c' vom scrie:

```
c & 0x0F
```

unde constanta 0x0F (în baza 16) reprezintã un octet cu primii 4 biti zero si ultimii 4 biti egali cu 1.

Operatorul pentru sumã logicã bit cu bit '|' se foloseste pentru a forta selectiv pe 1 anumiti biti si pentru a reuni douã configuratii binare într-un singur sir de biti. Exemplu:

```
a | 0x8000   // pune semn minus la numarul din a
```

Operatorul pentru sumã modulo 2 ("sau exclusiv") '^' poate fi folosit pentru inversarea logicã sau pentru anularea unei configuratii binare.

Operatorii pentru deplasare stânga '<<' sau dreapta '>>' se folosesc pentru modificarea unor configuratii binare. Pentru numere fãrã semn au acelasi efect cu înmultirea si respectiv împãrtirea cu puteri ale lui 2. Exemplu:

```
a >>10          // echivalent cu a / 1024
```

Functia urmãtoare afiseazã prin 4 cifre hexa un sir de 16 biti primit ca parametru :

```
 void printHex ( unsigned short h) {
   unsigned short  i, ch;
   for (i=1;i<=4;i++) {
     ch= h & 0xF000;             // extrage primii 4 biti din stanga
     h = h << 4;                 // se aduce urmatorul grup de 4 biti
     printf ("%01x",ch>>12);     // scrie ca cifra hexa ch aliniat la dreapta
   }
 }
```

Functia următoare afiseazã un numãr întreg pozitiv în binar (în baza 2):

```
 void binar(int n) {
  int m = 14;          // mai general:   m =8*sizeof(int)-2;
  int mask;            // masca binara ptr extragere biti din n
  while (m>=0) {       // repeta de 15 ori
    mask=1 << m;       // masca este 1 deplasat la stanga cu m biti
//    printf ("%d", (n & mask) >> m );
    printf ("%d", ((1<<m)&n)>>m);
    m--;               // pozitie bit extras si afisat
  }
}
```

Acelasi efect se poate obtine folosind operatii aritmetice :

```
void binar(int n) {
  int m =8*sizeof(int) -2;     // nr biti de valoare intr-un “int”
  int div;            // divizor = 2^14, 2^13,... 2^1, 2^0
  while (m>=0) {      // repeta de 15 ori
    div=1 << m;       // div = pow (2,m);
    printf ("%d", n /div );  // catul este cifra binara urmatoare
    n=n%div; m--;     // restul folosit mai departe ca deimpartit
  }
}
```

Ordinea de evaluare a expresiilor

Limbajul C are un numãr mare de operatori care pot fi combinati în expresii complexe. Ordinea în care actioneazã acesti operatori într-o expresie este datã în urmãtorul tabel de prioritãti

Prioritate	Operator
1	Paranteze si acces la structuri: ( ) [ ] -> .
2	Operatori unari: ! ~ + - ++ -- & * sizeof (tip)
3	Inmultire, împãrtire, rest : * / %
4	Adunare si scãdere: + -
5	Deplasãri: << >>
6	Relatii: <= < > >=
7	Egalitate: == !=
8	Produs logic bit cu bit: &

9	Sau exclusiv bit cu bit: ^
10	Sumã logicã bit cu bit: |
11	Produs logic: &&
12	Sumã logicã: ||
13	Operator conditional: ? :
14	Atribuiri: = *= /= %= += -= &= ^= |= <<= >>=
15	Operator virgula: ,

Ignorarea prioritãtii operatorilor conduce la erori de calcul detectabile numai la executie, prin depanarea programului.

Douã recomandări utile sunt evitarea expresiilor complexe (prin folosirea de variabile pentru rezultatul unor subexpresii) si utilizarea de paranteze pentru specificarea ordinii de calcul (chiar si atunci când ele nu sunt necesare).

Operatorii de aceeasi prioritate se evaluează în general de la stânga la dreapta, cu exceptia unor operatori care actionează de la dreapta la stânga (atribuire, operatorii unari si cel conditional).

Operatorii unari actionează înaintea operatorilor binari. Intre operatorii binari sunt de retinut câteva observatii:

- Operatorul de atribuire simplã si operatorii de atribuire combinatã cu alte operatii au prioritate foarte micã (doar operatorul virgulã are prioritate mai micã); de aceea pot fi necesare paranteze la subexpresii de atribuire din componenta altor expresii. Exemple în care atribuirea trebuie efectuatã înainte de a compara valoarea atribuitã:

```
while ( (c =getchar()) != EOF) ...
if ( (d= b*b-4*a*c) < 0) ...
```

- Operatorii aritmetici au prioritate înaintea celorlalti operatori binari, iar operatorii de relatie au prioritate fatã de operatorii logici. Exemplu:

```
(a<<3) + (a<<1)   // a*10 = a*8 + a*2
```

Instructiuni expresie în C

O expresie urmatã de caracterul ';' devine o instructiune expresie. Cazurile uzuale de instructiuni expresie sunt :

- Apelul unei functii (de tip "void" sau de alt tip) printr-o instructiune :

```
printf("n="); scanf("%d",&n);
```

- Instructiune de atribuire:

```
a=1;  b=a;  r=sqrt(a);  c=r/(a+b);  i=j=k=1;
```

- Instructiune vidã (expresie nulã):

;

- Instructiuni fãrã echivalent în alte limbaje:
  ++a; a++; a<<2;

Prin instructiuni expresie se exprimã operatiile de prelucrare si de intrare-iesire necesare oricãrui program.

Exemplu de program compus numai din instructiuni expresie:

```
#include <stdio.h>
#include <math.h>
int main () {
  float a,b,c,ua,ub,uc;
  printf("Lungimi laturi:");
  scanf ("%f%f%f",&a,&b,&c);
  ua = acos ( (b*b+c*c-a*a)/(2*b*c) );  // unghi A
  ub = acos ( (a*a+c*c-b*b)/(2*a*c) );  // unghi B
  uc = acos ( (b*b+a*a-c*c)/(2*a*b) );  // unghi C
  printf ("%8.6f%8.6f \n",ua+ub+uc, M_PI); // verificare
}
```

O declaratie cu initializare seamãnã cu o instructiune de atribuire, dar între ele existã cel putin douã diferente:

- O declaratie poate apare în afara unei functii, dar o instructiune nu poate fi scrisã decât într-o functie.
- O declaratie nu poate apare într-o structurã *if , for, while, do*. Exemplu:

```
while (int r=a%b) ...  // eroare sintacticã
```

Anumite initializãri permise la declararea unor variabile vector sau structurã nu sunt permise si ca atribuiri. Exemplu:

```
int a[4] = {1,2,3,4};          // declaratie cu initializare corecta
int a[4];  a={1,2,3,4};        // atribuire incorectã (eroare la compilare)
```

Erori de reprezentare a numerelor

La executia unor aplicatii numerice pot apãrea o serie de erori datoritã reprezentãrii numerelor în calculatoare si particularitãtilor operatorilor aritmetici:

- Erori la împãrtire de întregi si la atribuire la un întreg. Exemple:

```
x = 1/2*(a+b);       // x=0,  corect:  x=(a+b)/2 ;
int x = sqrt(2);  // x=1
```

- Erori de depãsire a valorilor maxime absolute la operatii cu întregi, chiar si în valori intermediare (în subexpresii). Un exemplu este calculul numãrului de secunde fatã de

ora zero pe baza a trei întregi ce reprezintã ora, minutul si secunda. Acest numãr poate depãsi cel mai mare întreg reprezentabil pe 16 biti (*short* sau *int* în unele implementãri). Exemplu:

```
#include <stdio.h>
 int main () {
   int h1,m1,s1, h2,m2,s2, h,m,s;          // h=ore,m=minute,s=secunde
   long t1,t2,t; int r;
   printf("timp1="); scanf("%d%d%d",&h1,&m1,&s1);
   printf("timp2="); scanf("%d%d%d",&h2,&m2,&s2);
   t1= 3600L*h1 + 60*m1 + s1;         // poate depasi daca t1 int
   t2= 3600L*h2 + 60*m2 + s2;          // poate depasi daca t2 int
   t=t1-t2;
   h= t/3600;   r=t%3600;                 // h=ore
   m=r/60;   s=r%60;                      // m=minute, s=secunde
   printf ("%02d:%02d:%02d \n",h, m, s);
}
```

Nu existã nici o metodã generalã de a detecta depãsirile la operatii cu întregi pe un numãr mare de calcule, dar în cazuri simple putem sã verificãm rezultatul unei operatii unde suspectãm o depãsire. Exemplu:

```
int main (){
  int a,b,c;
  scanf ("%d%d",&a,&b);
  c=a*b;
  if ( c/a != b)
    printf ("depasire !\n");
  else
    printf ("%d \n",c);
}
```

O alternativã este prevenirea aceste depãsiri. Exemplu:

```
  if (INT_MAX /a < b)          // INT_MAX definit in <limits.h>
    printf ("depasire ! \n");
  else
    printf ("%d \n", a*b);
```

- Erori la adunarea sau scãderea a douã numere reale cu valori foarte diferite prin aducerea lor la acelasi exponent înainte de operatie. Se poate pierde din precizia numãrului mai mic sau chiar ca acesta sã fie asimilat cu zero.

- Erori de rotunjire a numerelor reale datoritã numãrului limitat de cifre pentru mantisã. Mãrimea acestor erori depinde de tipul numerelor (*float* sau *double* sau *long double*), de tipul, numãrul si ordinea operatiilor aritmetice.

Pierderea de precizie este mai mare la împãrtire si de aceea se recomandã ca aceste operatii sã se efectueze cât mai târziu într-o secventã de operatii. Deci expresia:

(a*b)/c este preferabilã expresiei (a/c)*b

atât pentru a, b si c întregi cât si pentru valori reale (*float*).

Erorile de rotunjire se pot cumula pe un numãr mare de operatii, astfel cã în anumite metode iterative cresterea numãrului de pasi (de iteratii) peste un anumit prag nu mai reduce erorile de calcul intrinseci metodei, deoarece erorile de reprezentare însumate au o influentã prea mare asupra rezultatelor. Un exemplu este calculul valorii unor functii ca sumã a unei serii de puteri cu multi termeni; ridicarea la putere si factorialul au o crestere rapidã pentru numere supraunitare iar numerele subunitare ridicate la putere pot produce valori nesemnificative.

In general, precizia rezultatelor numerice este determinatã de mai multi factori: precizia datelor initiale (numãr de zecimale), numãrul si felul operatiilor, erori intrinseci metodei de calcul (pentru metode de aproximatii succesive), tipul variabilelor folosite, ordinea operatiilor într-o expresie.

3. Operatii de intrare-iesire în C

Functii standard de intrare-iesire

Pentru operatii de citire a datelor initiale si de afisare a rezultatelor sunt definite functii standard, declarate în fisierul antet STDIO.H. Aici vom prezenta numai acele functii folosite pentru citire de la tastaturã si pentru afisare pe ecran, deci pentru lucru cu fisierele standard numite “stdin” si “stdout”. Trebuie observat cã programele C care folosesc functii standard de I/E cu consola pot fi folosite, fãrã modificãri, pentru preluarea de date din orice fisier si pentru trimiterea rezultatelor în orice fisier, prin operatia numitã redirectare (redirectionare) a fisierelor standard. Redirectarea se face prin adãugarea unor argumente în linia de comandã la apelarea programului, ca în exemplele urmãtoare, unde “filter” este numele unui program (fisier executabil) care aplicã un filtru oarecare pe un text pentru a produce un alt text:

```
filter <input
filter >output
filter <input  >output
```

Exemplu de program “filter”:

```
#include <stdio.h>      // pentru functiile gets,puts
int main () {
  char line[256];       // aici se citeste o linie
  while ( gets(line) != NULL)          // repeta citire linie
    if ( line[0]=='/' && line[1]=='/')  // daca linie comentariu
      puts (line);                     // atunci se scrie linia
}
```

Utilizarea comenzii “filter” fãrã argumente citeste si afiseazã la consolã; utilizarea unui argument de forma <input redirectează intrãrile cãtre fisierul “input”, iar un argument de forma >output redirectează iesirile cãtre fisierul “output”. Redirectarea se poate aplica numai programelor cre lucrează cu fisiere text doarece fisierele stdin si stdout sunt fisiere text.

Un fisier text este un fisier care contine numai caractere ASCII grupate în linii (de lungimi diferite), fiecare linie terminatã cu un terminator de linie format din unul sau douã caractere. In sisteme Windows se folosesc douã caractere ca terminator de linie: ‘\n’ si ‘\r’, adicã “newline”=trecere la linie nouã si “return”=trecere la început de linie. In sisteme Unix/Linux se foloseste numai caracterul ‘\n’ (newline) ca terminator de linie, caracter generat de tasta “Enter”.

Functiile standard de I/E din C pot fi grupate în câteva familii:

- Functii de citire-scriere caractere individuale: getchar, putchar;
- Functii de citire-scriere linii de text: gets, puts;
- Functii de citire-scriere cu format (cu conversie): scanf, printf.

In general se va evita citirea de caractere individuale, fie cu “getchar”, fie cu “scanf”, mai ales dupã alte apeluri ale functiei “scanf” (cel putin la început, pânã la întelegerea modului de lucru a functiei “scanf”). Toate functiile de citire mentionate folosesc o zonã tampon (“buffer”) în care se adunã caracterele tastate pânã la apãsarea tastei “Enter”, când continutul zonei buffer este transmis programului. In acest fel este posibilã corectarea unor caractere introduse gresit înainte de apãsarea tastei “Enter”.

Caracterul ‘\n’ este prezent în zona buffer numai la functiile “getchar” si “scanf”, dar functia “gets” înlocuieste acest caracter cu un octet zero, ca terminator de sir în memorie (rezultatul functiei “gets” este un sir terminat cu zero).

Functia “scanf” recunoaste în zona buffer unul sau mai multe “câmpuri” (“fields”), separate si terminate prin caractere spatiu alb (blanc, ‘\n’, ‘\r’,’\f’); drept consecintã preluarea de caractere din buffer se opreste la primul spatiu alb, care poate fi caracterul terminator de linie ‘\n’. Urmãtorul apel al functiei “getchar” sau “scanf” se uitã în zona buffer dacã mai sunt caractere, înainte de a astepta introducerea de la taste si va gãsi caracterul terminator de linie rãmas de la citirea anterioarã. Din acest motiv secventele urmãtoare nu asteaptã douã introduceri de la consolã si afiseazã dupã o singurã introducere (terminatã cu Enter):

```
c=getchar(); c=getchar(); putchar (c);
scanf(“%c”, &c); scanf (“%c”,&c); putchar (c);
scanf(“%c”,&c); getchar();putchar (c);
getchar(); scanf(“%c”,&c); putchar (c);
```

Pentru ca un program sã citeascã corect un singur caracter de la tastaturã avem mai multe solutii:

- apelarea functiei fflush(stdin) înainte de oricare citire; aceastã functie goleste zona buffer asociatã tastaturii si este singura posibilitate de acces la aceastã zonã tampon.
- o citire falsã care sã preia terminatorul de linie din buffer (cu getchar(), de exemplu).
- utilizarea functiei nestandard “getch” (declaratã în “conio.h”), functie care nu foloseste o zonã buffer la citire (si nu necesitã actionarea tastei Enter la introducere).
- citirea unui singur caracter ca un sir de lungime 1:

```
char c[2];          // memorie ptr un caracter si pentru terminator de sir
scanf (“%1s”,c);    // citeste sir de lungime 1
```

Functia “scanf” cu orice format diferit de “%c” ignorã toate spatiile albe din buffer înainte de orice citire din buffer, deci va ignora caracterul ‘\n’ rãmas în buffer de la o citire anterioarã.

Mediul integrat Dev-Cpp închide automat fereastra în care se afiseazã rezultatele unui program, iar pentru mentinerea rezultatelor pe ecran vom folosi fie o citire falsã (cu “getch”) care sã punã programul în asteptare, fie instructiunea:

```
system(“pause”);          // functie declarata in stdlib.h
```

Functii de citire-scriere cu format

Functiile “scanf” si “printf” permit citirea cu format (ales de programator) si scrierea cu format pentru orice tip de date. Pentru numere se face o conversie automatã între formatul extern (sir de caractere cifre zecimale) si formatul intern (binar virgulã fixã sau binar virgulã mobilã).

Primul argument al functiilor “scanf” si “printf” este un sir de caractere cu rol de descriere a formatului de citire-scriere si acest sir poate contine:

- specificatori de format, adicã secvente de caractere care încep cu %.
- alte caractere, afisate ca atare de “printf” sau tratate de “scanf”.

Celelate argumente sunt variabile (la “scanf”) sau expresii (la “printf”) în care se citesc valori (“scanf”) sau ale cãror valori se scriu (“printf”).

Exemple de utilizare “printf”:

```
printf ("\n");                  // trecere la o noua linie
printf ("\n Eroare \n");        // scrie un sir constant
printf ("%d \n",a);             // scrie un intreg si schimba linia
printf ("a=%d  b=%d \n", a, b);      // scrie doi intregi
printf (“ %2d grade %2d min %2d sec \n”, g,m,s);
```

Programatorul trebuie sã asigure concordanta dintre tipul datelor din sirul format si tipul variabilelor sau expresiilor care urmeazã argumentului format, deoarece functiile scanf si printf nu fac nici o verificare si nu semnaleazã neconcordante. Exemplele urmãtoare trec de compilare dar afiseazã incorect valorile variabilelor:

```
int i=3; float f=3.14;
printf (“%f \n”, i);        // scrie 0.00       (Dev-Cpp)
printf (“%i \n”, f);        // scrie 1610612736  (Dev-Cpp)
```

Argumentele functiei “scanf” sunt de tip pointer si contin adresele unde se memoreazã valorile citite. Pentru variabile numerice aceste adrese se obtin cu operatorul de adresare (‘&’) aplicat variabilei care primeste valoarea cititã. Exemple:

```
scanf("%d",&n);             // citeste un întreg în variabila n
scanf("%d%d", &a,&b);  // citeste doi întregi in a si b
scanf (“%f”, &rad);         // citeste un numar real in “rad”
scanf (“%c”, &c);           // citeste un caracter in “c”
```

De retinut diferenta de utilizare a functiilor “scanf” si “printf”. Exemplu:

```
scanf("%d%d", &a,&b);  // citeste numere in a si b
printf("%d %d", a,b);       // scrie valorile din a si b
```

La citirea de siruri trebuie folosit un vector de caractere, iar în functia “scanf” se foloseste numele vectorului (echivalent cu un pointer). Exemplu:

```
char s[100];
```

```
scanf("%s", s);        // este gresit:  scanf("%s, &s);
```

Numerele citite cu "scanf" pot fi introduse pe linii separate sau în aceeasi linie dar separate prin spatii albe sau caractere "Tab". Intre numere succesive pot fi oricâte caractere separator ('\n','\t',' '). Un numãr se terminã la primul caracter care nu poate apare într-un numãr .

Functiile "scanf" si "printf" folosesc notiunea de "câmp" ("field"): un câmp contine o valoare si este separat de alte câmpuri prin spatii albe, inclusiv terminator de linie ('\n") ca spatiu alb. Fiecare descriptor de format poate contine mãrimea câmpului, ca numãr întreg. Aceastã mãrime se foloseste mai ales la afisare, pentru afisare numere pe coloane, aliniate la dreapta. In lipsa acestei informatii mãrimea câmpului rezultã din valoarea afisatã. Exemple:

```
printf("%d %d",a,b);          // 2 campuri separate prin blanc
printf("%8d8%d",a,b);         // 2 câmpuri de cate 8 caractere
```

Desi sunt permise si alte caractere în sirul cu rol de format din "scanf" se recomandã pentru început sã nu se foloseascã între specificatorii de format decât blancuri (pentru a usura întelegerea formatului de citire). Chiar si spatiile trebuie folosite cu atentie în sirul format din "scanf". Exemplu de citire care poate crea probleme la executie din cauza blancului din format:

```
scanf("%d  ",&a);      // corect este  scanf ("%d",&a);
printf("%d \n",a);
```

Functia "scanf" nu poate afisa nimic, iar pentru a precede introducerea de date de un mesaj trebuie folositã secventa "printf, scanf". Exemplu:

```
printf ("n= "); scanf ("%d", &n);
```

Specificatori de format în "scanf" si "printf"

Descriptorii de format admisi în functiile "scanf" si "printf" sunt:

%c	caractere individuale (cod ASCII)
%s	sir de caractere ASCII
%p	pointeri la *void*
%d, %i	numere întregi cu semn în baza 10 (zecimale)
%u	numere întregi fãrã semn în baza 10
%x,%X	numere întregi fãrã semn în baza 16 (hexa)
%ld,%li	numere întregi lungi
%f	numere reale, cu parte întreagã si fractionarã
%e,%E	numere reale cu mantisã si exponent (al lui 10)
%g	numere reale în format %f sau %e, functie de valoare
%lf,%le,%lg	numere reale în precizie dublã (*double*)
%Lf,%Le,%Lg	numere reale de tip *long double*

Dacã nu se precizeazã mãrimea câmpului si numãrul de cifre de la partea fractionarã pentru numere, atunci functia “printf” alege automat aceste valori:

- dimensiunea câmpului rezultã din numãrul de caractere necesar pentru afisarea cifrelor, semnului si altor caractere cerute de format;
- numãrul de cifre de la partea fractionarã este 6 indiferent dacã numerele sunt de tip *float* sau *double* sau *long double*, dacã nu este precizat explicit.

Se poate preciza numai mãrimea câmpului sau numai numãrul de cifre la partea fractionarã. Exemple:

```
float a=1.; double b=0.0002; long double c=7.5; float d=-12.34;
printf ("%.0f  %20lf  %20.10Lf  %f  \n", a, b, c, d);
```

Se poate specifica dimensiunea câmpului în care se afiseazã o valoare, ceea ce este util la scrierea mai multor valori în coloane. Dacã valoarea de afisat necesitã mai putine caractere decât este mãrimea câmpului, atunci aceastã valoare este aliniatã la dreapta în câmpul respectiv. Exemplu:

```
int a=203, b= 5, c=16;
printf (“%10d \n %10d \n %10d \n”,a,b,c);
```

Secventa anterioarã va scrie trei linii, iar numerele afisate vor apare într-o coloanã cu cifrele de aceeasi pondere aliniate unele sub altele.

Formatul cu exponent (“%e”) este util pentru numere foarte mari, foarte mici sau despre ale cãror valori nu se stie nimic. Numãrul este scris cu o mantisã fractionarã (între 0 si 10) si un exponent al lui 10, dupã litera E (e).

La formatul “%g” “printf” alege între formatul “%f” sau “%e” în functie de ordinul de mãrime al numãrului afisat: pentru numere foarte mari sau foarte mici formatul cu exponent, iar pentru celelalte formatul cu parte întreagã si parte fractionarã.

Intre caracterul ‘%’ si literele care desemneazã tipul valorilor citite/scrise mai pot apare, în ordine :

a) un caracter ce exprimã anumite optiuni de scriere:
- (minus) aliniere la stânga în câmpul de lungime specificatã
+ (plus) se afiseazã si semnul ‘+’ pentru numere pozitive
0 numerele se completeazã la stânga cu zerouri pe lungimea w
formã alternativã de scriere pentru numere (detalii în “Help”)

b) un numãr întreg ‘w’ ce aratã lungimea câmpului pe care se scrie o valoare, sau caracterul ‘*’ dacã lungimea câmpului se dã într-o variabilã de tip *int* care precede variabila a cãrei valoare se scrie.

c) punct urmat de un întreg, care aratã precizia (numãr de cifre dupã punctul zecimal) cu care se scriu numerele neîntregi.

d) una din literele ‘h’, ‘l’ sau ‘L’ care modificã lungimea tipului numeric.

Exemplu de utilizare a optiunii ‘0’ pentru a scrie întotdeauna douã cifre, chiar si pentru numere de o singurã cifrã :

```
int ora=9, min=7, sec=30;
printf ("%02d:%02d:%02d\n",ora, min, sec);   // scrie  09:07:30
```

Exemplu de utilizare a optiunii ‘-’ pentru aliniere siruri la stânga:

```
char  a[ ] = "unu", b[ ] ="cinci", c[ ]= "sapte" ;
printf (" %-10s \n %-10s \n %-10s \n", a, b, c);
```

In general trebuie să existe o concordantă între numărul si tipul variabilelor si formatul de citire sau scriere din functiile “scanf” si “printf”, dar această concordantă nu poate fi verificată de compilator si nici nu este semnalată ca eroare la executie, dar se manifestă prin falsificarea valorilor citite sau scrise. O exceptie notabilă de la această regulă generală este posibilitatea de a citi sau scrie corect numere de tip ***double*** cu formatul “%f” (pentru tipul ***float***), dar nu si numere de tip ***long double*** (din cauza diferentelor de reprezentare internă a exponentului).

4. Prelucrări conditionate

Blocul de instructiuni

Instructiunile expresie dintr-un program sunt executate în ordinea aparitiei lor în program (secventa de scriere este si secventa de exeutie).

In limbajul C un bloc grupează mai multe instructiuni (si declaratii) între acolade. Exemple:

```
{ t=a; a=b; b=t;}                    // schimba a si b între ele
{ int t; t=a; a=b; b=t;}          // schimba a si b prin t
```

Uneori un bloc contine doar o singură instructiune. Un bloc nu trebuie terminat cu ';' dar nu este gresit dacă se foloseste ';' considerat ca instructiune vidă.

Acoladele nu modifică ordinea de executie, dar permit tratarea unui grup de instructiuni ca o singură instructiune de către alte instructiuni de control (*if, while, do, for* s.a). Instructiunile de control au ca obiect, prin definitie, o singură instructiune (care se repetă sau care este selectată pentru executie). Pentru a extinde domeniul de actiune al acestor instructiuni la un grup de operatii se folosesc acolade pentru gruparea instructiunilor vizate de comenzile *if, for, while, do, switch*. Exemplu:

```
scanf ("%d", &n);
if ( n > MAX) {
  printf ("Eroare in date: n > %d \n",MAX);
  return;
}
```

Instructiunea "if"

Instructiunea introdusă prin cuvântul cheie *if* exprimă o decizie binară si poate avea două forme: o formă fără cuvântul *else* si o formă cu *else* :

```
if (e)  i                  // fara alternativa "else"
if (e)  i1 else I2         // cu alternativa "else"
```

In descrierea unor structuri de control vom folosi următoarele notatii:

e, e1, e2,... expresii (sau conditii)

i, i1, i2 instructiuni sau blocuri

Instructiunile i, i1,i2 pot fi:

- O instructiune expresie, terminată cu ';' (terminatorul face parte din instructiune).
- O instructiune compusă, între acolade.
- O altă instructiune de control.

Expresia din *if* este de obicei o expresie de relatie sau o expresie logicã, dar poate fi orice expresie cu rezultat numeric. Exemplu:

```
if ( b*b - 4*a*c < 0)      // discriminant ecuatie de gradul 2
   puts( “radacini complex conjugate”);
```

Valoarea expresiei dintre paranteze se comparã cu zero, iar instructiunea care urmeazã se va executa numai atunci când expresia are o valoare nenulã. In general expresia din instructiunea *if* reprezintã o conditie, care poate fi adevaratã (valoare nenulã) sau falsã (valoare nulã). De obicei expresia este o expresie de relatie (o comparatie de valori numerice) sau o expresie logicã care combinã mai multe relatii într-o conditie compusã.

Exemplu de instructiune *if* fãrã alternativã *else*:

```
if ( sec >= 60)     // verifica numar de secunde
   err=1;
```

De multe ori se alege o secventã de operatii (instructiuni) si trebuie folosite acoladele pentru precizarea acestei secvente. Exemplu:

```
// inversarea valorilor lui a si b daca a>b
if ( a > b) {
   t=a; a=b; b=t;
}
```

De observat cã pentru comparatia la diferit de zero nu trebuie neapãrat folosit operatorul de inegalitate (!=), desi folosirea lui poate face codul sursã mai clar:

```
if (d) return;      //  if (d != 0) return;
```

Forma instructiunii *if* care foloseste cuvântul cheie *else* permite alegerea dintre douã secvente de operatii posibile, în functie de o conditie. Exemplu:

```
// determinare minim dintre a si b
if ( a < b)
        min=a;
else
        min=b;
```

Secventa anterioarã se poate scrie si astfel:

```
min=a;          // considera ca a este minim
if (b < min)    // daca b este mai mic ca min
   min=b;       // atunci minim va fi b
```

Instructiunile precedate de *if* si *else* sunt de obicei scrise pe liniile urmãtoare si sunt deplasate spre dreapta, pentru a pune în evidentã structurile si modul de asociere între *if* si *else*. Acest mod de scriere permite citirea corectã a unor cascade de decizii. Exemplu:

```
// determinare tip triunghi cu laturile a,b,c
 if ( a==b && b==c)
          printf ("echilateral \n");
 else   if ( a==b || b==c || a==c)
             printf ("isoscel \n");
 else
             printf ("oarecare \n");
```

O problemã de interpretare poate apare în cazul a douã (sau mai multe) instructiuni *if* incluse, dintre care unele au alternativa *else*, iar altele nu contin pe *else*. Regula de interpretare este aceea cã *else* este asociat cu cel mai apropiat *if* fãrã *else* (dinaintea lui). Exemplu:

```
if ( a == b )
   if (b == c)
          printf ("a==b==c \n");
  else
          printf (" a==b si b!=c \n");
```

Pentru a programa o instructiune *if* cu *else* care contine un *if* fãrã *else* avem mai multe posibilitãti:

```
if ( e1) {            if ( ! e1)
  if (e2)               i2
    i1            else if (e2)
}               i1
else
   i2
```

Exemplu dintr-un program care inverseazã pe a cu b daca a<b:

```
if ( a>0 && b>0) {
   if ( a<b ) {
      c=a; a=b; b=c;
   }
}
else {
   printf (“eroare in date \n”);
   return;
}
```

O solutie mai simplã si mai clarã este urmãtoarea:

```
if ( a <= 0 || b <= 0) {
   printf (“eroare in date \n”);
   return;
}
if ( a<b ) {
   c=a; a=b; b=c;
}
```

Din exemplele anterioare se vede cã modul de exprimare a conditiilor verificate si ordinea lor poate simplifica sau poate complica inutil un program, mai ales atunci când instructiunea *if* este într-un ciclu *while* sau *for*.

Operatori de relatie si logici

Operatorii de relatie se folosesc de obicei între operanzi numerici si, mai rar, între variabile pointer. In limbajul C operatorii de comparatie la egalitate si inegalitate aratã mai deosebit:

```
==  comparatie la egalitate (identitate)
!=  comparatie la inegalitate
```

Operatorii pentru alte relatii au forma din matematicã si din alte limbaje:

```
< , <= , >, >=
```

Toti operatorii de relatie au rezultat zero (0) dacã relatia nu este adevãratã si unu (1) dacã relatia este adevãratã.

Comparatia la egalitate de numere neîntregi este nesigurã si trebuie evitatã, din cauza erorilor de reprezentare internã a numerelor reale. Se va compara mai bine diferenta celor douã valori cu un epsilon foarte mic. Exemplu:

```
// daca punctul (x0,y0) se afla pe dreapta y=a*x+b
  if ( fabs (y0- (a*x0+b)) < 1e-5) ...      //  in loc de  if ( y0 ==a*x0+b) ...
```

Operatorii logici se folosesc de obicei între expresii de relatie pentru a exprima conditii compuse din douã sau mai multe relatii. Operatorii logici au rezultat 1 sau 0 dupã cum rezultatul expresiei logice este adevãrat sau fals. Operatorii logici binari în C sunt:

```
&&    si-logic       (  a && b =1 dacã si a==1 si b==1)
||      sau-logic     ( a || b =1 dacã sau a==1 sau b==1 sau a==b==1)
```

Operatorul && se foloseste pentru a verifica îndeplinirea simultanã a douã sau mai multe conditii, iar operatorul || se foloseste pentru a verifica dacã cel putin una dintre douã (sau mai multe) conditii este adevãratã.

Exemple de conditii compuse:

```
if ( x >= a && x <= b)            // daca x mai mare ca a si x mai mic ca b
        printf(" x in [a,b] \n");
if ( x < a || x > b)              // daca x mai mic ca a sau x mai mare ca b
        printf ("x in afara interv. [a,b] \n");
```

De observat cã efectuarea mai multor verificãri poate fi exprimatã uneori fie prin mai multe instructiuni *if*, fie printr-o singurã instructiune *if* cu expresie logicã. Exemplu:

```
if ( x >= a)
  if ( x <= b)
        printf(" x in [a,b] \n");
```

Diferenta apare atunci când existã alternative la fiecare conditie testatã.

```
if ( x >= a)
  if ( x <= b)
         printf(" x intre a si b \n");
      else
         printf(" x > b \n");
else
       printf(" x < a\n");
```

Operatorul unar de negare logicã este '!'. Exemplu:

```
if (!d) return;      // preferabil:     if ( d==0) return;
```

Negarea unei sume logice este un produs logic si reciproc. Exemple:

```
a >=0 && b >=0      // echiv. cu  !(a<0 || b<0)
x < a || x > b          // echiv. cu  !(x>=a && x<=b)
```

Intotdeauna putem alege între testarea unei conditii sau a negatiei sale, dar consecintele acestei alegeri pot fi diferite, ca numãr de instructiuni, mai ales atunci când instructiunea *if* se aflã într-un ciclu.

Expresia continutã în instructiunea *if* poate include si o atribuire. Exemplu:

```
    if ( d = min2 - min1) return d;// intr-o functie de comparare ore,min,sec
```

Instructiunea anterioarã poate fi derutantã la citire si chiar este semnalatã cu avertisment de multe compilatoare, care presupun cã s-a folosit eronat atribuirea în loc de comparatie la egalitate (o eroare frecventã).

Prioritatea operatorilor logici este mai micã decât a operatorilor de relatie si de aceea nu sunt necesare paranteze în jurul expresiilor de relatie combinate prin operatori logici. Exemplu:

```
  // verifica daca a,b,c pot fi laturile unui triunghi
    if (a < b+c && b < a+c && c < a+b)
      printf ("a,b,c pot forma un triunghi \n");

  // verifica daca a,b,c nu pot fi laturile unui triunghi
    if ( a > b+c || b > a+c || c > a+b )
      printf (" a,b,c nu pot forma un triunghi \n");
```

Intr-o expresie logicã evaluarea operanzilor (expresii de relatie) se face de la stânga la dreapta; din acest motiv ordinea operanzilor într-o expresie logicã poate fi uneori importantã si poate conduce la erori de programare. Exemplu:

```
int main () {
  int k, b=9, a[]={1,2,3,4};
  k=0;
  while ( b != a[k] && k<5 )
    k++;
  if (k<5)
    printf ("gasit in pozitia %d \n",k);
  else
    printf ("negasit \n");
}
```

In programul anterior indicele “k” poate ajunge egal cu 4 iar, în anumite implementãri (Borland C, de ex.) rezultatul afisat este “gasit în pozitia 4” deoarece valoarea lui “b” este memoratã imediat lângã a[3]. In astfel de cazuri trebuie verificat mai întâi dacã variabila “k” este în domeniul permis si apoi sã fie folositã în comparatie:

```
  while ( k < 5 && b != a[k] )
    k++;
```

Dacã primul operand dintr-o expresie logicã determinã rezultatul expresiei prin valoarea sa, nu se mai evalueazã si ceilalti operanzi (în expresii de relatie care pot include si calcule). Evaluarea unui produs logic se opreste la primul operand nul, deoarece este sigur cã rezultatul produsului va fi nul (fals), indiferent de valorile celorlalti operanzi. La fel, evaluarea unei sume logice se opreste la primul operand nenul, cu rezultat 1 (adevãrat).

In general se vor evita expresii complicate care includ si calcule si atribuiri si verificãri de conditii.

Expresii conditionale

Limbajul C contine o expresie ternarã (cu trei operanzi), care poate fi privitã ca o expresie concentratã a unei instructiuni *if*:

```
e1 ? e2 : e3
```

Instructiunea x =e1?e2:e3 este echivalentã ca efect cu instructiunea urmãtoare:

```
if (e1) x=e2;
    else x=e3;
```

Diferenta este cã expresia conditionalã nu necesitã o variabilã care sã primeascã rezultatul (exp2 sau exp3) si poate reduce lungimea unor secvente de program sau unor functii . Exemple:

```
  // functie pentru minim intre doua variabile
int minim (int a, int b) {
   return a<b ? a:b;
}
  // afisarea unui mesaj dintre 2 posibile
   printf ( prim ? "este prim \n" : "nu este prim \n");
```

Uneori se poate reduce numãrul de instructiuni *if* fãrã expresii conditionale, dar folosind alte observatii specifice problemei. Exemplu de secventã pentru adunarea a douã momente de timp exprimate prin orã, minut, secundã:

```
s=s1+s2;         // secunde
if (s >=60) {
  s=s-60; m1++;
}
m=m1+m2;          // minute
if (m >=60) {
  m=m-60; h1++;
}
h=h1+h2;        // ore
```

Solutia fãrã instructiuni *if* este datã mai jos:

```
x=s1+s2; s= x%60;           // secunde
x=m1+m2 + x/60; m=x%/60;       // minute
h=h1+h2+x/60;             // ore
```

Instructiunea "switch"

Selectia multiplã, dintre mai multe cazuri posibile, se poate face cu mai multe instructiuni *if* incluse unele în altele sau cu instructiunea *switch.* Instructiunea *switch* face o enumerare a cazurilor posibile (fiecare precedat de cuvântul cheie "case") între acolade si foloseste o expresie de selectie, cu rezultat întreg. Forma generalã este:

```
switch (e) {      // e= expresie de selectie
  case c1: s1;       // cazul c1
  case c2: s2;       // cazul c2
   . . .             // alte cazuri
  default: s;      // cazul implicit ( poate lipsi)
}
```

unde: c1,c2,.. sunt constante sau expresii constante întregi (inclusiv "char")
s, s1, s2 ... sunt secvente de instructiuni (cu sau fãrã acolade)

Deseori cazurile enumerate se exclud reciproc si fiecare secventã de instructiuni se terminã cu ***break***, pentru ca dupã selectia unui caz sã se sarã dupã blocul ***switch***. Exemplu:

```
swich ( c=getchar()) {       // c poate fi +,-,*,/
  case ' + ':  r=a + b; break;
  case ' - ':   r=a - b; break;
  case ' * ':   r=a * b; break;
  case ' / ':   r=a / b; break;
  default:   error();       // tratare erori
}
```

Secventã de instructiuni ***if*** echivalentã cu instructiunea anterioarã:

```
c=getchar();
if (c=='+')  r=a+b;
else if (c=='-') r=a-b;
else if (c=='*') r=a*b;
else if (c=='/') r=a/b;
else error();
```

Prin definitia instructiunii ***switch*** dupã executarea instructiunilor unui caz se trece la cazul imediat urmãtor (în lipsa unei instructiuni ***break***). Aceastã interpretare permite ca mai multe cazuri sã foloseascã în comun aceleasi operatii (partial sau în totalitate). Exemple:

```
// determinare semn numar din primul caracter citit
 switch (c=getchar()) {            // c este semn sau cifra
   case '-' :  semn=1; c=getchar(); break;
   case '+': c=getchar();      // si semn=0
   default:  semn=0;           // semn implicit
 }

 // determina nr de zile dintr-o lunã a unui an nebisect
switch (luna) {
 case 2: zile=28; break;           // februarie
```

```
   // aprilie, iunie,..., noiembrie
 case 4: case 6: case 9: case 11: zile =30; break;
   // ianuarie, martie, mai,.. decembrie
  default: zile=31; break;          //  celelalte (1,3,5,..)
 }
```

Cazul ***default*** poate lipsi, dar când este prezent atunci este selectat când valoarea expresiei de selectie diferã de toate cazurile enumerate explicit. Dacã lipseste cuvântul cheie ***default*** si nu este satisfãcut nici un caz, atunci se trece la instructiunea urmãtoare lui ***switch***.

Macroinstructiunea "assert"

Macroinstructiunea ***assert***, definitã în fisierul <assert.h>, este expandatã printr-o instructiune ***if*** si este folositã pentru verificarea unor conditii, fãrã a încãrca programele cu instructiuni ***if***, care le-ar face mai greu de citit.

assert verificã o asertiune, adicã o afirmatie presupusã a fi adevãratã, dar care se poate dovedi falsã.

Utilizarea este similarã cu apelul unei functii de tip ***void***, cu un argument al cãrei rezultat poate fi "adevãrat" sau "fals" (nenul sau nul). Parametrul efectiv este o expresie de relatie sau logicã care exprimã conditia verificatã. Dacã rezultatul expresiei din ***assert*** este nenul (adevãrat) atunci programul continuã normal, dar dacã expresia este nulã (falsã) atunci se afiseazã un mesaj care include expresia testatã, numele fisierului sursã si numãrul liniei din fisier, dupã care programul se opreste. Exemple de utilizare:

```
assert ( n <= MAX);          // verifica daca n <= MAX
assert ( a > 0 && b > 0);    // nici o actiune daca a>0 si b>0
```

Prin simplitatea de utilizare ***assert*** încurajeazã efectuarea cât mai multor verificãri asupra corectitudinii datelor initiale citite sau primite ca argumente de functii si asupra unor rezultate intermediare.

Macroinstructiunea ***assert*** se foloseste pentru erori irecuperabile si mai ales în etapa de punere la punct a programelor, deoarece pentru versiunea finalã se preferã afisarea unor mesaje mai explicite pentru utilizatorii programului, eventual în altã limbã decât engleza, însotite de semnale sonore sau de imagini (pentru programe cu interfatã graficã). Exemplu:

```
#include <assert.h>
#include <stdio.h>
#define MAX 1000
int main () {
  int n;
  printf ("n = "); scanf ("%d",&n);
  if ( n > MAX) {
     printf (" Eroare: n > %d \n",MAX);
```

```
    return ;
  }
  ...
}
```

De asemenea, *assert* se poate folosi pentru erori foarte putin probabile dar posibile totusi. Exemplu:

```
double arie (double a, double b, double c) {    // arie triunghi cu laturile a,b,c
   double p;
   assert (a > 0 && b > 0 && c > 0);       // verificare argumente functie
   p =(a+b+c)/2;
   return sqrt (p*(p-a)*(p-b)*(p-c));
}
```

Anumite erori la operatii de citire de la consolã sunt recuperabile, în sensul cã se poate cere operatorului repetarea introducerii, si nu se va folosi *assert*.

Eliminarea tuturor apelurilor *assert* dintr-un program se poate face prin secventa de directive:

```
 #define NDEBUG      // inainte de include <assert.h>
 #include <assert.h>
```

4. Prelucrãri repetitive în C

Instructiunea "while"

Instructiunea *while* exprimã structura de ciclu cu conditie initialã si cu numãr necunoscut de pasi si are forma urmãtoare:

while (e) i

unde ‘e’ este o expresie, iar ‘i’ este o instructiune (instructiune expresie, bloc sau instructiune de control).

Efectul este acela de executare repetatã a instructiunii continute în instructiunea *while* cât timp expresia din paranteze are o valoare nenulã (este adevaratã). Este posibil ca numãrul de repetãri sã fie zero dacã expresia are valoarea zero de la început. Exemplu:

```
    // cmmdc prin incercari succesive de posibili divizori
d= min (a,b);                  // divizor maxim posibil
while (a % d || b % d )      // repeta cat timp nici a nici b nu se divid prin d
  d=d -1;                  // incearca alt numar mai mic
```

In exemplul anterior, dacã a=8 si b=4 atunci rezultatul este d=4 si nu se executã niciodatã instructiunea din ciclu (d=d-1).

Ca si în cazul altor instructiuni de control, este posibil sã se repete un bloc de instructiuni sau o altã instructiune de control. Exemplu:

```
// determinare cmmdc prin algoritmul lui Euclid
while (a%b > 0) {
  r = a % b;       // restul impartirii a prin b
  a =b; b = r;
}            // la iesirea din ciclu b este cmmdc
```

Este posibil ca în expresia din instructiunea *while* sã se efectueze atribuiri sau apeluri de functii înainte de a compara rezultatul operatiei efectuate. Exemplu:

```
// algoritmul lui Euclid
while (r=a%b) {
  a=b; b=r;
}                // b este cmmdc
```

Instructiunea "for"

Instructiunea *for* din C permite exprimarea compactã a ciclurilor cu conditie initialã sau a ciclurilor cu numãr cunoscut de pasi si are forma:

```
for (e1; e2; e3)  i
```

Efectul acestei instructiuni este echivalent cu al secventei urmãtoare:

```
e1;          // operatii de initializare
while (e2){        // cat timp exp2 !=0 repeta
  i;          // instructiunea repetata
  e3;              // o instructiune expresie
}
```

Oricare din cele 3 expresii pot fi expresii vide, dar nu pot lipsi separatorii de expresii (caracterul ';'). Dacã lipseste "e2" atunci se considerã cã e2 are valoarea 1, deci ciclul se va repeta neconditionat. Exemplu de ciclu infinit (sau din care se va iesi cu *break* sau *return*):

```
    // repetare fara sfarsit
for ( ; ; ) instructiune       // sau while(1) instructiune
```

Cel mai frecvent instructiunea *for* se foloseste pentru programarea ciclurilor cu numãr cunoscut de pasi (cu contor). Exemple:

```
// stergere ecran prin defilare repetata de 24 ori
 for (k=1;k<=24;k++)
    putchar('\n');   // avans la linie noua

// alta secventa de stergere ecran de 25 de linii
 for (k=24;k>0;k--)
    putchar('\n');
```

Exemplul urmãtor aratã cum se poate folosi *for* în loc de *while*:

```
// determinare cmmdc pornind de la definitie
 for (d=min(a,b); a%d || b%d; d--)
                  ;    // repeta nimic

// determinare cmmdc pornind de la definitie
 d=min(a,b);       // sau o instr. "if"
 for (; a%d || b%d;)
    d--;
```

Cele trei expresii din instructiunea *for* sunt separate prin ';' deoarece o expresie poate contine operatorul virgulã (','). Este posibil ca prima sau ultima expresie sã reuneascã mai multe expresii separate prin virgule. Exemplu:

```
        // calcul factorial de n
for (nf=1, k=1 ; k<=n ; nf=nf * k, k++)
                ;           // repeta instr. vida
```

Este posibilã mutarea unor instructiuni din ciclu în paranteza instructiunii *for*, ca expresii, si invers - mutarea unor operatii repetate în afara parantezei. Pentru calculul lui n! probabil se va scrie instructiunea urmãtoare:

```
        // calcul factorial de n
for (nf=k=1 ; k<=n ; k++)
                nf = nf * k;
```

In general vom prefera programele mai usor de înteles (si de modificat) fatã de programele mai scurte dar mai criptice.

Nu se recomandã modificarea variabilei contor folositã de instructiunea *for* în interiorul ciclului, prin atribuire sau incrementare. Pentru iesire fortatã dintr-un ciclu se folosesc instructiunile break sau return si nu atribuirea unei valori mari variabilei contor.

Instructiunea "do"

Instructiunea *do-while* se foloseste pentru exprimarea ciclurilor cu conditie finalã, cicluri care se repetã cel putin o datã. Forma uzualã a instructiunii *do* este urmãtoarea:

```
do i while (e);
do { i } while (e);
```

Acoladele pot lipsi dacã se repetã o singurã instructiune, dar chiar si atunci se recomandã folosirea lor. Exemplu de utilizare a instructiunii *do*:

```
    // calcul radical din x prin aproximatii succesive
r2=x;               // aproximatia initiala
do {
  r1=r2;                // r1 este aprox. veche
  r2=(r1+x/r1) / 2;     // r2 este aprox. mai noua
} while ( abs(r2-r1)) ;     // pana cand r2==r1
```

Un ciclu *do* tipic apare la citirea cu validare a unei valori, citire repetatã pânã la introducerea corectã a valorii respective. Exemplu:

```
do {
  printf ("n (<1000): "); // n trebuie sa fie sub 1000
  scanf("%d", &n);
  if ( n <=0 || n>=1000)
          printf (" Eroare la valoarea lui n ! \n");
```

```
} while (n>1000) ;
```

Exemplu de ciclu *do* pentru verificarea unor functii cu diferite date initiale:

```
do {
  printf("x="); scanf("%f",&x);         // citeste un x
  printf ("sqrt(%f)= %lf \n", x, sqrt(x));
} while (x>0);
```

Motivatia instructiunii *do* este aceea cã expresia verificatã contine valori calculate (citite) în operatiile din ciclu, deci (aparent) expresia trebuie plasatã dupã instructiunile din ciclu si nu înaintea lor (ca în cazul instructiunii *while*). Cu pretul repetãrii unor instructiuni, un ciclu *do* poate fi rescris ca ciclu *while*

```
// echivalent cu:   do i while(e);
    i ;            // prima executie a intructiunii i
   while (e)  // daca e nevoie mai repeta
        i ;       //   instructiunea i
```

Exemplu de citire repetatã cu validare:

```
printf (“n=“); scanf (“%d”,&n);         // prima citire
while ( n > 1000) {              // daca n<=1000 se terminã
  printf (“ Eroare, repetati introducerea lui n :”);
  scanf(“%d”,&n);
}
```

Existã si alte situatii când instructiunea *do* poate reduce numãrul de instructiuni, dar în general se foloseste mult mai frecvent instructiunea *while*.

Instructiunile "break" si "continue"

Instructiunea *break* permite iesirea fortatã dintr-un ciclu sau dintr-o structurã *switch*. Sintaxa instructiunii este simplã:

```
break;
```

Efectul instructiunii *break* este un salt imediat dupã instructiunea sau blocul repetat prin *while, do, for* sau dupã blocul *switch*. Exemple:

```
// determinare cmmdc pornind de la definitie
 for (d=min(a,b); d>0; d--)
    if (a%d==0 && b%d==0)
                  break;
 printf ("%d \n",d);    // d este cmmdc(a,b)

// verifica daca un numar dat n este prim
```

```
for (k=2; k<n;k++)
   if ( n%k==0)
                break;
if (k==n) printf ("prim \n");
else printf ("neprim \n");
```

Un ciclu din care se poate iesi dupã un numãr cunoscut de pasi sau la îndeplinirea unei conditii (iesire fortatã) este de obicei urmat de o instructiune ***if*** care stabileste cum s-a iesit din ciclu: fie dupã numãrul maxim de pasi, fie mai înainte datoritã satisfacerii conditiei.

Utilizarea instructiunii ***break*** poate simplifica expresiile din ***while*** sau ***for*** si poate contribui la urmãrirea mai usoarã a programelor, desi putem evita instructiunea ***break*** prin complicarea expresiei testate în ***for*** sau ***while***. Secventele urmãtoare sunt echivalente:

```
for (k=0 ; k<n; k++)
   if (e) break;

for (k=0 ; k<n && !e ; k++)
     ;
```

Exemple de cicluri cu iesire fortatã care nu folosesc instructiunea ***break***:

```
     // verifica daca n este prim
for (k=2; k<n && n%k ; k++)
            ;
printf ( k==n? “prim”: “neprim”);

        // verifica daca n este prim
for (prim=1,k=2; k<n && prim;k++)
  if (n%k==0)
                prim=0;
printf (prim? “prim”:”neprim”);
```

Instructiunea ***continue*** este mai rar folositã fatã de ***break*** si are ca efect un salt la prima instructiune din ciclu, pentru reluarea sa. ***continue*** sare peste toate instructiunile din ciclu care-i urmeazã, dar nu iese în afara ciclului. Exemplu :

```
  // validare date citite (ore, minute, secunde)
int h,m,s;        // h=ore, m=min, s= sec
int corect=0;
while ( ! corect ) {
   printf (“ ore, minute, secunde: “);
   if ( scanf(“%i%i%i”, &h, &m, &s) !=3 ) {
      printf (“ eroare in datele citite \n”);
      fflush(stdin); continue;      // salt peste instruct. urmatoare
   }
```

```
    if (h <0 || h >24) {
        printf (“ eroare la ore \n”);
        fflush(stdin); continue;       // salt peste instruct. urmatoare
    }
    if ( m<0 || m > 59) {
        printf (“ eroare la minute \n”);
        fflush(stdin); continue;       // salt peste instruct. urmatoare
    }
    . . .      // verificare nr de secunde
    corect=1;
}
```

Instructiunea *continue* poate fi evitatã prin inversarea conditiei care o precede. Exemplu:

```
 int h,m,s;    // ora,min, sec
 int nr=0;// nr de valori citite cu scanf
 while ( 1 ) {
       printf ("ore,min,sec: ");
       nr=scanf ("%d%d%d",&h,&m,&s);
       if ( nr ==3 &&
    ( h >=0 && h <=24) &&
          ( m>=0 && m <60) &&
          ( s>=0 && s <60 ) )
             break;
       printf("Eroare in date \n");
       fflush(stdin);
 }
```

Vectori în limbajul C

Prin "vector" se întelege în programare o colectie liniarã de date omogene (toate de acelasi tip). In limba englezã se foloseste si cuvântul "array" pentru vectori si matrice. Fiecare element din vector este identificat printr-un indice întreg, pozitiv care aratã pozitia sa în vector. La o primã vedere vectorii sunt declarati si folositi în limbajul C în mod asemãnãtor cu alte limbaje. Ulterior vom arãta cã un nume de vector este similar cu un pointer si cã este posibilã o tratare diferitã a componentelor unui vector (fatã de alte limbaje).

O altã particularitate a vectorilor în C este numerotarea elementelor de la zero, deci primul element din orice vector are indicele zero, iar ultimul element dintr-un vector are un indice mai mic cu 1 decât numãrul elementelor din vector. Exemplu:

```
// suma elementelor 0..n-1 dintr-un vector a
    for (k=0, s=0; k<n; k++)
       s = s + a[k];
```

Anumite aplicatii (cu grafuri sau cu matrice de exemplu) folosesc în mod traditional o numerotare de la 1 (nu există un nod zero într-un graf). O solutie simplă este nefolosirea primei pozitii din vector (pozitia zero) si o alocare suplimentară de memorie, pentru a folosi pozitiile 1..n dintr-un vector cu n+1 elemente. Exemplu de însumare a elementelor a[1],..a[n] din vectorul a:

```
for (i=1,s=0; i<=n; i++)
    s = s + a[i];
```

Utilizarea unui vector presupune repetarea unor operatii asupra fiecărui element din vector deci folosirea unor structuri repetitive.

Exemplu de program care citeste si afisează un vector de întregi:

```
int main () {
  int a[100],n,i; // vectorul a de max 100 de intregi
  scanf ("%d",&n); // citeste nr de elemente vector
  for (i=0;i<n;i++)
    scanf ("%d", &a[i]); // citire elemente vector
  for (i=0;i<n;i++)
    printf ("%d ", a[i]); // scrie elemente vector
}
```

In exemplul anterior memoria pentru vector este alocată la compilare si nu mai poate fi modificată la executie. Programatorul trebuie să estimeze o dimensiune maximă pentru vector, care este o limită a programului. De obicei se folosesc constante simbolice pentru aceste dimensiuni si se verifică încadrarea datelor citite în dimensiunile maxime. Exemplu:

```
#define MAX 100   // dimensiune maxima vectori
int main () {
  int a[MAX], n,i;
  scanf ("%d", &n);  // citeste dimensiune efectiva
  if ( n > MAX) {
    printf ("Eroare: n > %d \n",MAX); return;
  }
  ...     // citire si utilizare elemente vector
```

La declararea unui vector se poate face initializarea partială sau integrală a componentelor, folosind o listă de constante între acolade. Exemple:

```
int azi[3]={ 01,04,2001 };      // zi,luna,an
int xmas[ ]={ 25,12,2000 };  // dimensiune=3
int prime[1000]={1,2,3};  // restul elememtelor zero
int a[1000]={0};          // toate elementele initial zero
```

Dimensiunea unui vector initializat la declarare poate rezulta din numãrul valorilor folosite la initializare. Elementele alocate si neinitializate explicit dintr-un vector initializat partial sunt automat initializate cu zero.

Introducerea de valori într-un vector se poate face numai într-un ciclu si nu printr-o singurã atribuire. Exemplu:

```
  // initializare vector a
for (i=0;i<n;i++)
   a[i]=1;
// creare vector a cu numarul de valori mai mici sau egale cu fiecare x[i]
for (i=0;i<n;i++) {
   for (j=0;j<n;j++)
      if (x[ j ] < x[ i ])
     a[ i ]++;        // a[i] = cate elem. x[j] sunt <= x[i]
}
```

De observat cã notatiile cu indici din matematicã nu se traduc automat în C pentru cã uneori elementele unui sir de numere nu sunt necesare simultan în memorie si se folosesc succesiv, putând fi memorate pe rând într-o singurã variabilã. Exemplu:

```
// calcul exp(x) prin dezvoltare in serie de puteri
     s = t =1;         // s=t[0]=1;
     for (k=1;k<=n;k++) {
        t = t * x / k ; s=s+t ;        // t[k] *= x/k; s += t[k];
     }
```

In exemplul urmãtor se face interclasarea a doi vectori ordonati într-un singur vector ordonat:

```
void main () {
  int a[100], b[100], c[200];     // reunire a si b in c
  int na, nb, nc, ia, ib, ic, k;
    // citire vectori
    ...
    // interclasare
  ia=ib=ic=1;
  while ( ia <= na && ib <= nb) {
     if ( a[ia] < b[ib] )
        c[ic++]=a[ia++];
     else
        c[ic++]=b[ib++];
  }

    // transfera in c elementele ramase in a sau in b
  for (k=ia;k<=na;k++)
     c[ic++]=a[k];
  for (k=ib;k<=nb;k++)
     c[ic++]=b[k];
```

```
  nc=ic-1;
    // afisare vector rezultat
  for (k=1;k<=nc;k++)
      printf ("%d ",c[k]);
}
```

Matrice în limbajul C

O matrice bidimensionalã este privitã în C ca un vector cu componente vectori, deci un vector de linii. Exemplu de matrice cu dimensiuni constante:

```
int a[20][10];   // maxim 20 linii si 10 coloane
```

Notatia a[i][j] desemneazã elementul din linia "i" si coloana "j" a unei matrice "a", sau elementul din pozitia 'j' din vectorul a[i].

Este posibilã initializarea unei matrice la definirea ei, iar elementele care nu sunt initializate explicit primesc valoarea zero. Exemple:

```
  float unu[3][3] = { {1,0,0}, {0,1,0}, {0,0,1} };
  int a[10][10] ={0};      // toate elementele zero
```

Prelucrarea elementelor unei matrice se face prin douã cicluri; un ciclu repetat pentru fiecare linie si un ciclu pentru fiecare coloanã dintr-o linie:

```
    // afisare matrice cu nl linii si nc coloane
  for (i=0;i<nl;i++) {
    for (j=0;j<nc;j++)
       printf ("%6d", a[i][j]);
    printf("\n");
  }
```

Numãrul de cicluri incluse poate fi mai mare dacã la fiecare element de matrice se fac prelucrãri repetate. De exemplu, la înmultirea a douã matrice, fiecare element al matricei rezultat se obtine ca o sumã:

```
 for (i=0;i<n;i++)
   for (j=0;j<m;j++) {
    c[i][j]=0;
    for (k=0;k<p;k++)
      c[i][j] += a[i][k]*b[k][j];
   }
```

In limbajul C matricele sunt liniarizate pe linii, deci în memorie linia 0 este urmatã de linia 1, linia 1 este urmatã de linia 2 s.a.m.d.

Numerotarea liniilor si coloanelor din C este diferitã de numerotarea uzualã din matematicã (care începe de la 1 si nu de la 0), folositã pentru datele initiale si

rezultatele programelor numerice. O solutie este nefolosirea liniei 0 si coloanei 0, iar altã solutie este modificarea indicilor cu 1.

In exemplul urmãtor se citesc arce dintr-un graf orientat (cu nodurile 1..n) si se construieste o matrice de adiacente în care a[i][j]=1 dacã existã arc de la nodul ‘i’ la nodul ‘j’ si a[i][j]=0 dacã nu existã arcul (i-j) :

```
short a[20][20]={0}; int i,j,n;
printf(“numar noduri: “); scanf (“%d”,&n);
printf (" lista arce: \n");
while ( scanf ("%d%d",&i,&j) == 2)
   a[i][j]=1;
printf (“ matrice de adiacente: \n”);
for (i=1;i<=n;i++) {
   for (j=1;j<=n;j++)
      printf("%2hd",a[i][j]);
  printf("\n");
}
```

Este posibilã utilizarea unei linii dintr-o matrice ca un vector. Exemplu de functie pentru însumarea valorilor absolute a elementelor dintr-un vector, folositã pentru a calcula norma unei matrice (maximul dintre sumele pe linii):

```
#define M 20      // dimensiuni maxime matrice
  // suma valori absolute dintr-un vector (cu indici 1..n)
double sumabs (double v[],int n) {
  int k; double s=0.;
  for (k=1;k<=n;k++)
      s=s+fabs(v[k]);
  return s;
}
  // norma unei matrice (cu numerotare de la 1 ptr linii si col)
double norma (double a[M][M], int nl, int nc ) {
   int i,j; double s,smax;
   smax=0;
   for (i=1;i<=nl;i++) {
      s=sumabs(a[i],nc);
      if ( smax < s)
         smax=s;
   }
   return smax;
}
```

5. Programare modularã în C

Importanta functiilor în programare

Practic nu existã program care sã nu apeleze functii din bibliotecile existente si functii definite în cadrul aplicatiei respective. Ceea ce numim uzual un "program" (o aplicatie) este o colectie de functii.

Motivele utilizãrii de subprograme sunt multiple:

- Un program mare poate fi mai usor de scris, de înteles si de modificat dacã este modular, deci format din module functionale relativ mici.
- Un subprogram poate fi reutilizat în mai multe aplicatii, ceea ce reduce efortul de programare al unei noi aplicatii.
- Un subprogram poate fi scris si verificat separat de restul aplicatiei, ceea ce reduce timpul de punere la punct a unei aplicatii mari (deoarece erorile pot apare numai la comunicarea între subprograme corecte).
- Intretinerea unei aplicatii este simplificatã, deoarece modificãrile se fac numai în anumite subprograme si nu afecteazã alte subprograme (care nici nu mai trebuie recompilate).

Utilizarea de functii permite dezvoltarea progresivã a unui program mare, fie de jos în sus ("bottom up"), fie de sus în jos ("top down"), fie combinat.

In limbajele anterioare limbajului C subprogramele erau de douã feluri:

- Functii, care au un singur rezultat, asociat cu numele functiei.
- Proceduri (subrutine), care pot avea mai multe rezultate sau nici unul, iar numele nu are asociatã nici o valoare.

In limbajul C existã numai functii, iar în loc de proceduri se folosesc functii de tip *void*. Pentru o functie cu rezultat diferit de *void* tipul functiei este tipul rezultatului functiei.

Standardul limbajului C contine si o serie de functii care trebuie sã existe în toate implementãrile limbajului. Declaratiile acestor functii sunt grupate în fisiere antet cu acelasi nume pentru toate implementãrile.

In afara acestor functii standard existã si alte functii specifice sistemului de operare, precum si functii utile pentru anumite aplicatii (graficã pe calculator, baze de date, aplicatii de retea s.a.).

Utilizarea functiilor standard din biblioteci reduce timpul de dezvoltare a programelor, mãreste portabilitatea lor si contribuie la reducerea diversitãtii programelor, cu efect asupra usurintei de citire si de întelegere a lor.

Utilizarea functiilor în C

O functie de tip *void* se va apela printr-o instructiune expresie. Exemple:

```
printf ("\n n=");
clearerr (stdin);        // sterge indicator de eroare si EOF
```

O functie de un tip diferit de *void* este apelatã prin folosirea ei ca operand într-o expresie. Exemple:

```
z=sqrt(x)+ sqrt(y);
printf ("%lf \n", sqrt(x));
comb = fact(n) / ( fact(k) * fact(n-k));  // combinari
y = atan (tan(x));              // functie in functie
```

In limbajul C este uzual ca o functie sã raporteze prin rezultatul ei (numãr întreg) modul de terminare (normal/cu eroare) sau numãrul de valori citite/scrise (la functiile de intrare-iesire). Uneori acest rezultat este ignorat iar functia cu rezultat este apelatã ca o functie *void*. Exemple:

```
scanf ("%d",&n);     // rezultatul lui scanf este 1
getchar();           // rezultatul este caracterul citit
gets(adr);      // rezultatul este adresa "adr"
```

Argumentele folosite la apelul functiei se numesc argumente efective si pot fi orice expresii (constante, functii etc.). Argumentele efective trebuie sã corespundã ca numãr si ca ordine (ca semnificatie) cu argumentele formale (cu exceptia unor functii cu numãr variabil de argumente). Exemplu de functie unde ordinea argumentelor este importantã:

```
   // calculul unui unghi dintr-un triunghi
double unghi (double a, double b, double c) {
   return acos ((b*b+c*c-a*a) / (2.*b*c));     // unghiul A
}
    // utilizari
  ua = unghi (a,b,c); ub=unghi (b,c,a); uc = unghi (c,c,b);
```

Este posibil ca tipul unui argument efectiv sã difere de tipul argumentului formal corespunzãtor, cu conditia ca tipurile sã fie "compatibile" la atribuire. Conversia de tip (între numere sau pointeri) se face automat, la fel ca si la atribuire. Exemple:

```
x=sqrt(2);   // arg. formal "double", arg.efectiv "int"
y=pow(2,3);     //  arg. formale de tip "double"
```

Deci o functie cu argument formal de un tip numeric (de ex. *int*) poate fi apelatã cu argumente efective de orice tip numeric (inclusiv *long, float, double, long double*).

Conversia automatã nu se face si pentru argumente vectori de numere, iar conversia explicitã de tip conduce la erori de interpretare. Exemplu:

```
    // suma elemente vector de intregi
  float suma ( float a[ ], int n) {
```

```
   int k; float s=0;
   for (k=0;k<n;k++)
      s=s+a[k];
   return s;
 }
#include <stdio.h>
void main () {
  int x[ ]={1,2,3,4,5};
  printf ("%f \n", suma(x,5));  // nu scrie 15 !
}
```

De retinut cã unele erori de utilizare a functiilor nu pot fi detectate de compilator si se pot manifesta la executie prin rezultate gresite. Exemplu:

```
printf("%d",pow(10,3));   // (int) pow(10,3)
```

Definirea de functii în C

Sintaxa definirii functiilor în C s-a modificat de la prima versiune a limbajului la versiunea actualã (standardizatã), pentru a permite verificarea utilizãrii corecte a oricãrei functii la compilare. Forma generalã a unei definitii de functie, conform standardului, este:

```
tipf numef (tip1 arg1, tip2 arg2, ...) {
  declaratii
  instructiuni (blocuri)
}
```

unde:
tipf este tipul functiei (tipul rezultatului sau *void*)
tip1, tip2,... sunt tipurile argumentelor (parametrilor) functiei

Tipul unei functii C poate fi orice tip numeric, orice tip pointer, orice tip structurã (*struct*) sau *void*.

Argumentele formale pot fi doar nume de variabile (fãrã indici) sau nume de vectori, deci nu pot fi expresii sau componente de vectori.

Exemplu de functie de tip *void*:

```
  // sterge ecran prin defilare cu 24 de linii
void erase () {
  int i;
  for (i=0;i<24;i++)
    printf("\n");
}
```

Este preferabil ca definitia functiei "erase" sã preceadã definitia functiei "main" (sau a unei alte functii care o apeleazã). Dacã functia "erase" este definitã dupã functia "main" atunci este necesarã o declaratie pentru functia "erase" înaintea functiei "main":

```
void erase ();        // declaratie functie
 void main () {
   erase(); . . .          // utilizare functie
 }
 void erase() {
   . . .             // definitie functie
 }
```

Când se declarã prototipul unei functii cu argumente este suficient sã se declare tipul argumentelor, iar numele argumentelor formale pot lipsi. Exemplu:

```
double unghi(double, double, double);        // 3 argumente double
```

In lipsa unei declaratii de tip explicite se considerã cã tipul implicit al functiei este *int*. Functia "main" poate fi declaratã fie de tip *void*, fie de tip *int*.

Si argumentele formale fãrã un tip declarat explicit sunt considerate implicit de tipul *int*, dar nu trebuie abuzat de aceastã posibilitate. Exemplu:

```
  rest (a,b) {      // int rest (int a, int b)
    return a%b;
  }
```

Variabilele definite într-o functie pot fi folosite numai în functia respectivã, cu exceptia celor declarate *extern*. Pot exista variabile cu aceleasi nume în functii diferite, dar ele se referã la adrese de memorie diferite.

O functie are în general un numãr de argumente formale (fictive), prin care primeste datele initiale necesare si poate transmite rezultatele functiei. Aceste argumente pot fi doar nume de variabile (nu orice expresii) cu tipul declarat în lista de argumente, pentru fiecare argument în parte. Exemplu:

```
int comb (int n, int k) {      // combinari de n luate cate k
  int i, cmb=1;
  for (i=1;i<=k;i++)
    cmb = cmb * (n-i+1) / i;
  return cmb;
}
```

Se recomandã ca o functie sã îndeplineascã o singurã sarcinã si sã nu aibã mai mult de câteva zeci de linii sursã (preferabil sub 50 de linii).

In limbajul C se pot defini si functii cu numãr variabil de argumente, care pot fi apelate cu numãr diferit de argumente efective. Exemplu de functie pentru adunarea unui numãr oarecare de valori:

```
#include <stdarg.h>
int va_add( int na, ...) {              // na= numar de argumente
  va_list ap;                           // tip definit in <stdarg.h>
  int i,sum = 0;
  va_start(ap, na);        // macro din <stdarg.h>
  for (i=0;i<na;i++)
    sum += va_arg(ap,int);
  va_end(ap);              // macro din <stdarg.h>
  return sum;
}
   // exemple de apelare
  va_add(4, 10, 20, 30, 40);   // cu 4 argumente
  va_add(2, 500, 700);      // cu 2 argumente
```

Instructiunea "return"

Instructiunea ***return*** se foloseste pentru revenirea dintr-o functie apelatã la functia care a fãcut apelul si poate contine o expresie ce reprezintã rezultatul functiei. Conversia rezultatului la tipul functiei se face automat, dacã e posibil

O functie de un tip diferit de ***void*** trebuie sã continã cel putin o instructiune ***return*** prin care se transmite rezultatul functiei. Exemplu:

```
    long fact (int n) {     // factorial de n
       long nf=1L;          // ptr calcul rezultat
       while ( n)
          nf=nf * n--;      // nf=nf * n; n=n-1;
       return nf;        // rezultat functie
    }
```

Compilatorul "gcc" (folosit si de mediul Dev-Cpp) nu verificã si nu semnaleazã dacã o functie de un tip diferit de ***void*** contine sau nu instructiuni ***return***, iar eroarea se manifestã numai la executie. Exemplu:

```
long fact (int k) {                     // calcul factorial
  long kf=1L;
  while (k > 0 )
     kf=kf*k--;
//  return kf;                  // corect este fara comentariu
}
long comb (int n, int m) {       // calcul combinari
  return fact(n) / (fact(m)*fact(n-m));          // eroare la executie !
}
```

O functie poate contine mai multe instructiuni *return*. Exemplu:

```
char toupper (char c) {    // trece car. c in litere mari
  if (c>='a'&& c<='z')  // daca c este litera mica
    return c+'A'-'a';          // cod litera mare
  else                  // altceva decat litera mica
    return c;          // ramane neschimbat
}
```

Cuvântul *else* dupã o instructiune *return* poate lipsi, dar de multe ori este prezent pentru a face codul mai clar. Exemplu fãrã *else*:

```
char toupper (char c) {    // trece car. c in litere mari
  if (c>='a' && c<='z')      // daca c este litera mica
    return c+'A'-'a';              // cod litera mare
  return c;                    // ramane neschimbat
}
```

Instructiunea *return* poate fi folositã pentru iesirea fortatã dintr-un ciclu si din functie, cu reducerea lungimii codului sursã. Exemplu:

```
// verifica daca un numar dat este prim
  int esteprim (int n) {
    int k;
    for (k=2; k<=n/2; k++)
     if (n % k==0)
        return 0;               // nu este prim
    return 1;              // este prim
  }
```

Functia pentru verificarea unui numãr dacã este prim se putea scrie si asa:

```
  int esteprim (int n) {
    int k, prim=1;
    for (k=2; k<=n/2; k++)
      if (n % k==0) {
        prim=0; break;
      }
    return prim;          // 1 daca este prim
  }
```

Dacã tipul expresiei din instructiunea *return* diferã de tipul functiei atunci se face o conversie automatã, pentru tipuri numerice. Exemplu:

```
int sqr (int x ) {
  return pow(x,2);        // conversie de la double la int
}
int main () {
```

```
   printf ("%d \n", sqr(3));
}
```

Intr-o functie de tip *void* se poate folosi intructiunea *return* fãrã nici o expresie, iar dacã lipseste se adaugã automat ca ultimã instructiune.

In functia “main” instructiunea *return* are ca efect terminarea întregului program.

Functii cu argumente vectori

O functie C nu poate avea ca rezultat direct un vector, dar poate modifica elementele unui vector primit ca argument. Exemplu:

```
// genereaza vector cu cifrele unui nr.natural dat n
  void cifre (int n, char c[5] ) {
     int k;
     for (k=4;k>=0;k--) {
        c[k]=n%10;      // cifra din pozitia k
        n=n/10;
     }
  }
```

In exemplul anterior vectorul are dimensiune fixã (5) si contine toate zerourile initiale, dar putem defini o functie de tip *int* cu rezultat egal cu numãrul cifrelor semnificative. Pentru argumentele formale de tip vector nu trebuie specificatã dimensiunea vectorului. Exemplu:

```
float maxim (float a[ ], int n ) {
          int k;
     float max=a[0];
     for (k=1;k<n;k++)
       if ( max < a[k])
          max=a[k];
     return max;
       }
     // exemplu de utilizare
     float xmax, x[ ]= {3,6,2,4,1,5,3};
      xmax = maxim (x,7);
```

De remarcat cã pentru un argument efectiv vector nu mai trebuie specificat explicit cã este un vector, deoarece existã undeva o declaratie pentru variabila respectivã, care stabileste tipul ei. Este chiar gresit sintactic sã se scrie:

```
xmax =maxim (x[ ],7);      // nu !
```

Un argument efectiv poate fi însã un element dintr-un vector. Exemplu:

```
for (k=0;k<n;k++)
```

```
    printf (“%f \n”, sqrt(x[k]));
```

Un argument formal de tip vector este echivalent cu un pointer, iar functia primeste adresa zonei de memorie unde se aflã un vector sau unde se va crea un vector. Functia poate sã si modifice componentele unui vector primit ca argument. Exemplu de functie pentru ordonarea unui vector:

```
void sort (float a[ ],int n) {
  int n,i,j, gata; float aux;
    // repeta cat timp mai sunt schimbari de elemente
  do {
    gata =1;
      // compara n-1 perechi vecine
     for (i=0;i<n-1;i++)
    if ( a[i] > a[i+1] ) {        // daca nu sunt in ordine
      // schimba pe a[i] cu a[i+1]
      aux=a[i]; a[i]=a[i+1]; a[i+1]=aux;
      gata =0;
    }
  } while ( ! gata);
}
```

Probleme pot apare la argumentele de functii de tip matrice din cauza interpretãrii diferite a zonei ce contine elementele matricei de cãtre functia apelatã si respectiv de functia apelantã. Pentru a interpreta la fel matricea liniarizatã este important ca cele douã functii sã foloseascã acelasi numãr de coloane în formula de liniarizare. Din acest motiv nu este permisã absenta numãrului de coloane din declaratia unui argument formal matrice. Exemplu incorect sintactic:

```
    void printmat ( int a[ ][ ], int nl, int nc);          // gresit !
```

O solutie simplã dar care nu e posibilã întotdeauna ar fi specificarea aceleeasi constante pentru numãr de coloane în toate functiile si în definitia matricei din programul principal. Exemplu:

```
    void printmat(int a[ ][10], int nl, int nc);          // nc <= 10
```

Multe compilatoare considerã tipul argumentului “a” ca fiind “pointer la un vector de 10 întregi” si nu ca “pointer la pointer la întreg”, pentru a forta transmiterea numãrului de coloane si a evita erori de transmitere a matricelor la functii. Pentru functiile de bibliotecã nu se poate preciza numãrul de coloane si trebuie gãsite alte solutii de definire a acestor functii si/sau de alocare a matricelor.

Transmiterea de date între functii

Transmiterea argumentelor efective la apelul unei functii se face în C prin copierea valorilor argumentelor efective în argumentele formale (care sunt variabile locale ale functiei). In acest fel functia apelatã lucreazã cu duplicate ale variabilelor argumente efective si nu poate modifica accidental variabile din functia apelantã.

Compilatorul C genereazã o secventã de atribuiri la argumentele formale înainte de efectuarea saltului la prima instructiune din functia apelatã. Din acest motiv toate conversiile de tip efectuate automat la atribuire se aplicã si la transmiterea argumentelor.

In functia urmãtoare se modificã aparent valoarea lui k dar de fapt se modificã o variabilã localã, fãrã sã fie afectatã variabila ce contine pe k în "main":

```
long fact (int k) {                    // calcul factorial
  long kf=1L;
  while (k > 0 )
     kf = kf * k--;                 // k=k-1 dupa inmultire
  return kf;                        // rezultat functie
}
```

Un alt exemplu clasic este o functie care încearcã sã schimbe între ele valorile a douã variabile, primite ca argumente:

```
    void swap (int a, int b) {    // nu este corect !!!
       int aux;
       aux=a; a=b; b=aux;
    }
    void main () {
      int x=3, y=7;
      swap(x,y);
      printf ("%d,%d \n",x,y);       // scrie 3,7 !
    }
```

In general o functie C nu poate transmite rezultate si nu poate modifica argumente de un tip numeric. In C pentru transmiterea de rezultate prin argumente de cãtre o functie trebuie sã folosim argumente formale de tip pointer (adrese de memorie).

Versiunea corectã pentru functia “swap” este urmãtoarea:

```
 void swap (int * pa, int * pb) {        // pointeri la intregi
   int aux;
   aux=*pa; *pa=*pb; *pb=aux;
 }
```

Apelul acestei functii foloseste argumente efective pointeri:

```
    int x, y ; . . .
    swap (&x, &y) ;  // schimba valorile x si y intre ele
```

Pentru variabilele locale memoria se alocã la activarea functiei (deci la executie) si este eliberatã la terminarea executãrii functiei. Initializarea variabilelor locale se face tot la executie si de aceea se pot folosi expresii pentru initializare (nu numai constante). Exemplu:

```
double arie (double a, double b, double c) {
   double p = (a+b+c)/2.;       // initializare cu expresie
   return sqrt(p*(p-a)*(p-b)*(p-c));
}
```

Practic nu existã nici o diferentã între initializarea unei variabile locale la declarare sau printr-o instructiune de atribuire.

Functiile pot comunica date între ele si prin variabile externe, definite înaintea functiilor care le folosesc. Exemplu:

```
int a[20][20],n;      // variabile externe
void citmat() {       // citire matrice
  int i,j;
  printf ("n="); scanf("%d",&n);  // dimensiuni
  for (i=0;i<n;i++)       // citire matrice
    for (j=0;j<n;j++)
      scanf("%d",&a[i][j]);
}
void scrmat() {      // afisare matrice
  int i,j;
  for (i=0;i<n;i++) {
    for (j=0;j<n;j++)
      printf("%5d",a[i]);
    printf("\n");       // dupa fiecare linie
  }
}
```

Nu se recomandã utilizarea de variabile externe decât în cazuri rare, când mai multe functii folosesc în comun mai multe variabile si se doreste simplificarea utilizãrii functiilor, sau în cadrul unor biblioteci de functii. Mai putem folosi variabile externe în cazul unor functii recursive pentru reducerea numãrului de argumente (se scot argumentele care nu se modificã între apeluri).

Functii recursive

O functie (direct) recursivã este o functie care se apeleazã pe ea însãsi. Se pot deosebi douã feluri de functii recursive:

- Functii cu un singur apel recursiv, ca ultimã instructiune, care se pot rescrie usor sub forma nerecursivã (iterativã).
- Functii cu unul sau mai multe apeluri recursive, a cãror formã iterativã trebuie sã foloseascã o stivã pentru memorarea unor rezultate intermediare.

Recursivitatea este posibilã în C datoritã faptului cã, la fiecare apel al functiei, adresa de revenire, variabilele locale si argumentele formale sunt puse într-o stivã (gestionatã de compilator), iar la iesirea din functie (prin *return*) se scot din stivã toate datele puse la intrarea în functie (se "descarcã" stiva).

Exemplu de functie recursivã de tip *void* :

```
void binar (int n) {          // se afiseaza n in binar
  if (n>0) {
     binar(n/2);              // scrie echiv. binar al lui n/2
     printf("%d",n%2);        // si restul impartirii n la 2
  }
}
```

Functia de mai sus nu scrie nimic pentru n=0, dar poate fi usor completatã cu o ramurã *else* la instructiunea *if*.

Orice functie recursivã trebuie sã continã (cel putin) o instructiune *if* (de obicei chiar la început), prin care se verificã dacã (mai) este necesar un apel recursiv sau se iese din functie. Reamintim cã orice functie *void* primeste implicit o instructiune *return* ca ultimã instructiune. Absenta instructiunii *if* conduce la o recursivitate infinitã (la un ciclu fãrã conditie de terminare).

Pentru functiile de tip diferit de *void* apelul recursiv se face printr-o instructiune *return*, prin care fiecare apel preia rezultatul apelului anterior.

Anumite functii recursive corespund unor relatii de recurentã. Exemplu:

```
long fact (int n) {
  if (n==0)
    return 1L;          // 0! = 1
  else
    return n * fact(n-1);       // n!=n*(n-1)!
}
```

Algoritmul lui Euclid poate folosi o relatie de recurentã:

```
int cmmdc (int a,int b) {
   if ( a%b==0)
      return b;
   return cmmdc( b,a%b); // cmmdc(a,b)=cmmdc(b,a%b)
}
```

Pentru determinarea cmmdc mai existã si o altã relatie de recurentã.

Functiile recursive nu contin în general cicluri explicite (cu unele exceptii), iar repetarea operatiilor este obtinutã prin apelul recursiv.

O functie care contine un singur apel recursiv ca ultimã instructiune poate fi transformatã într-o functie nerecursivã, înlocuind instructiunea *if* cu *while*.

```
int fact (int n) {            // recursiv
```

```
    if (n>0)
      return n*fact(n-1);        // n!=n*(n-1)!
    else
      return 1;             // 0! = 1
  }
```

Functiile recursive cu mai multe apeluri sau cu un apel care nu este ultima instructiune pot fi rescrise iterativ numai prin folosirea unei stive. Aceastã stivã poate fi un simplu vector local functiei. Exemplu:

```
void binar ( int n) {      // afisare in binar
  int c[16],i;       // c este stiva de cifre
   // pune resturi in stiva c
  i=0;
  while ( n>0) {
    c[i++]=n%2;
    n=n/2;
  }
    // descarca stiva: scrie vector in ordine inversa
  while (i>0)
    printf ("%d",c[--i]);
}
```

Exemplul urmãtor este o functie recursivã cu argument vector :

```
double max2 (double a, double b) {      // maxim dintre doua valori
   return a > b ? a:b;
}
double maxim (double a[ ], int n) {       // maxim dintr-un vector
  if (n==1)
      return a[0];
  else
      return max2 (maxim (a,n-1),a[n-1]);
}
```

Fiecare apel recursiv are parametri diferiti, sau mãcar o parte din parametri se modificã de la un apel la altul.

Varianta recursivã a unor functii poate necesita un parametru în plus fatã de varianta nerecursivã pentru aceeasi functie, dar diferenta se poate elimina printr-o altã functie. Exemplul urmãtor este o functie recursivã de cãutare binarã într-un vector ordonat, care reduce succesiv portiunea din vector în care se cautã, portiune definitã prin doi indici întregi.

```
  // cauta b între a[i] si a[j]
int cautb (int b, int a[], int i, int j) {
 int m;                                  // indice median intre i si j
  if ( i > j)                            // daca indice inferior mai mare ca indice superior
```

```
   return -1;                          // atunci b negasit in a
  m=(i+j)/2;                           // m = indice intre i si j (la mijloc)
  if (a[m]==b)
     return m;
  if (b < a[m])                            // daca b in prima jumatate
    return cautb (b,a,i,m-1);          // atunci se cauta intre a[i] si a[m-1]
  else                                 // daca b in a doua jumatate
     return cautb (b,a,m+1,j);         // atunci se cauta intre a[m+1] si a[j]
 }
  // functie auxiliara cu mai putine argumente
int caut (int b, int a[], int n) {     // n este dimensiunea vectorului a
   return cautb (b,a,0,n-1);
}
```

Este posibilã si o recursivitate mutualã sau indirectã între douã sau mai multe functii f1 si f2 , de forma f1→ f2 → f1→ f2 → … In astfel de cazuri trebuie declarate functiile apelate, deoarece nu se pot evita declaratiile prin ordinea în care sunt definite functiile. Exemplu:

```
void f1 ( ) {
    void f2 ( );   // functie apelata de f1
     …
     f2( );        // apel f2
}
void f2 ( ) {
    void f1 ( );   // functie apelata de f2
     …
     f1( );        // apel f1
}
```

Altã variantã de scriere:

```
 // declaratii functii
void f1 ( );
void f2 ( );
 // definitii functii
void f1 ( ) {
     …
     f2( );        // apel f2
}
void f2 ( ) {
     …
     f1( );        // apel f1
}
```

6. Programare structuratã în limbajul C

Structuri de control

Instructiunile de control dintr-un limbaj permit selectarea si controlul succesiunii în timp a operatiilor de prelucrare. In limbajele masinã si în primele limbaje de programare controlul succesiunii se realiza prin instructiuni de salt în program (instructiunea *go to* mai existã si în prezent în C si în alte limbaje, desi nu se recomandã utilizarea ei).

S-a demonstrat teoretic si practic cã orice algoritm (program) poate fi exprimat prin combinarea a trei structuri de control:

- secventa liniarã de operatii
- decizie binarã (alegere dintre douã alternative posibile)
- ciclul cu conditie initialã (repetarea unor operatii în functie de o conditie)

Limbajul C este un limbaj de programare structuratã deoarece posedã instructiuni pentru exprimarea directã a acestor trei structuri de control, fãrã a se mai folosi instructiuni de salt. Aceste instructiuni sunt blocul, *if* si *while*.

Limbajul C contine si alte structuri de control, pe lângã cele strict necesare:

- selectie multiplã (dintre mai multe alternative)
- ciclul cu conditie finalã (verificatã dupã executarea operatiilor din ciclu)
- ciclul *for* (cu conditie initialã sau cu numãr cunoscut de pasi)

Combinarea structurilor de control se face prin includere; orice combinatie este posibilã si pe oricâte niveluri de adâncime (de includere). Deci un ciclu poate contine o secventã sau o decizie sau un alt ciclu, s.a.m.d.

Pentru evidentierea structurilor de control incluse se practicã scrierea decalatã (indentatã): fiecare structurã inclusã se va scrie decalatã la dreapta cu un caracter Tab sau cu câteva blancuri; la revenirea în structura de nivel superior se revine la alinierea cu care a început acea structurã. Exemplu:

```
  // secventele de numere naturale consecutive a caror suma este egala cu n
int main () {
 int n, m, k, j, s;
 printf("n= "); scanf ("%d",&n);
 m= n/2+1;           // unde poate incepe o secventa
 for (i=1;i<m;i++) {         // i = inceput secventa
    s=i;
    for (j=i+1;j<=k; j++) {      // j = sfarsit secventa
      s=s+j;
      if (s>=n)
         break;
    }
       // afisare numere între i si j
    if (s==n) {
       for (k=i;k<=j;k++)        // scrie secventa i,i+1,...j
         printf ("%d ",k);
```

```
      printf ("\n");
    }
  }
 }
```

Reducerea numãrului de structuri incluse unele în altele se poate face prin definirea si utilizarea de functii care sã realizeze o parte din operatii. Exemplu

```
// determina sfarsit secventa de nr naturale care incepe cu k si are suma n
int end ( int k, int n) {
  int j,s=0;
  for (j=k;s<n;j++)
      s=s+j;
  if ( s == n)
      return j -1;
  else
      return -1;    // nu exista secventa cautata
}
   // toate secventele de numere consecutive  a caror suma este  n
int main () {
 int n,k,i,j;
 printf("n= "); scanf ("%d",&n);
 for (i=1;i<n;i++)
     if ( (j=end(i,n)) > 0) {
        for (k=i;k<=j;k++)
      printf ("%d ",k);
        printf ("\n");
     }
}
```

Programare structuratã în limbajul C

Programarea structuratã în C are câteva particularitãti fatã de programarea în Pascal, prin existenta instructiunilor ***break, continue*** si ***return*** care permit salturi în afara sau în cadrul unor structuri.

Instructiunile ***break*** si ***return*** permit simplificarea programãrii ciclurilor cu iesire fortatã, fãrã a folosi variabile auxiliare sau conditii compuse. Exemplu:

```
    // cautarea primei aparitii a lui b in vectorul a
  int index (int a[ ],int n, int b) {
    int k;
    for (k=0;k <n;k++)
       if (b==a[k])
          return k;          // gasit in pozitia k
     return -1;              // negasit
  }
```

Solutia stil Pascal pentru cãutarea într-un vector neordonat este:

```
int index (int a[ ],int n, int b) {
    int k, este= -1;
    for (k=0;k <n && este < 0;k++)
       if (b==a[k])
          este=k;
     return este;
  }
```

Desi există o instructiune *goto* în limbajul C se pot scrie orice programe fără a recurge la această instructiune.

O situatie care ar putea justifica folosirea instructiunii *goto* ar fi iesirea dintr-un ciclu interior direct în afara ciclului exterior. Exemplu:

```
  // cauta prima aparitie a lui b in matricea a
for (i=0;i<n;i++)
   for (j=0;j<n;j++)
      if ( a[i][j]==b )
        goto gasit;
 printf ("negasit \n");
 return;
 gasit:
   printf("gasit in linia %d si coloana %d \n", i, j);
```

Un ciclu *for* care contine o instructiune *if* (fără *break*) este în general diferit de un ciclu *while*, deoarece repetarea ciclului *while* se opreste la primul test cu rezultat neadevărat în timp ce un ciclu *for* repetă toate testele indiferent de rezultatul lor. Exemple:

```
  // verifica daca un vector este ordonat crescator
int cresc (int a[ ], int n) {
  int k=1;
  while ( k < n && a[k-1]<a[k] )
        k++;
   return k==n;
}
// utilizare
int main () {
  int x[ ]={1,2,3,0};
  printf ("%d\n", cresc(x,4));
}
```

Solutii alternative

In general logica de rezolvare a unei probleme impune structurile de control folosite, dar uneori avem de ales între două sau mai multe alternative de codificare a unui algoritm.

Primul exemplu este o problemã uzualã la afisarea unui numãr mare de valori pe ecran sau la imprimantã: numerele afisate vor fi grupate pe mai multe coloane. Se pot considera douã cicluri incluse: un ciclu pentru fiecare linie afisatã si un ciclu pentru numerele dintr-o linie. Exemplu:

```
 // afisarea a n valori pe nc coloane
void print ( int x[ ],int n,int nc) {
  int nl,k,j;
  nl = n/nc+1;          // nr de linii afisate
  k=0;
  for (i=0;i<nl;i++) {
      for (j=0; j<nc && k<n; j++)
          printf ("%6d",x[k++]);
      printf ("\n");
  }
}
```

Un alt punct de vedere este acela cã avem un singur ciclu de afisare, dar la îndeplinirea unei conditii se trece la linie nouã. Exemplu:

```
void print ( int x[ ],int n,int nc) {
  int i;
  for (i=0;i<n;i++) {
     if ( i % nc ==0)
        printf ("\n");
     printf ("%6d",x[i]);
  }
}
```

Numãrul de coloane “nc” este transmis la apelul functiei, dar el poate fi calculat de functie raportând lungimea liniei la numãrul de cifre zecimale pe care îl are cea mai mare valoare absolutã din vectorul dat.

In problema urmãtoare se dã un vector de coduri ale unor produse si un vector de cantitãti ale acestor produse si se cere totalizarea cantitãtilor pentru fiecare produs în parte. Exemplu de vector de coduri neordonat, cu repetarea unor coduri:

```
Cod :   2   7   2   3   7   2   3   7   2
Cant:      10   10 10 10 10 10 10 10 10
```

Rezultate pentru aceste date:

```
Cod:    2   7   3
Cant:  40  30 20
```

Dacã vectorul de coduri este ordonat se pot utiliza douã cicluri *while* : un ciclu (interior) repetat pentru produsele cu acelasi cod, inclus într-un ciclu (exterior) repetat cât timp mai existã elemente în vectori.

```
   // totalizare cu doua cicluri while
 i=0;
  while (i < n) {
     c=cod[i]; sum=0;
     while ( c == cod[i] )
          sum=sum+val[i++];
     printf ("%6d  %6d \n", c,sum);
  }
```

In locul celor douã cicluri se poate folosi un singur ciclu *for*, repetat pentru toate elementele vectorului, care contine un *if* pentru a verifica trecerea de la un produs la altul (schimbarea codului la înaintarea în vectorul de coduri).

```
   // totalizare cu un singur ciclu
 c=cod[0]; sum=val[0];
 for (i=1;i<n;i++) {
    if ( c == cod[i])
        sum=sum+val[i];
    else {
        printf ("%6d  %6d \n",c,sum);
        c=cod[i];  sum=val[i];
    }
 }
 printf ("%6d  %6d \n",c,sum);    // ultima grupa
```

In exemplul urmãtor trebuie sã clasificãm n valori x[1]..x[n] în m intervale cu limitele a[0],a[1],..a[m]. Presupunem fãrã verificare cã limitele sunt ordonate crescãtor si cã toate valorile x[i] sunt cuprinse între a[0] si a[m]. Functia creeazã un vector k de m-1 numere în care k[j] este numãrul de valori x cuprinse în intervalul j (j între 1 si m). In prima variantã se numãrã succesiv valorile din fiecare interval j.

```
  //  varianta 1
void histo (float x[ ],int n, float a[ ], int m, int k[ ]) {
   int i,j;
   for (j=1;j<=m;j++) {
      k[j]=0;
      for (i=1;i<=n;i++)
          if ( x[i] > a[j-1] && x[i] <= a[j] )
      k[j]++;
   }
}
```

In varianta urmãtoare se clasificã succesiv x[1], x[2],.. x[n].

```
  // varianta 2
```

```
void histo (float x[ ],int n, float a[ ], int m, int k[ ]) {
  int i,j;
  for (j=1;j<m;j++)
     k[j]=0;
  for (i=1;i<=n;i++)
      for (j=1;j<=m;j++)
         if ( x[i] <= a[j] ) {
            k[j]++; break;
         }
}
```

Eficienta programelor

De multe ori avem de ales între mai multe variante corecte ale unei functii si care diferã între ele prin timpul de executie. In general vom prefera secventele de instructiuni mai eficiente ca timp.

Metodele de reducere a timpului de executie pot fi împãrtite în metode cu caracter general si metode specifice fiecãrei probleme de rezolvat.

Dintre metodele cu caracter general mentionãm:

- Se scot din cicluri apeluri de functii si alte calcule care pot fi efectuate o singurã datã, înainte de intrarea în ciclu.

In exemplul urmãtor se genereazã toate numerele naturale de n cifre:

```
for (k=1;k< pow(10,n)-1;k++) { ... }
```

Calculul celui mai mare numãr de n cifre se poate face în afara ciclului:

```
max=(int)pow(10,n) -1;
for (k=1;k<max;k++) { ... }
```

- Intreruperea ciclurilor de verificare a elementelor unor vectori cu prima ocazie când poate fi trasã o concluzie, fãrã a mai testa si elementele rãmase. De exemplu, functia urmãtoare care verificã dacã un vector este ordonat crescãtor nu se opreste la prima pereche de elemente adiacente neordonate:

```
// verifica daca un vector este ordonat crescator
int cresc (int a[ ], int n) {
  int k, este=1;
  for ( k=1; k < n; k++ )
       if (a[k-1] > a[k])
          este=0;
   return este;
}
```

- Evitarea apelurilor repetate, inutile de functii. Exempluîn care se determinã cel mai lung segment din toate segmentele care unesc n puncte date prin coordonatele lor.

```
float dmax ( float x[ ], float y[ ], int n) {
  int i,j; float d, dmax=0;
  for (i=0;i<n-1;i++)
     for (j=i+1;j<n;j++) {
        d= dist(x[i],y[i],x[j],y[j]);       // distanta dintre punctele i si j
     if ( d > dmax)
       dmax=d;
     }
   return dmax;
}
```

O programare inabilã ar fi putut arãta astfel :

```
dmax=0;
for (i=0;i<n-1;i++)
     for (j=i+1;j<n;j++)
          if (dmax < dist(x[i],y[i],x[j],y[j]))    // un apel al functiei dist
             dmax= dist(x[i],y[i],x[j],y[j]) ;          // alt apel cu aceleasi argumente
```

- Utilizarea de macrouri sau de functii “in-line” pentru functii mici dar apelate de mai multe ori într-un program. Anumite “functii” de bibliotecã sunt de fapt macrouri expandate la fiecare apel.

Vom exemplifica câteva metode de reducere a timpului de executie care pot fi aplicate numai în anumite cazuri specifice.

Ridicarea la pãtrat se va face prin înmultire si nu folosind functia “pow”:

```
double  sqr (double x) {          // functie de ridicare la patrat
   return x * x;
}
```

Polinomul de interpolare Newton are forma urmãtoare:

$$P(a) = c[0] + c[1]*(a-x[1]) + c[2] *(a-x[1])*(a-x[2]) + c[3] *(a-x[1])*(a-x[2])*(a-x[3]) + ...$$

Calculul valorii acestui polinom înseamnã o sumã de produse si, aparent, se realizeazã prin douã cicluri incluse:

```
float valPolN (float c[ ], float x[ ], int n, float a) {
 float s,p; int i, j;
 s=c[0];
 for (i=1;i<n;i++) {
   p=1;
   for (j=1;j<=i;j++)
```

```
      p=p * (a-x[j]);
   s=s+c[i] *p;
 }
 return s;
}
```

Observãm cã fiecare produs diferã de produsul din termenul anterior printr-un singur factor, deci putem calcula într-un singur ciclu suma de produse:

```
float valPolN (float c[ ], float x[ ], int n, float a) {
  float s,p; int i;
  s=c[0]; p=1;
  for (i=1;i<n;i++) {
     p=p * (a-x[i]);
     s=s+c[i]*p;
  }
  return s;
}
```

Diferenta de timp dintre cele douã solutii este insesizabilã pentru valori uzuale ale lui n (< 50). In general, optimizarea programelor este justificatã pentru probleme de dimensiuni mari sau pentru secvente de program repetate frecvent, din programe des utilizate.

Existã însã anumite categorii de probleme unde diferenta dintre un algoritm sau altul este semnificativã si ea creste accelerat odatã cu dimensiunea problemei. Studiul complexitãtii algoritmilor este un subiect de programare avansatã si este independent de limbajul de programare utilizat.

7. Tipuri structurã în C

Definirea de tipuri si variabile structurã

Un tip structurã reuneste câteva variabile (numite "câmpuri"), având fiecare un nume si un tip. Tipurile câmpurilor unei structuri pot fi si sunt în general diferite. Definirea unui tip structurã are sintaxa urmãtoare:

```
struct tag { tip1 c1, tip2 c2, …  };
```

unde:
"tag" este un nume de tip folosit numai precedat de cuvântul cheie ***struct*** (în C, dar în C++ se poate folosi singur ca nume de tip).
"tip1","tip2",... este tipul unei componente
"c1","c2",... este numele unei componente (câmp)
Ordinea enumerãrii câmpurilor unei structuri nu este importantã, deoarece ne referim la câmpuri prin numele lor. Se poate folosi o singura declaratie de tip pentru mai multe câmpuri. Exemple:

```
    // momente de timp (ora,minut,secunda)
struct time {
  int ora,min,sec;
};
    // o activitate
struct activ {
  char numeact[30];      // nume activitate
  struct time start;     // ora de incepere
  struct time stop;      // ora de terminare
};
```

De remarcat cã orice declaratie ***struct*** se terminã obligatoriu cu caracterul ';' chiar dacã acest caracter urmeazã dupã o acoladã; aici acoladele nu delimiteazã un bloc de instructiuni ci fac parte din declaratia ***struct***.
In structuri diferite pot exista câmpuri cu acelasi nume, dar într-o aceeasi structurã numele de câmpuri trebuie sã fie diferite.
Declararea unor variabile de un tip structurã se poate face fie dupã declararea tipului structurã, fie simultan cu declararea tipului structurã. Exemple:

```
struct time t1,t2, t[100];                  // t este vector de structuri
struct complex {float re,im;} c1,c2,c3;     // numere complexe
struct complex cv[200];                     // un vector de numere complexe
```

Printr-o declaratie ***struct*** se defineste un nou tip de date de cãtre utilizator.
Utilizarea tipurilor structurã pare diferitã de utilizarea tipurilor predefinite, prin existenta a douã cuvinte care desemneazã tipul (***struct*** numestr).

Declaratia *typedef* din C permite atribuirea unui nume oricãrui tip, nume care se poate folosi apoi la fel cu numele tipurilor predefinite ale limbajului. Sintaxa declaratiei *typedef* este la fel cu sintaxa unei declaratii de variabilã, dar se declarã un nume de tip si nu un nume de variabilã.

In limbajul C declaratia typedef se utilizeazã frecvent pentru atribuirea de nume unor tipuri structurã. Exemple:

```
  // definire nume tip simultan cu definire tip structurã
typedef struct { float re,im;} complex;
  // definire nume tip dupã definire tip structura
typedef struct activ act;
```

Deoarece un tip structurã este folosit în mai multe functii (inclusiv "main"), definirea tipului structurã (cu sau fãrã *typedef*) se face la începutul fisierului sursã care contine functiile (înaintea primei functii). Dacã un program este format din mai multe fisiere sursã atunci definitia structurii face parte dintr-un fisier antet (de tip .H), inclus în fisierele sursã care se referã la acea structurã.

Utilizarea unor nume de structuri permite utilizatorilor extinderea limbajului cu noi tipuri de date, mai adecvate problemei rezolvate. Exemplu:

```
  // definitii de tipuri
typedef struct { float x,y;} punct;
typedef struct { int nv; punct v[50];} poligon;
  // lungime segment delimitat de doua puncte
float lung (punct a, punct b) {
 float dx= b.x-a.x;
 float dy= b.y-a.y;
 return sqrt ( dx*dx+dy*dy);
}
  // calcul primetru poligon
float perim ( poligon p) {
 int i,n; float rez=0;
 n=p.nv;
 for (i=0;i<n-1;i++)
   rez = rez + lung (p.v[i],p.v[i+1]);
 return rez+lung(p.v[n-1],p.v[0]);
}
```

Se pot folosi ambele nume ale unui tip structurã (cel precedat de *struct* si cel dat prin *typedef*), care pot fi chiar identice. Exemplu:

```
typedef struct complex  {float re; float im;} complex;
typedef struct point { double x,y;} point;
struct point p[100];
 // calcul arie triunghi dat prin coordonatele varfurilor
double arietr ( point a, point b, point c) {
  return  a.x * (b.y-c.y) - b.x * (a.y-c.y) + c.x * (a.y-b.y);
```

```
}
```

Atunci când numele unui tip structurã este folosit frecvent, inclusiv în argumente de functii, este preferabil un nume introdus prin *typedef*, dar dacã vrem sã punem în evidentã cã este vorba de tipuri structurã vom folosi numele precedat de cuvântul cheie *struct*.

Utilizarea tipurilor structurã

Un tip structurã poate fi folosit în :
- declararea de variabile structuri sau pointeri la structuri :
- declararea unor argumente formale de functii (structuri sau pointeri la structuri)
- declararea unor functii cu rezultat de un tip structurã.

Operatiile posibile cu variabile de un tip structurã sunt:
- atribuirea între variabile de acelasi tip structurã.
- transmiterea ca argument efectiv la apelarea unei functii.
- transmiterea ca rezultat al unei functii, într-o instructiune *return*.

Nu existã constante de tip structurã, dar este posibilã initializarea la declarare a unor variabile structurã. Exemplu:

```
struct complex c1= {1,-1}, c2= {2,3};
```

Exemplu de utilizare a unei structuri initializate într-o functie:

```
  // ridicare numar complex la o putere intreaga prin inmultiri repetate
void put_cx (complex a, int n, complex * pc) {
  complex c={1,0}; int k;
  for (k=0;k<n;k++)
    prod_cx (a,c, &c);
  *pc=c;
}
```

Astfel de variabile initializate si cu atributul *const* ar putea fi folosite drept constante simbolice. Exemplu:

```
const complex unu={1,0};
void put_cx (complex a, int n, complex * pc) {
  complex c=unu; int k;
  for (k=0;k<n;k++)
    prod_cx (a,c, &c);
  *pc=c;
}
```

Alti operatori ai limbajului, în afarã de atribuire, nu se pot folosi cu operanzi de un tip structurã si trebuie definite functii pentru operatii cu structuri: comparatii, operatii

aritmetice, operatii de citire-scriere etc. Exemplul urmãtor aratã cum se poate ordona un vector de structuri “time”, tip definit anterior:

```
  // scrie ora,min,sec
void wrtime ( struct time t) {
  printf ("%02d:%02d:%02d \n", t.ora,t.min,t.sec);
}
  // compara momente de timp
int cmptime (struct time t1, struct time t2) {
  int d;
  d=t1.ora - t2.ora;
  if (d) return d;
  d=t1.min - t2.min;        // <0 daca t1<t2 si >0 daca t1>t2
  if (d) return d;          // rezultat negativ sau pozitiv
  return t1.sec - t2.sec;        // rezultat <0 sau =0 sau > 0
}
   // utilizare functii
int main () {
  struct time tab[200], aux; int i, j, n;
  . . .  // citire date
  // ordonare vector
 for (j=1;j<n;j++)
    for (i=1;i<n;i++)
       if ( cmptime (tab[i-1],tab[i]) > 0) {
          aux=tab[i-1]; tab[i-1]=tab[i]; tab[i]=aux;
       }
   // afisare vector ordonat
 for (i=0;i<n;i++)
    wrtime(tab[i]);
}
```

Câmpurile unei variabile structurã nu se pot folosi decât dacã numele câmpului este precedat de numele variabilei structurã din care face parte, deoarece existã un câmp cu acelasi nume în toate variabilele de un acelasi tip structurã. Exemplu:

```
int main () {
   complex c1,c2;
   scanf (“%f%f", &c1.re, &c1.im);      // citire c1
   c2.re= c1.re; c2.im= -c1.im;      // complex conjugat
   printf (“(%f,%f) “, c2.re, c2.im);      // scrie c2
}
```

Dacã un câmp este la rândul lui o structurã, atunci numele unui câmp poate contine mai multe puncte ce separã numele variabilei si câmpurilor de care apartine (în ordine ierarhicã). Exemplu:

```
struct activ a;
printf (“%s începe la %d: %d si se termina la %d: %d \n”,
    a.numeact, a.start.ora, a.start.min, a.stop.ora, a.stop.min);
```

Principalele avantaje ale utilizării unor tipuri structură sunt:
- Programele devin mai explicite dacă se folosesc structuri în locul unor variabile separate.
- Se pot defini tipuri de date specifice aplicatiei iar programul reflectă mai bine universul aplicatiei.
- Se poate reduce numărul de argumente al unor functii prin gruparea lor în argumente de tipuri structură si deci se simplifică utilizarea acelor functii.
- Se pot utiliza structuri de date extensibile, formate din variabile structură alocate dinamic si legate între ele prin pointeri (liste înlănţuite, arbori s.a).

Functii cu argumente si rezultat structură

Operatiile cu variabile structură se realizează prin functii, definite de utilizator. Exemplu de functie pentru afisarea unui număr complex:

```
void writex ( complex c) {
   printf ("(%.2f,%.2f) ", c.re, c.im);
}
```

O functie care produce un rezultat de un tip structură poate fi scrisă în două moduri, care implică si utilizări diferite ale functiei.

In exemplul următor functia are rezultat de tip structură:

```
   // citire numar complex (varianta 1)
complex readx () {
  complex c;
  scanf ("%f%f",&c.re, &c.im);
  return c;
}
. . . // utilizare
complex a[100]; . . .
for (i=0;i<n;i++)
  a[i]=readx();
```

In exemplul următor functia este de tip *void* si depune rezultatul la adresa primită ca argument (pointer la tip structură):

```
   // citire numar complex (varianta 2)
void readx ( complex * px) {                  // px = pointer la o structură complex
    scanf ("%f%f", &px→re, &px→im);
}
. . . // utilizare
complex a[100]; . . .
for (i=0;i<n;i++)
  readx (&a[i]);                 // adresa variabilei structură a[i]
```

Notatia px→re este echivalentã cu notatia (*px).re si se interpretează astfel: “câmpul “re” al structurii de la adresa px“.

Uneori mai multe variabile descriu împreunã un anumit obiect de date si trebuie transmise la functiile ce lucrează cu obiecte de tipul respectiv. Gruparea acestor variabile într-o structurã va reduce numãrul de argumente si va simplifica apelarea functiilor. Exemple de obiecte definite prin mai multe variabile: obiecte geometrice (puncte, poligoane s.a), date calendaristice si momente de timp, structuri de date (stiva, coada, s.a), vectori, matrice, etc.

Exemplu de grupare într-o structurã a adresei si dimensiunii unui vector:

```
typedef struct {
   int vec[1000];
   int dim;
} vector;
   // afisare vector
void scrvec (vector v) {
  int i;
  for (i=0;i<v.dim;i++)
       printf ("%d  ",v.vec[i]);
  printf ("\n");
 }
    // extrage elemente comune din doi vectori
vector comun (vector a, vector b) {
  vector c; int i,j,k=0;
  for (i=0;i<a.dim;i++)
   for (j=0;j<b.dim;j++)
     if (a.vec[i]==b.vec[j])
       c.vec[k++]=a.vec[i];
  c.dim=k;
  return c;
}
```

De remarcat cã o functie nu poate avea ca rezultat direct un vector, dar poate avea ca rezultat o structurã care include un vector.

Pentru structurile care ocupã un numãr mare de octeti este mai eficient sã se transmitã ca argument la functii adresa structurii (un pointer) în loc sã se copieze continutul structurii la fiecare apel de functie si sã se ocupe loc în stiva de variabile *auto*, chiar dacã functia nu face nici o modificare în structura a cãrei adresã o primeste. De exemplu functia de bibliotecã “asctime” primeste adresa unei structuri, al cãrui continut îl trasnformã într-un sir de caractere ASCII:

```
char *asctime(const struct tm *tp) {                  // din lcc-win32
  static char wday[7][3] = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};
   static const char mon[12][3] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
     "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
   static char result [26];
```

```
   sprintf (result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",wday[tp→tm_wday],
      mon[tp→tm_mon], tp→tm_mday, tp→tm_hour,  tp→tm_min, tp→tm_sec,
   1900 + tp→tm_year);
   return result;
}
```

Pe de altã parte, functiile cu argumente pointeri la structuri, ca si functiile cu argumente de orice tip pointer, pot produce efecte secundare (laterale) nedorite, prin modificarea involuntarã a unor variabile din alte functii (pentru care s-a primit adresa).

Structuri cu continut variabil

Cuvântul cheie ***union*** se foloseste la fel cu ***struct***, dar defineste un grup de variabile care nu se memoreazã simultan ci alternativ. In felul acesta se pot memora diverse tipuri de date la o aceeasi adresã de memorie. Alocarea de memorie se face (de cãtre compilator) în functie de variabila ce necesitã maxim de memorie. O uniune face parte de obicei dintr-o structurã care mai contine si un câmp discriminant, care specificã tipul datelor memorate (alternativa selectatã la un moment dat). Exemplul urmãtor aratã cum se poate lucra cu numere de diferite tipuri si lungimi, reunite într-un tip generic :

```
   // numar de orice tip
struct numar {
  char tipn;       // tip numar (un caracter)
  union {
   int ival;  long lval;  float fval;  double dval;
  } v;
};
  // afisare numar
void write (struct numar n) {
  switch (n.tipn) {
   case 'i': printf ("%d ",n.v.ival);break;
   case 'l': printf ("%ld ",n.v.lval);break;
   case 'f': printf ("%f ",n.v.fval);break;
   case 'd': printf ("%.15lf ",n.v.dval);break;
  }
}
  int main () {
      struct numar a,b,c,d;
      a = read('i'); b=read('l');
      c = read('f'); d=read('d');
      write(a); write(b); write(c); write(d);
  }
```

Pentru câmpul discriminant se poate defini un tip prin enumerare, împreunã cu valorile constante (simbolice) pe care le poate avea. Exemplu:

```
enum tnum {I,L,F,D} ;          // definire tip “tnum”
void write (struct numar n) {
  switch (n.tipn) {
   case I : printf ("%d ",n.v.ival); break;       // int
   case L: printf ("%ld ",n.v.lval);break;        // long
   case F: printf ("%f ",n.v.fval);break;         // float
   case D: printf ("%.15lf ",n.v.dval);break;  // double
  }
}
struct numar read (tnum tip) {
  struct numar n;
  n.tipn=tip;
  switch (tip) {
   case I: scanf ("%d", &n.v.ival);break;
   . . .
}
  return n;
}
int main () {
     struct numar a,b,c,d;
     a = read(I); write(a);
}
```

In locul constructiei ***union*** se poate folosi o variabilã de tip ***void**** care va contine adresa unui numãr, indiferent de tipul lui. Memoria pentru numãr se va aloca dinamic. Exemplu:

```
enum tnum {I,L,F,D} ;
struct number {
  tnum tipn; // tip numar
  void * pv;  // adresa numar
};
  // afisare numar
void write (number n) {
  switch (n.tipn) {
   case I: printf ("%d ", *(int*) n.pv);break;
   ...
   case D: printf ("%.15lf ",*(double*) n.pv);break;
  }
}
```

Structuri predefinite

Anumite functii de bibliotecã folosesc tipuri structurã definite în fisiere de tip H. Câmpurile unor structuri predefinite nu interesează pe utilizatori; un exemplu este tipul FILE definit în "stdio.h" si a cãrui definitie depinde de implementare si de sistemul de operare gazdã si care este folositã numai sub forma FILE*.

Un exemplu de structurã a cãrei definitie trebuie cunoscutã este “struct tm” definitã în <time.h> cu componente ce definesc complet un moment de timp:

```
struct tm {
  int   tm_sec, tm_min, tm_hour;      // secunda, minut, ora
  int   tm_mday,  tm_mon,  tm_year;     // zi, luna, an
  int   tm_wday, tm_yday;                  // numar zi in saptamana si in an
  int   tm_isdst;                             // 1=se modifica ora (iarna/vara), 0= nu
};
```

Exemplul urmãtor aratã cum se poate afisa ora si ziua curentã, folosind numai functii standard:

```
#include <stdio.h>
#include <time.h>
int main(void) {
  time_t t;                    // time_t  este alt nume pentru  long
  struct tm *area;             // pentru rezultat functie localtime
  t = time(NULL);              // obtine ora curenta
  area = localtime(&t);        // conversie din time_t in struct tm
  printf("Local time is: %s", asctime(area));
}
```

Functia “time” transmite rezultatul si prin numele functiei si prin argument:

```
long time (long*);
```

deci se putea apela si astfel:

```
time (&t);
```

Structura “struct stat” este definitã în fisierul <sys/stat.h> si reuneste date despre un fisier, cu exceptia numelui. Primii 14 octeti contin informatii valabile numai pentru sisteme de tip Unix si sunt grupate într-un singur câmp în definitia urmãtoare:

```
struct stat {
  short unix [7];                  // fara semnificatie in sisteme Windows
  long st_size;                    // dimensiune fisier (octeti)
  long st_atime, st_mtime;         // ultimul acces / ultima modificare
  long st_ctime;                       // data de creare
};
```

Functia “stat” completeazã o astfel de structurã pentru un fisier cu nume dat:

```
int stat (char* filename, struct stat * p);
```

Pentru a afla dimensiunea unui fisier normal (care nu este fisier director) vom putea folosi functia urmãtoare:

```
long filesize (char * filename) {
```

```
   struct stat fileattr;
   if (stat (filename, &fileattr) < 0)      // daca fisier negasit
      return -1;
   else                                     // fisier gasit
      return (fileattr.st_size);            // campul st_size contine lungimea
}
```

8. Tipuri pointer în C

Variabile pointer

O variabilã pointer poate avea ca valori adrese de memorie. Aceste adrese pot fi:
- Adresa unei valori de un anumit tip (pointer la date)
- Adresa unei functii (pointer la o functie)
- Adresa unei zone cu continut necunoscut (pointer la *void*).

Cel mai frecvent se folosesc pointeri la date.

Existã o singurã constantã de tip pointer, cu numele NULL (valoare zero) si care este compatibilã la atribuire si comparare cu orice tip pointer. Totusi, se poate atribui o constantã întreagã convertitã la un tip pointer unei variabile pointer. Exemplu:

```
char * p = (char*)10000;          // o adresa de memorie
```

Desi adresele de memorie sunt de multe ori numere întregi pozitive, tipurile pointer sunt diferite de tipurile întregi si au utilizãri diferite.

In limbajul C tipurile pointer se folosesc în principal pentru:
- Declararea si utilizarea de vectori, mai ales pentru vectori ce contin siruri de caractere.
- Argumente de functii prin care se transmit rezultate (adresele unor variabile din afara functiei).
- Acces la date alocate dinamic si care nu pot fi adresate printr-un nume.
- Argumente de functii prin care se transmite adresa unei alte functii.

Declararea unei variabile (sau argument formal) de un tip pointer include declararea tipului datelor (sau functiei) la care se referã acel pointer. Sintaxa declarãrii unui pointer la o valoare de tipul “tip” este

```
tip * ptr;      // sau  tip* ptr;  sau  tip  *ptr;
```

Exemple de variabile si argumente pointer:

```
char * pc;              // pc= adresa unui caracter sau sir de car.
int * pi;               // pi= adresa unui intreg sau vector de int
void * p;               // p= adresa de memorie
int * * pp;             // pp= adresa unui pointer la un intreg
int strlen (char* str); // str=adr. unui sir de caractere
```

Atunci când se declarã mai multe variabile pointer de acelasi tip, nu trebuie omis asteriscul care aratã ca este un pointer. Exemple:

```
int *p, m;     // m de tip "int", p de tip "int*"
int *a, *b ;       // a si b de tip pointer
```

Dacã se declarã un tip pointer cu *typedef* atunci se poate scrie astfel:

```
typedef int* intptr;        // intptr este nume de tip
intptr p1,p2,p3;            // p1, p2, p3 sunt pointeri
```

Tipul unei variabile pointer este important pentru cã determinã câti octeti vor fi folositi de la adresa continutã în variabila pointer si cum vor fi interpretati. Un pointer la *void* nu poate fi utilizat direct, deoarece nu se stie câti octeti trebuie folositi si cum.

Operatii cu pointeri la date

Operatiile posibile cu variabile pointer pot fi rezumate astfel:
- Indirectarea printr-un pointer (diferit de *void* *), pentru acces la datele adresate de acel pointer: operatorul unar '*'. Exemple:

```
*p = y;
x = *p;
*s1++ = *s2++;
```

- Atribuire la un pointer. In partea dreaptã poate fi un pointer de acelasi tip (eventual cu conversie de tip) sau constanta NULL sau o expresie cu rezultat pointer. Exemple:

```
p1=p1;
p=NULL;
p=&x;
p=*pp; p =(int*)malloc(n);
```

Operatorul unar '&' aplicat unei variabile are ca rezultat adresa variabilei respective (deci un pointer). Functia "malloc" si alte functii au ca rezultat un pointer de tip *void**.

Unei variabile de tip ***void**** i se poate atribui orice alt tip de pointer fãrã conversie de tip explicitã si un argument formal de tip ***void**** poate fi înlocuit cu un argument efectiv de orice tip pointer.

Atribuirea între alte tipuri pointer se poate face numai cu conversie de tip explicitã ("cast") si permite interpretarea diferitã a unor date din memorie. De exemplu, putem extrage cei doi octeti dintr-un întreg scurt astfel:

```
short int n; char c1, c2;
c1= *(char*)&n;
c2= *(char*)(&n+1);
```
sau:
```
char * p = (char*) &n;
c1= *p;  c2 = *(p+1);
```

- Compararea sau scãderea a douã variabile pointer de acelasi tip (de obicei adrese de elemente dintr-un acelasi vector). Exemplu:

```
  // pozitia (indicele) in sirul s1 a sirului s2
  // sau un numar negativ daca s1 nu contine pe s2
int pos ( char* s1, char * s2) {
  char * p1 =strstr(s1,s2);     // adresa lui s2 in s1
  if (p1)
     return p1-s1;
  else
     return -1;
}
```

- Adunarea sau scãderea unui întreg la (din) un pointer, incrementarea si decrementarea unui pointer. Exemplu:

```
  // afisarea unui vector
 void printVector ( int a [], int n) {
    while (n--)
       printf ("%d ", *a++);
 }
```

Trebuie observat cã incrementarea unui pointer si adunarea unui întreg la un pointer nu adunã întotdeauna întregul 1 la adresa continutã în pointer; valoarea adaugatã (scãzutã) depinde de tipul variabilei pointer si este egalã cu produsul dintre constantã si numãrul de octeti ocupat de tipul adresat de pointer. Expresiile urmãtoare sunt echivalente:

```
  p = p+ c;    p = p+c*sizeof(tip);          // tip * p ;
  ++p;  p=p+sizeof(tip);                     // tip * p ;
```

Aceastã conventie permite referirea simplã la elemente succesive dintr-un vector folosind indirectarea printr-o variabilã pointer.

Operatorul unar ***sizeof*** aplicat unui nume de tip sau unei variabile are ca rezultat numãrul de octeti alocati pentru tipul sau pentru variabila respectivã:

```
char c; float f;
sizeof(char)= sizeof c = 1
sizeof(float) = sizeof f = 4
```

Operatorul ***sizeof*** permite scrierea unor programe portabile, care nu depind de lungimea pe care se reprezintã în memorie fiecare tip de date. De exemplu, tipul ***int*** ocupã uneori 2 octeti iar alteori 4 octeti.

O eroare frecventã este utilizarea unei variabile pointer care nu a primit o valoare (adicã o adresã de memorie) prin atribuire sau prin initializare la declarare. Efectul este accesul la o adresã de memorie imprevizibilã, chiar în afara spatiului de memorie ocupat de programul ce contine eroarea. Exemplu:

```
int * a;              // declarata dar neinitializata
scanf ("%d",a) ;      // citeste la adresa continuta in variabila a
```

Vectori si pointeri

O variabilã vector contine adresa de început a vectorului (adresa primei componente din vector) si de aceea este echivalentã cu un pointer la tipul elementelor din vector. Aceasta echivalentã este utilizatã de obicei în argumentele de tip vector si în lucrul cu vectori alocati dinamic.

O functie poate avea ca rezultat un pointer dar nu si rezultat vector.

Pentru declararea unei functii care primeste un vector de întregi si dimensiunea lui avem cel putin douã posibilitãti:

```
void printVec (int a [ ], int n);
void printVec (int * a, int n);
```

In interiorul functiei ne putem referi la elementele vectorului "a" fie prin indici, fie prin indirectare, indiferent de felul cum a fost declarat parametrul vector "a". Exemplu:

```
  // prin indexare
void printVec (int a[ ], int n) {
  int i;
  for (i=0;i<n;i++)
    printf (%6d",a[i]);
}
  // prin indirectare
void printVec (int *a, int n) {
  int i;
  for (i=0;i<n;i++)
    printf (%6d", *a++);
}
```

Citirea elementelor unui vector se poate face asemãnãtor:

```
for (i=0;i<n;i++)
  scanf ("%d", a+i);      // echivalent cu &a[i] si cu a++
```

In general, existã urmãtoarele echivalente de notatie pentru un vector "a":

```
a[0]   *a              &a[0] a
a[1]   *(a+1)          &a[1] a+1
a[k]   *(a+k)              &a[k] a+k
```

Aritmetica cu pointeri este diferitã de aritmetica cu numere întregi.

In aplicatiile numerice se preferã argumentele de tip vector si adresarea cu indici, iar în functiile cu siruri de caractere se preferã argumente de tip pointer si adresarea indirectã prin pointeri.

Diferenta majorã dintre o variabilã pointer si un nume de vector este aceea cã un nume de vector este un pointer constant (adresa este alocatã de compilatorul C si nu mai poate fi modificatã la executie) Un nume de vector nu poate apare în stânga unei atribuiri, în timp ce o variabilã pointer are un continut modificabil prin atribuire sau prin operatii aritmetice. Exemple:

```
int a[100], *p;
p=a; ++p;          // corect
a=p; ++a;          // ambele instructiuni produc erori
```

Când un nume de vector este folosit ca argument atunci se transmite un pointer cu aceeasi valoare ca numele vectorului, iar functia poate folosi argumentul formal în stânga unei atribuiri. Exemplu:

```
 // copiere sir terminat cu zero de la s la d
void stcpy ( char d[], char s[]) {
  while (*d++=*s++) ;          // copiaza caractere pana la un octet zero
  *d=0;                        // adauga terminator de sir
}
int main () {
 char a[100],b[100];
 while (scanf("%s",b)==1) {
   stcpy(a,b);  puts(a);
 }
}
```

Declararea unui vector (alocat la compilare) nu este echivalentã cu declararea unui pointer, deoarece o declaratie de vector alocã memorie si initializeaza pointerul ce reprezintã numele vectorului cu adresa zonei alocate (operatii care nu au loc automat la declararea unui pointer).

```
int * a;  a[0]=1;      // gresit !
int *a={3,4,5};        // echivalent cu: int a[]={3,4,5}
```

Nu se poate declara un vector cu componente de tip *void*. Exemple:

```
void a[100];     // incorect
void * a;        // corect
```

Operatorul ***sizeof*** aplicat unui nume de vector cu dimensiune fixã are ca rezultat numãrul total de octeti ocupati de vector, dar aplicat unui argument formal de tip vector (sau unui pointer la un vector alocat dinamic) are ca rezultat mãrimea unui pointer:

```
float x[10], * y=(float*)malloc (10*sizeof(float));
printf ("%d,%d \n",sizeof(x), sizeof(y));          // scrie 40, 4
```

Numãrul de elemente dintr-un vector alocat la compilare sau initializat cu un sir de valori se poate afla prin expresia: sizeof (x) / sizeof(x[0])

Pointeri în functii

In definirea functiilor se folosesc pointeri pentru:
- Transmiterea de rezultate prin argumente;
- Transmiterea unei adrese prin rezultatul functiei;

O functie care trebuie sã modifice mai multe valori primite prin argumente sau care trebuie sã transmitã mai multe rezultate calculate de functie trebuie sã foloseascã argumente de tip pointer.

O functie care primeste un numãr si trebuie sã modifice acel numãr poate transmite prin rezultatul ei (prin *return*) valoarea modificatã. Exemplu:

```
// functie care incrementeaza un intreg n modulo m
  int incmod (int n, int m ) {
    return ++n % m;
  }
```

O functie care primeste douã sau mai multe numere pe care trebuie sã le modifice va avea argumente de tip pointer sau un argument vector care reuneste toate rezultatele (datele modificate). Exemplu:

```
 // calculeaza urmatorul moment de timp (ora,min,sec)
void inctime (int*h,int*m,int*s) {
 *s=incmod(*s,60);          // secunde
 if (*s==0) {
   *m=incmod(*m,60);        // minute
   if (*m==0)
     *h=incmod(*h,24);      // ore
 }
}
  // utilizare functie
int main () {
 int h,m,s;
 while ( scanf ("%d%d%d",&h,&m,&s) >0) {
   inctime (&h,&m,&s);
   printf ("%4d%4d%4d \n",h,m,s);
 }
}
```

In exemplul anterior cele trei argumente întregi pot fi reunite într-un vector, pentru simplificarea functiei:

```
void inctime (int t[3]) {        // t[0]=h, t[1]=m, t[2]=s
  t[2]=incmod (t[2],60);         // secunde
  if (t[2]==0) {
    t[1]=incmod (t[1],60);       // minute
    if (t[1]==0)
      t[0]=incmod (t[0],24);         // ore
  }
}
```

O functie poate avea ca rezultat un pointer, dar acest pointer nu trebuie sã continã adresa unei variabile locale. De obicei, rezultatul pointer este egal cu unul din argumente, eventual modificat în functie. Exemplu:

```
      // incrementare pointer p
    char * incptr ( char * p) {
       return ++p;
    }
```

O variabilã localã are o existentã temporarã, garantatã numai pe durata executãrii functiei în care este definitã (cu exceptia variabilelor locale statice) si de aceea adresa unei astfel de variabile nu trebuie transmisã în afara functiei, pentru a fi folositã ulterior. Exemplu gresit:

```
  // vector cu cifrele unui nr intreg
int * cifre (int n) {
  int k, c[5];       // vector local
  for (k=4;k>=0;k--) {
    c[k]=n%10; n=n/10;
  }
  return c;          // aici este eroarea !
}
```

Anumite functii cu mai multe rezultate si argumente de tip pointer pot fi înlocuite prin mai multe functii, fiecare cu un singur rezultat. De exemplu, în locul functiei urmãtoare vom scrie functii separate pentru minim si maxim:

```
void minmax (int a[ ], int n, int * min, int* max) {
  int i;
  *min=INT_MAX; *max = INT_MIN;
  for (i=0;i<n;i++){
    if (*min > a[i])
      *min=a[i];
    if (*max < a[i])
      *max=a[i];
```

```
  }
}
```

O functie care trebuie sã transmitã ca rezultat un vector poate fi scrisã corect în douã feluri:

- Primeste ca argument adresa vectorului (definit si alocat în altã functie) si depune rezultatele la adresa primitã (este solutia recomandatã). Exemplu:

```
void cifre (int n, int c[ ]) {
  int k;
  for (k=4;k>=0;k--) {
    c[k]=n%10; n=n/10;
  }
}
```

- Alocã dinamic memoria pentru vector (cu "malloc"), iar aceastã alocare se mentine si la iesirea din functie. O solutie oarecum echivalentã este utilizarea unui vector local static, care continuã sã existe dupã terminarea functiei. Functia are ca rezultat adresa vectorului alocat în cadrul functiei.Problema este unde si când se eliberează memoria alocatã. Exemplu:

```
int * cifre (int n) {
  int k, *c;                    // vector local
  c = (int*) malloc (5*sizeof(int));
  for (k=4;k>=0;k--) {
    c[k]=n%10; n=n/10;
  }
  return c;                     // corect
}
```

Pointeri la functii

Anumite aplicatii numerice necesitã scrierea unei functii care sã poatã apela o functie cu nume necunoscut, dar cu prototip si efect cunoscut. De exemplu, o functie care sã calculeze integrala definitã a oricãrei functii cu un singur argument sau care sã determine o rãdãcinã realã a oricãrei ecuatii (neliniare). Aici vom lua ca exemplu o functie "listf" care poate afisa (lista) valorile unei alte functii cu un singur argument, într-un interval dat si cu un pas dat. Exemple de utilizare a functiei "listf" pentru afisarea valorilor unor functii de bibliotecã:

```
main () {
  listf (sin,0.,2.*M_PI, M_PI/10.);
  listf (exp,1.,20.,1.);
}
```

Problemele apar la definirea unei astfel de functii, care primeste ca argument numele (adresa) unei functii.

Prin conventie, în limbajul C, numele unei functii neînsotit de o listã de argumente (chiar vidã) este interpretat ca un pointer cãtre functia respectivã (fãrã a se folosi operatorul de adresare '&'). Deci "sin" este adresa functiei "sin(x)" în apelul functiei "listf". O eroare de programare care trece de compilare si se manifestã la executie este apelarea unei functii fãrã paranteze; compilatorul nu apeleazã functia si considerã cã programatorul vrea sã foloseascã adresa functiei. Exemplu:

```
if ( kbhit ) break;      // gresit, echiv. cu if(1) break;
if ( kbhit() ) break;    // iesire din ciclu la orice tasta
```

Declararea unui argument formal (sau unei variabile) de tip pointer la o functie are forma urmãtoare:

```
tip (*pf) (lista_arg_formale)
```

unde:

pf este numele argumentului (variabilei) pointer la functie

tip este tipul rezultatului functiei

Parantezele sunt importante, deoarece absenta lor modifica interpretarea declaratiei. Exemplu de declaratie functie cu rezultat pointer:

```
tip * f (lista_arg_formale)
```

In concluzie, definirea functiei "listf" este:

```
void listf (double (*fp)(double), double min, double max, double pas) {
  double x,y;
  for (x=min; x<=max; x=x+pas) {
    y=(*fp)(x);                          // sau:  y=fp(x);
    printf ("\n%20.10lf %20.10lf", x,y);
  }
}
```

Pentru a face programele mai explicite se pot defini nume de tipuri pentru tipuri pointeri la functii, folosind declaratia *typedef*. Exemplu:

```
typedef double (* ftype) (double);
void listf(ftype fp,double min,double max, double pas) {
  double x,y;
  for (x=min; x<=max; x=x+pas) {
    y=fp(x);
    printf ("\n%20.10lf %20.10lf", x,y);
  }
}
```

O functie B transmisã, printr-un pointer, ca argument unei alte functii F se numeste si functie “callback”, pentru cã ea va fi apelatã “înapoi” de functia F. De obicei, functia F este o functie de bibliotecã, iar functia B este parte din aplicatie. Functia F poate apela o diversitate de functii, dar toate cu acelasi prototip, al functiei B.

In fisierul “stdlib.h” sunt declarate patru functii generice pentru sortarea, cãutarea liniarã si cãutarea binarã într-un vector cu componente de orice tip, care ilustreazã o modalitate simplã de generalizare a tipului unui vector. Argumentul formal de tip vector al acestor functii este declarat ca *void** si este înlocuit cu un argument efectiv pointer la un tip precizat (nume de vector).

Un alt argument al acestor functii este adresa unei functii de comparare a unor date de tipul celor memorate în vector, functie furnizatã de utilizator si care depinde de datele folosite în aplicatia sa.

Pentru exemplificare urmeazã declaratiile pentru trei din aceste functii (“lfind” este la fel cu “lsearch”):

```
void *bsearch (const void *key, const void *base, size_t nelem, size_t width,
          int (*fcmp)(const void*, const void*));
void *lsearch (const void *key, void *base, size_t * pnelem, size_t width,
          int (*fcmp)(const void *, const void *));
void qsort(void *base, size_t nelem, size_t width,
           int (*fcmp)(const void *, const void *));
```

“base” este adresa vectorului, “key” este cheia (valoarea) cãutatã în vector (de acelasi tip cu elementele din vector), “width” este dimensiunea unui element din vector (ca numãr de octeti), “nelem” este numarul de elemente din vector, “fcmp” este adresa functiei de comparare a douã elemente din vector.

Exemplul urmãtor aratã cum se poate ordona un vector de numere întregi cu functia “qsort” :

```
  // comparare numere intregi
int intcmp (const void * a, const void * b) {
  return *(int*)a-*(int*)b;
}
int main () {
   int a[]= {5,2,9,7,1,6,3,8,4};
   int i, n=9;                          // n=dimensiune vector
   qsort ( a,9, sizeof(int),intcmp);   // ordonare vector
   for (i=0;i<n;i++)                    // afisare rezultat
     printf("%d ",a[i]);
}
```

Un vector de pointeri la functii poate fi folosit în locul unui bloc *switch* pentru selectarea unei functii dintr-un grup de mai multe functii, într-un program cu meniu de optiuni prin care operatorul alege una din functiile realizate de programul respectiv. Exemplu:

```
  // functii ptr. operatii realizate de program
void  unu () {
  printf ("unu\n");
}
void  doi () {
  printf ("doi\n");
}
void  trei () {
  printf ("trei\n");
}
  // selectare si apel functie
typedef void (*funPtr) ();
int main () {
  funPtr tp[ ]= {unu,doi,trei};          // vector de pointeri la functii
  short option=0;
  do {
     printf(“Optiune (1/2/3):“);
     scanf ("%hd", &option);
     if (option >=1 && option <=3)
       tp[option-1]();                   // apel functie (unu /doi / trei)
  } while (1);
}
```

Secventa echivalentã cu *switch* este :

```
do {
   printf(“Optiune (1/2/3):“);
   scanf ("%hd", &option);
   switch (option) {
     case 1: unu(); break;
     case 2: doi(); break;
     case 3: trei(); break;
     default: continue;
   }
} while (1);
```

9. Alocarea dinamicã a memoriei în C

Clase de memorare în C

Clasa de memorare aratã când, cum si unde se alocã memorie pentru o variabilã. Orice variabilã are o clasã de memorare care rezultã fie din declaratia ei, fie implicit din locul unde este definitã variabila.

Existã trei moduri de alocare a memoriei, dar numai douã corespund unor clase de memorare:

- Static: memoria este alocatã la compilare în segmentul de date din cadrul programului si nu se mai poate modifica în cursul executiei. Variabilele externe, definite în afara functiilor, sunt implicit statice, dar pot fi declarate ***static*** si variabile locale, definite în cadrul functiilor.
- Automat: memoria este alocatã automat, la activarea unei functii, în zona stivã alocatã unui program si este eliberatã automat la terminarea functiei. Variabilele locale unui bloc (unei functii) si argumentele formale sunt implicit din clasa ***auto***. Memoria se alocã în stiva atasatã programului.
- Dinamic: memoria se alocã la executie în zona "heap" atasatã programului, dar numai la cererea explicitã a programatorului, prin apelarea unor functii de bibliotecã (malloc, calloc, realloc). Memoria este eliberatã numai la cerere, prin apelarea functiei "free". Variabilele dinamice nu au nume si deci nu se pune problema clasei de memorare (clasa este atribut al variabilelor cu nume).

Variabilele statice pot fi initializate numai cu valori constante (pentru cã are loc la compilare), dar variabilele ***auto*** pot fi initializate cu rezultatul unor expresii (pentru cã initializarea are loc la executie). Exemplu de functie care afiseazã un întreg pozitiv în binar folosind câturile împãrtirii cu puteri descrescãtoare a lui 10:

```
   // afisare intreg in binar
void binar ( int x) {
  int n= digits(x);            // numar de cifre zecimale in x
  int d= pw10 (n-1);           // pw10 = ridicare la puterea 10 , d=10^(n-1)
  while ( x >0) {
     printf("%d",x/d);         // scrie catul impartirii lui x prin d
     x=x%d; d=d/10;            // continua cu x%d si d/10
  }
}
```

Toate variabilele externe (si statice) sunt automat initializate cu valori zero (inclusiv vectorii).

O variabilã staticã declaratã într-o functie îsi pãstreazã valoarea între apeluri succesive ale functiei, spre deosebire de variabilele ***auto*** care sunt realocate pe stivã la fiecare apel al functiei si pornesc de fiecare datã cu valoarea primitã la initializarea lor (sau cu o valoare imprevizibilã, dacã nu sunt initializate). Variabilele locale statice se

folosesc foarte rar în practica programãrii (functia de bibliotecã “strtok” este un exemplu de functie cu o variabilã staticã).

Cantitatea de memorie alocatã pentru variabilele cu nume rezultã din tipul variabilei si din dimensiunea declaratã pentru vectori. Memoria alocatã dinamic este specificatã explicit ca parametru al functiilor de alocare, în numãr de octeti.

O a treia clasã de memorare este clasa “register” pentru variabile cãrora li se alocã registre ale procesorului si nu locatii de memorie, pentru un timp de acces mai bun. Aceastã clasã nu se va folosi deoarece se lasã compilatorului decizia de alocare a registrelor masinii pentru anumite variabile *auto* din functii.

Memoria neocupatã de datele statice si de instructiunile unui program este împãrtitã între stivã si “heap”. Consumul de memorie “stack” (stiva) este mai mare în programele cu functii recursive si numãr mare de apeluri recursive, iar consumul de memorie “heap” este mare în programele cu vectori si matrice alocate (si realocate) dinamic.

De observat cã nu orice vector cu dimensiune constantã este un vector static; un vector definit într-o functie (alta decât “main”) nu este static deoarece nu ocupã memorie pe toatã durata de executie a programului, desi dimensiunea sa este stabilitã la scrierea programului. Un vector definit într-o functie este alocat pe stivã, la activarea functiei, iar memoria ocupatã de vector este eliberatã automat la terminarea functiei.

Functii de alocare si eliberare a memoriei

Aceste functii standard sunt declarate în fisierul <stdlib.h>. Cele trei functii de alocare au ca rezultat adresa zonei de memorie alocate (de tip *void **) si ca argument comun dimensiunea zonei de memorie alocate (de tip "size_t"). Dacã cererea de alocare nu poate fi satisfãcutã, pentru cã nu mai exista un bloc continuu de dimensiunea solicitatã, atunci functiile de alocare au rezultat NULL.

Functiile de alocare au rezultat *void** deoarece functia nu stie tipul datelor ce vor fi memorate la adresa respectivã. La apelarea functiilor de alocare se folosesc:

- Operatorul *sizeof* pentru a determina numãrul de octeti necesar unui tip de date (variabile);
- Operatorul de conversie “cast” pentru adaptarea adresei primite de la functie la tipul datelor memorate la adresa respectiva (conversie necesarã atribuirii între pointeri de tipuri diferite). Exemple:

```
   //aloca memorie pentru 30 de caractere
 char * str = (char*) malloc(30);
   //aloca memorie ptr. n întregi
 int * a = (int *) malloc( n * sizeof(int));
```

Alocarea de memorie pentru un vector si initializarea zonei alocate cu zerouri se poate face si cu functia “calloc”. Exemplu:

```
int * a= (int*) calloc (n, sizeof(int) );
```

Realocarea unui vector care creste (sau scade) fatã de dimensiunea estimatã anterior se poate face cu functia “realloc”, care primeste adresa veche si noua dimensiune si întoarce noua adresã:

```
 // dublare dimensiune curenta a zonei de la adr. a
a = (int *)realloc (a, 2*n* sizeof(int));
```

In exemplul anterior noua adresã este memoratã tot în variabila pointer “a”, înlocuind vechea adresã (care nu mai este necesarã si nu mai trebuie folositã).

Functia “realloc” realizeazã urmãtoarele operatii:

- Alocã o zonã de dimensiunea specificatã ca al doilea argument.
- Copiazã la noua adresã datele de la adresa veche (primul argument).
- Eliberează memoria de la adresa veche.

Exemplu de functie cu efectul functiei “realloc”:

```
char * ralloc (char * p, int size) {      // p = adresa veche
  char *q;                                // q=adresa noua
  if (size==0) {                          // echivalent cu free
     free(p); return NULL;
  }
  q= (char*) malloc(size);                    // aloca memorie
  if (q) {                                // daca alocare reusita
     memcpy(q,p,size);                        // copiere date de la p la q
     free(p);                             // elibereaza adresa p
  }
  return q;                               // q poate fi NULL
}
```

In functia “ralloc” ar trebui ca ultimul argument al functiei “memcpy” sã fie dimensiunea blocului vechi, dar ea nu este disponibilã într-un mod care sã nu depindã de implementare; oricum, la mãrirea blocului continutul zonei alocate în plus nu este precizat, iar la micsorarea blocului se pierd datele din zona la care se renuntã.

Functia “free” are ca argument o adresã (un pointer) si eliberează zona de la adresa respectivã (alocatã prin apelul unei functii “...alloc”). Dimensiunea zonei nu mai trebuie specificatã deoarece este memoratã la începutul zonei alocate (de cãtre functia de alocare). Exemplu:

```
free(a);
```

Eliberarea memoriei prin "free" este inutilã la terminarea unui program, deoarece înainte de încãrcarea si lansarea în executie a unui nou program se eliberează automat toatã memoria "heap".

Uneori mãrimea blocurilor alocate este un multiplu de 8 octeti, dar detaliile gestiunii memoriei pentru alocare dinamicã depind de fiecare implementare. Există

chiar si alte biblioteci de functii pentru alocare/eliberare de memorie, cu anumite avantaje fatã de functiile standard.

Vectori alocati dinamic

Structura de vector are avantajul simplitãtii si economiei de memorie fatã de alte structuri de date folosite pentru memorarea unei colectii de date. Dezavantajul unui vector cu dimensiune fixã (stabilitã la declararea vectorului si care nu mai poate fi modificatã la executie) apare în aplicatiile cu vectori de dimensiuni foarte variabile, în care este dificil de estimat o dimensiune maximã, fãrã a face risipã de memorie.

De cele mai multe ori programele pot afla (din datele citite) dimensiunile vectorilor cu care lucreazã si deci pot face o alocare dinamicã a memoriei pentru acesti vectori. Aceasta este o solutie mai flexibilã, care foloseste mai bine memoria disponibilã si nu impune limitãri arbitrare asupra utilizãrii unor programe. In limbajul C nu existã practic nici o diferentã între utilizarea unui vector cu dimensiune fixã si utilizarea unui vector alocat dinamic, ceea ce încurajeazã si mai mult utilizarea unor vectori cu dimensiune variabilã.

Un vector alocat dinamic se declarã ca variabilã pointer care se initializeazã cu rezultatul functiei de alocare. Tipul variabilei pointer este determinat de tipul componentelor vectorului.

Exemplul urmãtor aratã cum se poate defini si utiliza un vector alocat dinamic:

```
#include <stdlib.h>
#include <stdio.h>
int main() {
  int n,i; int * a;                    // adresa vector alocat dinamic
  printf ("n="); scanf ("%d", &n);     // dimensiune vector
  a=(int *) calloc (n,sizeof(int));    // aloca memorie pentru vector
// sau:   a=(int*) malloc (n*sizeof(int));
 printf ("componente vector: \n");
 for (i=0;i<n;i++)
   scanf ("%d", &a[i]);                    // sau   scanf (“%d”, a+i);
 for (i=0;i<n;i++)                    // afisare vector
   printf ("%d ",a[i]);
}
```

Existã si cazuri în care datele memorate într-un vector rezultã din anumite prelucrãri, iar numãrul lor nu poate fi cunoscut de la începutul executiei. Un exemplu poate fi un vector cu toate numerele prime mai mici ca o valoare datã. In acest caz se poate recurge la o realocare dinamicã a memoriei. In exemplul urmãtor se citeste un numãr necunoscut de valori întregi într-un vector extensibil:

```
#define INCR 100      // cu cat creste vectorul la fiecare realocare
int main() {
 int n,i,m ;
 float x, * v;              // v = adresa vector
```

```
 n=INCR; i=0;
 v = (float *)malloc (n*sizeof(float));   //alocare initiala
 while ( scanf("%f",&x) != EOF) {
    if (++i == n) {              // daca este necesar
      n= n+ INCR;                // creste dimensiune vector
      v=(float *) realloc (vector,n*sizeof(float));
    }
    v[i]=x;                      // memorare in vector
 }
 for (i=0;i<n;i++)               // afisare vector
   printf ("%f ",v[i]);
}
```

Realocarea repetatã de memorie poate conduce la fragmentarea memoriei “heap”, adicã la crearea unor blocuri de memorie libere dar neadiacente si prea mici pentru a mai fi reutilizate ulterior. De aceea se va evita redimensionarea unui vector cu o valoare foarte micã de un numãr mare de ori; o strategie de realocare folositã pentru vectori este dublarea capacitãtii lor anterioare.

Din exemplele anterioare lipseste eliberarea memoriei alocate pentru vectori, dar fiind vorba de un singur vector alocat în functia “main” si necesar pe toatã durata de executie, o eliberare finalã este inutilã. Eliberarea explicitã poate fi necesarã pentru vectori de lucru, alocati dinamic în functii.

Matrice alocate dinamic

Alocarea dinamicã pentru o matrice este importantã deoarece:
- Foloseste economic memoria si evitã alocãri acoperitoare, estimative.
- Permite matrice cu linii de lungimi diferite.
- Reprezintã o solutie bunã la problema argumentelor de functii de tip matrice.

O matrice alocatã dinamic este de fapt un vector de pointeri cãtre fiecare linie din matrice, deci un vector de pointeri la vectori alocati dinamic. Dacã numãrul de linii este cunoscut sau poate fi estimatã valoarea lui maximã, atunci vectorul de pointeri are o dimensiune constantã. Exemplu de declarare matrice de întregi:

```
int * a[M];        // M este o constanta simbolica
```

Dacã nu se poate estima numãrul de linii din matrice atunci si vectorul de pointeri se alocã dinamic, iar declararea matricei se face ca pointer la pointer:

```
int** a;
```

In acest caz se va aloca mai întâi memorie pentru un vector de pointeri (functie de numãrul liniilor) si apoi se va aloca memorie pentru fiecare linie (functie de numãrul coloanelor) cu memorarea adreselor liniilor în vectorul de pointeri. O astfel de matrice se poate folosi la fel ca o matrice declaratã cu dimensiuni constante. Exemplu:

```
int main () {
```

```
    int ** a; int i,j,nl,nc;
    printf (“nr. linii=“); scanf (“%d”,&nl);
    printf (“nr. col. =“); scanf (“%d”,&nc);
  // memorie pentru vectorul de pointeri la linii
    a = (int**) malloc (nl*sizeof(int*));
    for (i=0; i<nl;i++)                          // aloca memorie pentru fiecare linie i
        a[i] = (int*) calloc (nc, sizeof(int));        //o linie
  //  completare elemente matrice
    for (i=0;i<nl;i++)
        for (j=0;j<nc;j++)
            a[i][j]= nc*i+j+1;
 // afisare matrice
   for (i=0;i<nl;i++) {
      for (j=0;j<nc;j++)
          printf (“%d “, a[i][j] );
      printf (“\n”);
   }
```

Notatia a[i][j] este interpretatã astfel pentru o matrice alocatã dinamic:
a[i] contine un pointer (o adresã b)
b[j] sau b+j contine întregul din pozitia “j” a vectorului cu adresa “b”.
Se poate defini si o functie pentru alocarea de memorie la executie pentru o matrice.
Exemplu:

```
   // rezultat adresa matrice sau NULL
 int * * intmat ( int nl, int nc) {
  int i;
  int ** p=(int **) malloc (nl*sizeof (int*));
  if ( p != NULL)
    for (i=0; i<nl ;i++)
      p[i] =(int*) calloc (nc,sizeof (int));
  return p;
}
 // afisare matrice
void printmat (int ** a, int nl, int nc) {
 int i,j;
 for (i=0;i<nl;i++) {
   for (j=0;j<nc;j++)
      printf (“%2d”, a[i][j] );
   printf(“\n”);
 }
}
 // utilizare matrice
int  main () {
int  nl, nc, i,j , ** a;
printf ("nr linii si nr coloane: \n");
scanf ("%d%d", &nl, &nc);
a= intmat(nl,nc);
   // completare matrice
```

```
  for (i=0;i<nl;i++)
    for (j=0;j<nc;j++)
      a[i][j]= nc*i+j+1;
 printmat (a ,nl,nc);
}
```

Functia “printmat” datã anterior nu poate fi folositã pentru afisarea unei matrice cu dimensiuni constante. Exemplul urmãtor este corect sintactic dar nu se executã corect:

```
 int main () {
   int x [2][2]={{1,2},{3,4}};       // 2 linii si 2 coloane
   printmat ( (int**)x, 2, 2);
 }
```

Explicatia este interpretarea diferitã a continutului zonei de la adresa aflatã în primul argument.

Structuri alocate dinamic

In cazul variabilelor structurã alocate dinamic si care nu au nume se va face o indirectare printr-un pointer pentru a ajunge la variabila structurã.

Avem de ales între urmãtoarele douã notatii echivalente:

pt→ora (*pt).ora

unde “pt” este o variabilã care contine un pointer la o structurã cu câmpul “ora”.

O colectie de variabile structurã alocate dinamic se poate memora în douã moduri:

- Ca un vector de pointeri la variabilele structurã alocatre dinamic;
- Ca o listã înlãntuitã de variabile structurã, în care fiecare element al listei contine si un câmp de legãturã cãtre elementul urmãtor (ca pointer la structurã).

O listã înlantuitã (“linked list”) este o colectie de variabile alocate dinamic (de acelasi tip), dispersate în memorie, dar legate între ele prin pointeri, ca într-un lant. Intr-o listã liniarã simplu înlãntuitã fiecare element al listei contine adresa elementului urmãtor din listã. Ultimul element poate contine ca adresã de legaturã fie constanta NULL, fie adresa primului element din listã (lista circularã). Adresa primului element din listã este memoratã într-o variabilã cu nume si numitã cap de lista (“list head”). Pentru o listã vidã variabila cap de listã este NULL.

Structura de listã este recomandatã atunci când colectia de elemente are un continut foarte variabil (pe parcursul executiei) sau când trebuie pãstrate mai multe liste cu continut foarte variabil.

Un element din listã (un nod de listã) este de un tip structurã si are (cel putin) douã câmpuri: un câmp de date (sau mai multe) si un câmp de legãturã. Definitia unui nod de listã este o definitie recursivã, deoarece în definirea câmpului de legãturã se foloseste tipul în curs de definire. Exemplu pentru o listã de întregi:

```
 typedef struct snod {
```

```
    int val ;                  //  camp de date
    struct snod * leg ;        // camp de legatura
  } nod;
```

Programul următor aratã cum se poate crea si afisa o listã cu adãugare la început (o stivã realizatã ca listã înlãntuitã):

```
int main ( ) {
  nod *lst=NULL, *nou, * p;          // lst = adresa cap de lista
  int x;
   // creare lista cu numere citite
  while (scanf("%d",&x) > 0) {       // citire numar intreg x
    nou=(nod*)malloc(sizeof(nod));     // creare nod nou
    nou→val=x;                         // completare camp de date din nod
    nou→leg=lst; lst=nou;              // legare nod nou la lista
  }
   // afisare listã (fara modificare cap de lista)
  p=lst;
  while ( p != NULL) {               // cat mai sunt noduri
    printf("%d  ", p→val);           // afisare numar de la adr p
    p=p→leg;                           // avans la nodul urmator
  }
}
```

Câmpul de date poate fi la rândul lui o structurã specificã aplicatiei sau poate fi un pointer la date alocate dinamic (un sir de caractere, de exemplu).

De obicei se definesc functii pentru operatiile uzuale cu liste. Exemple:

```
typedef struct nod {           // un nod de lista inlantuita
  int val;                     // date din fiecare nod
  struct snod *leg;            // legatura la nodul urmator
  } nod;

  // insertie la inceput lista
nod* insL( nod* lst, int x) {
  nod* nou ;                   // adresa nod nou
  if ((nou=(nod*)malloc(sizeof(nod))) ==NULL)
    return NULL;
  nou→val=x; nou→leg=lst;
  return nou;                      // lista incepe cu “nou”
}
    // afisare continut lista
void printL ( nod* lst) {
  while (lst != NULL) {
     printf("%d  ",lst→val);       // scrie informatiile din nod
     lst=lst→leg;                // si se trece la nodul urmãtor
  }
}
int main () {                      // creare si afisare lista stiva
```

```
  nod* lst; int x;
  lst=NULL;                    // initial lista e vida
  while (scanf("%d",&x) > 0)
      lst=insL(lst,x);         // introduce pe x in lista lst
  printL (lst);                // afisare lista
}
```

Alte structuri dinamice folosesc câte doi pointeri sau chiar un vector de pointeri; într-un arbore binar fiecare nod contine adresa succesorului la stânga si adresa succesorului la dreapta, într-un arbore multicãi fiecre nod contine un vector de pointeri cãtre succesorii acelui nod.

Vectori de pointeri la date alocate dinamic

Ideea folositã la matrice alocate dinamic este aplicabilã si pentru alte date alocate dinamic: adresele acestor date sunt reunite într-un vector de pointeri. Situatiile cele mai frecvente sunt vectori de pointeri la siruri de caractere alocate dinamic si vectori de pointeri la structuri alocate dinamic.

Exemplu de utilizare a unui vector de pointeri la structuri alocate dinamic:

```
typedef  struct {int zi, luna, an; } date;
   // afisare date reunite în vector de pointeri
void print_vp ( date * vp[], int n) {
  int i;
  for(i=0;i<n;i++)
      printf ("%4d %4d %4d \n", vp[i] →zi, vp[i] →luna, vp[i] →an);
  printf ("\n");
}
int main () {
   date d, *dp;  date *vp[100];
   int n=0;
   while (scanf ("%d%d%d", &d.zi, &d.luna, &d.an) {
       dp= (date*) malloc (sizeof(date));       // alocare dinamica ptr structurã
       *dp=d;                                   // copiaza datele citite la dp
        vp[n++]=dp;                             // memoreaza adresa in vector
   }
   print_vp (vp,n);
}
```

De retinut este cã trebuie create adrese distincte pentru fiecare variabile structurã si cã ar fi gresit sã punem adresa variabilei “d” în toate pozitiile din vector. Este posibilã si varianta urmãtoare pentru ciclul principal din “main”:

```
  scanf (“%d”,&n);          // numar de structuri ce vor fi citite
  for (k=0;k<n;k++) {
       dp= (date*) malloc (sizeof(date));       // alocare dinamica ptr structurã
```

```
    scanf ("%d%d%d", &dp→zi, &dp→luna, &dp→an)
    vp[n++]=dp;                              // memoreaza adresa in vector
}
```

10. Operatii cu siruri de caractere în C

Memorarea sirurilor de caractere în C

In limbajul C nu există un tip de date "sir de caractere", desi există constante sir (între ghilimele). Sirurile de caractere se memorează în vectori cu componente de tip *char*, dar există anumite particularitãti în lucrul cu siruri fatã de lucrul cu alti vectori.

Sirurile de caractere reprezintã nume de persoane, produse, localitãti iar uneori chiar propozitii sau fragmente de texte. Prin natura lor sirurile pot avea o lungime variabilã în limite foarte largi, iar lungimea lor se poate modifica chiar în cursul executiei unui program ca urmare a unor operatii cum ar fi alipirea a douã siruri, stergerea sau inserarea unui subsir într-un sir s.a.

Operatiile uzuale cu siruri sunt realizate în C prin functii si nu prin operatori ai limbajului. O astfel de functie primeste unul sau douã siruri si eventual produce un alt sir (de obicei sirul rezultat înlocuieste primul sir primit de functie). Pentru fiecare sir functia ar trebui sã primeascã adresa de început a sirului (numele vectorului) si lungimea sa, lungime care se modificã la anumite operatii.

Pentru simplificarea listei de argumente si a utilizãrii functiilor pentru operatii cu siruri s-a decis ca fiecare sir memorat într-un vector sã fie terminat cu un octet zero ('\0') si sã nu se mai transmitã explicit lungimea sirului. Multe functii care produc un nou sir precum si functiile standard de citire adaugã automat un octet terminator la sirul produs (citit), iar functiile care prelucreazã sau afiseazã siruri detecteazã sfârsitul sirului la primul octet zero.

Citirea unui sir de la tastaturã se poate face fie cu functia "scanf" si descriptor "%s", fie cu functia "gets" astfel:

- Citirea unei linii care poate include spatii albe se va face cu "gets".
- Citirea unui cuvânt (sir delimitat prin spatii albe) se va face cu "scanf".

Ambele functii primesc ca argument adresa unde se citeste sirul si înlocuiesc caracterul '\n' introdus de la tastaturã cu terminatorul de sir (zero).

Exemplu de citire si afisare linii de text, cu numerotare linii:

```
int main () {
  char lin[128]; int nl=0;              // linii de maxim 128 car
  while ( gets (lin) != NULL) {     // citire linie in lin
    printf ("%4d ",++nl);           // mareste numar linie
    printf (%d: %s \n", nl, lin);   // scrie numar si continut linie
  }
}
```

Pentru a determina lungimea unui sir terminat cu zero se poate folosi functia de bibliotecã "strlen". Exemplu:

```
  while (scanf ("%s",sir) != EOF)
    printf ("sirul %s are %d caractere \n", sir, strlen(sir));
```

Nu se recomandã citirea caracter cu caracter a unui sir, cu descriptorul “%c” sau cu functia “getchar()”, decât dupã apelul functiei “fflush”, care goleste zona tampon de citire. In caz contrar se citeste caracterul ‘\n’ (cod 10), care rãmâne în zona tampon dupã citire cu “scanf(“%s”,..) sau cu getchar().

Pentru a preveni erorile de depãsire a zonei alocate pentru citirea unui sir se poate specifica o lungime maximã a sirului citit în functia “scanf”. Exemplu:

```
char nume[30];
while (scanf (“%30s”,nume) != EOF)
     printf (“%s \n”, nume);              // numai primele 30 de caractere citite
```

Memorarea unei liste de siruri se poate face într-o matrice de caractere în care fiecare linie din matrice reprezintã un sir, dar solutia este ineficientã dacã sirurile au lungime foarte variabilã, pentru cã numãrul de coloane din matrice este determinat de lungimea maximã a unui sir. Exemplu:

```
char kwords [5][8] =  {"int","char","float","long","double","short"};
   // cauta un cuvant in tabelul de cuv cheie
  int keyw ( char nume[8], char kw[][8], int n ) {
    int i;
    for (i=0;i<n;i++)
      if (strcmp(nume,kw[i])==0)
       return i;
    return -1;
  }
```

O solutie care foloseste mai bine memoria este alocarea dinamicã de memorie (la executie) pentru fiecare sir, în functie de lungimea lui si reunirea adreselor acestor siruri într-un vector de pointeri. Solutia corespunde unei matrice cu linii de lungimi diferite, alocate dinamic.

Functia standard “strdup” (duplicare sir) primeste un sir pe care îl copiazã la o adresã obtinutã printr-o cerere de alocare :

```
char * strdup ( char * adr) {
  int len=strlen(adr);           // lungime sir de la adresa adr
  char * rez = (char*) malloc(len+1);  // aloca memorie pentru sir si terminator
  strcpy (rez,adr);              // copiaza sir de la adr la adresa rez
  return rez;                    // rezultatul este adresa duplicatului
}
```

Functia “strdup” se foloseste pentru crearea de adrese distincte pentru mai multe siruri citite într-o aceeasi zona buffer. Exemplu:

```
int main () {
  char buf[40]; char* tp[100];        // tp este un tabel de pointeri la siruri
```

```
  int i=0,j;
  while (gets(buf))                  // gets are rezultat zero la sfârsit de fisier (^Z)
    tp[i++]= strdup(buf);              // copiaza sirul citit si memoreaza adresa
copiei
  for (j=0;j<i;j++)                  // afisarea sirurilor
    puts(tp[j]);                     // folosind tabelul de pointeri
}
```

Din familia functiilor de intrare-iesire se considerã cã fac parte si functiile standard “sscanf” si “sprintf”, care au ca prim argument un sir de caractere ce este analizat (“scanat”) de “sscanf” si respectiv produs de “sprintf” (litera ‘s’ provine de la cuvântul “string”). Aceste functii se folosesc fie pentru conversii interne în memorie, dupã citire sau înainte de scriere din/în fisiere text, fie pentru extragere de subsiruri dintr-un sir cu delimitatori diferiti de spatii albe. Exemplu:

```
   // extragere zi, luna si an dintr-un sir de forma  zz-ll-aaaa
int main () {
  char d[ ]="25-12-1989";  int z,l,a;
  sscanf (d,"%d-%d-%d", &z, &l, &a);
  printf ("\n %d ,%d, %d \n", z, l, a);
}
```

Erori uzuale la operatii cu siruri de caractere

Numele unui vector este un pointer si nu mai trebuie aplicat operatorul ‘&’ de obtinere a adresei, asa cum este necesar pentru variabile simple. Exemplu de citire a unui singur caracter (urmat de “Enter”) în douã feluri; diferenta dintre ele apare la citirea repetatã de caractere individuale.

```
    char c, s[2];
    scanf (“%c”, &c);  scanf(“%1s”, s);
```

Poate cea mai frecventã eroare de programare (si care nu se manifestã întotdeauna ca eroare, la executie) este utilizarea unei variabile pointer neinitializate în functia “scanf” (sau “gets”), datoritã confuziei dintre vectori si pointeri. Exemplu gresit:

```
  char * s;         // corect este: char s[80;  80= lungime maxima
  scanf (“%s”,s);    // citeste la adresa continuta in “s”
```

O altã eroare frecventã (nedetectatã la compilare) este compararea adreselor a douã siruri în locul comparatiei celor douã siruri. Exemplu:

```
  char a[50], b[50];          // aici se memoreaza doua siruri
  scanf (%50s%50s”, a,b);   // citire siruri a si b
  if (a==b) printf(“egale\n”); //gresit,rezultat zero
```

Pentru comparare corectã de siruri se va folosi functia “strcmp”. Exemplu :

```
if (strcmp(a,b)==0)  printf (“egale\n”);
```

Aceeasi eroare se poate face si la compararea cu un sir constant. Exemple:

```
if ( nume == “." ) break; ...}                  // gresit !
if ( strcmp(nume,”.”) == 0 ) break;... }   // corect
```

Din aceeasi categorie de erori face parte atribuirea între pointeri cu intentia de copiere a unui sir la o altã adresã, desi o parte din aceste erori pot fi semnalate la compilare. Exemple:

```
char a[100], b[100], *c ;
 // memorie alocata dinamic la adresa “c”
c = (char*) malloc(100);
a = b;                  // eroare la compilare
 // corect sintactic dar nu copiaza sir (modifica  “c”)
c = a;
 // copiaza sir de la adresa “a” la adresa “c”
strcpy (c,a);
   // copiaza la adresa “a” sirul de la adresa “b”
strcpy (a,b);
```

Functiile standard "strcpy" si strcat" adaugã automat terminatorul zero la sfârsitul sirului produs de functie.

Functii standard pentru operatii cu siruri

Principalele categorii de functii care lucrează cu siruri de caractere sunt:
- Functii pentru siruri terminate cu zero (siruri complete); numele lor începe cu “str”.
- Functii pentru subsiruri de lungime maximã; numele lor începe cu “strn”
- Functii pentru operatii cu blocuri de octeti (neterminate cu zero); numele lor începe cu “mem”.

Aceste functii sunt declarate în fisierele <string.h> si <mem.h>, care trebuie incluse în compilarea programelor care lucrează cu siruri de caractere (alãturi de alte fisiere de tip “h”).

Urmează o descriere putin simplificatã a celor mai folosite functii standard pe siruri de caractere.

```
    // strlen:  lungimea sirului “s” ( “s” terminat cu un octet zero)
int strlen(char * s);
   // strcmp:  comparã sirurile de la adresele s1 si s2
int strcmp (char * s1, char * s2);
  // strncmp:  comparã primele n caractere din sirurile s1 si s2
```

```
int strncmp ( char * s1, char * s2, int n);
  // copiazã la adresa “d” tot sirul de la adresa “s” (inclusiv terminator sir)
char * strcpy (char * d, char * s);
  // strncpy:  copiazã primele n caractere de la “s” la “d”
char * strncpy ( char *d, char * s, int n);
  // strcat:  adaugã sirul “s” la sfârsitul sirului “d”
 char * strcat (char *d, char* s);
  // strncat:  adaugã primele n car. de la adresa “s” la sirul “d”
 char * strncat (char *d, char *s, int n);
  // strchr: are ca rezultat pozitia lui “c” în sirul “d” (prima aparitie a lui c)
 char * strchr (char *d, char c);
   // cautã ultima aparitie a lui “c” în sirul “d”
 char *strrchr (char *d,char c);
  // strstr: are ca rezultat adresa în sirul “d” a sirului “s”
 char * strstr (char *d, char*s);
 // stristr: la fel ca strstr dar ignorã diferenta intre litere mici si mari
 char * strstr (char *d, char*s);
```

Functia “strncpy” nu adaugã subsirului copiat la adresa “d” terminatorul de sir atunci dacã n<strlen(s) dar subsirul este terminat cu zero dacã n>strlen(s).

Functiile de comparare siruri au urmãtorul rezultat:

== 0 dacã sirurile comparate contin aceleasi caractere (sunt identice)

< 0 dacã primul sir (s1) este inferior celui de al doilea sir (s2)

> 0 dacã primul sir (s1) este superior celui de al doilea sir (s2)

Rezultatul functiei de comparare nu este doar -1, 0 sau 1 ci orice valoare întreagã cu semn, deoarece comparatia de caractere se face prin scãdere. Exemplu de implementare a functiei “strcmp”:

```
int strcmp ( char * s1, char * s2) {
  while (*s1 || *s2)          // repeta cât mai sunt caractere în s1 sau în s2
     if (*s1 == *s2) {             // daca sunt caractere egale
       s1++;s2++;             // continua comparatia cu caracterele urmatoare
     }
     else                        // daca *s1 != *s2
       return *s1-*s2;        // rezultat < 0 daca s1<s2
  return 0;                   // siruri egale pe toata lungimea
}
```

Se poate defini o functie care sã facã mai evidentã comparatia la egalitate:

```
  // rezultat 1 dacã siruri egale , 0 dacã siruri diferite
 int strequ (char * s1, char * s2) {
   return strcmp(s1,s2)==0;
 }
```

Functiile de copiere si de concatenare au ca rezultat primul argument (adresa sirului destinatie) pentru a permite exprimarea mai compactã a unor operatii succesive pe siruri. Exemplu:

```
// compune un nume de fisier din nume si extensie
char fnume[20], *nume="test", *ext="cpp";
strcat(strcat(strcpy(fnume,nume),"."),ext);
```

Utilizarea unor siruri constante în operatii de copiere sau de concatenare poate conduce la erori prin depãsirea memoriei alocate (la compilare) sirului constant. Exemplu gresit :

```
strcat ("test",".cpp");       // efecte nedorite !
```

Functiile pentru operatii pe siruri nu pot verifica depãsirea memoriei alocate pentru siruri, deoarece primesc numai adresele sirurilor; cade în sarcina programatorului sã asigure memoria necesarã rezultatului unor operatii cu siruri. Exemplu corect:

```
char nume[30]="test";
strcat (nume,".cpp");
```

Definirea de noi functii pe siruri de caractere

Argumentele de functii ce reprezintã siruri se declarã de obicei ca pointeri dar se pot declara si ca vectori. Ca exemple vom scrie în douã moduri o functie de copiere care functioneazã corect chiar dacã cele douã siruri se suprapun partial (functia standard "strcpy" nu garanteazã aceastã comportare) :

```
// cu pointeri
char* scopy ( char * d, char * s) {
   char * aux = strdup(s);
   strcpy(d,aux);
   free (aux);
   char * d;
}
   // cu vectori
 void scopy ( char d[ ], char s[ ]) {
   int k,n;  char aux[2000];          // o limita ptr sirurile copiate !
   n=strlen(s)+1;                     // nr de caractere copiate (si terminator de sir)
   for (k=0; k<n ; k++)               // copiaza in aux din s
      aux[k]=s[k];
   for (k=0; k<n ; k++)               // copiaza in d din aux
      d[k]=aux[k];
 }
```

Functiile standard pe siruri din C lucrează numai cu adrese absolute (cu pointeri) si nu folosesc ca argumente adrese relative în sir (indici întregi). Nu există functii care să

elimine un caracter dintr-un sir, care să insereze un caracter într-un sir sau care să extragă un subsir dintr-o pozitie datã a unui sir.

La definirea unor noi functii pentru operatii pe siruri programatorul trebuie să asigure adãugarea terminatorului de sir la rezultatul functiei, pentru respectarea conventiei si evitarea unor erori. Exemplu:

```
  // transforma un intreg intr-un sir de caractere (cifre in baza radix)
char *itoa(int value, char *string, int radix) {   // acelasi prototip cu functia standard
 char digits[] = "0123456789ABCDEF";       // dar “radix” poate fi cel mult 16
 char t[20], *tt=t, * s=string;          // adresa “string” se modifica !
 do {
  *tt++ = digits [value % radix];     // sau:  t[i++] = digits [value/radix];
 } while ( (value = value / radix) != 0 );
 while ( tt != t)                  // copiere de la tt la string in ordine inversa
   *string++= *(--tt);
 *string=0;                        // adauga terminator de sir
 return s;                         // rezultatul este adresa sirului creat
}
```

Functiile care produc ca rezultat un nou sir primesc o adresã (un pointer), unde se depune sirul creat în functie. Functia urmãtoare modificã sau adaugã o extensie datã la un nume de fisier, fãrã să modifice numele primit:

```
char* modext (char* oldfile, char* newfile, char *ext) {
  char * p;
  strcpy(newfile,oldfile);
  p =strrchr(newfile,'.');                  // cauta pozitia ultimului punct din nume
  if (p==NULL)                              // daca nu are extensie
    strcat( strcat(newfile,"."),ext);       // atunci se adauga extensia “ext”
  else                                      // daca avea o extensie
    strcpy (p+1,ext);                       // atunci se inlocuieste cu extensia “ext’
  return newfile;                           // dupa modelul functiilor standard strcpy,…
}
```

Functia care apeleazã pe “modext” trebuie să asigure suficientã memorie la adresa “newfile”, dar compilatorul nu poate verifica aceastã conditie. Exemplu:

```
  int main ( ) {
     char fname[]= “prog.com”, ext[]= ”exe”;
     char file[80];                         // extensie la nume fisier (tip fisier)
     modext (fname, file,ext);              // modificare extensie
     puts(file);                            // afisare ptr verificare
  }
```

O eroare posibilã si care trece de compilare ar fi fost urmãtoarea declaratie:

```
  char * file;        // pointer neinitializat, nu se aloca memorie
```

O versiune mai sigurã si cu mai putine argumente ar folosi alocarea dinamicã de memorie pentru noul nume de fisier creat de functie:

```
char* modext (char* oldfile, char *ext) {
  char * p, *newfile;
  newfile = (char*) calloc (strlen(oldfile) + strlen(ext)+1,1);  // aloca memorie
  strcpy(newfile,oldfile);                  // copiaza vechiul nume
  p =strrchr(newfile,'.');                  // cauta pozitia ultimului punct din nume
  if (p != NULL)                            // daca exista o extensie
    *p=0;                                   // elimina extensia veche din nume
  strcat( strcat(newfile,"."),ext);         // atunci se adauga extensia "ext"
  return newfile;                           // rezultatul functiei
}
```

In general nu se scriu functii care sã aloce memorie fãrã sã o elibereze, deoarece apelarea repetatã a unor astfel de functii poate duce la consum inutil de memorie. Nici nu este bine ca eliberarea memoriei alocate sã revinã celui care apeleazã functia.

O solutie înselãtoare este utilizarea unui vector local cu dimensiune constantã:

```
char* modext (char* oldfile, char *ext) {
  char * p, newfile[80];
  strcpy(newfile,oldfile);                  // copiaza vechiul nume
  . . .  }
```

In general nu se recomandã functii care au ca rezultat adresa unei variabile locale, desi erorile de utilizare a unor astfel de functii apar numai la apeluri succesive (în cascadã) de functii cu variabile locale. Exemplu:

```
   // extrage un subsir de lungime n de la adresa s
char * substr (char* s, int n) {
  char aux[1000];       // o variabila locala pentru subsirul rezultat
  int m;
  m=strlen(s);          // cate caractere mai sunt la adresa s
  if (n>m) n=m;
  strncpy(aux,s,n);
  aux[n]=0;             // terminator de (sub)sir
  return aux;           // pointer la o variabila locala !
}
  // verificare functie
int main () {
  puts (substr("abcdef",3));                // corect: abc
  puts (substr (substr("abcdef",4),2));     // corect sau nu, functie de
implementare
}
```

Functia “substr” va avea rezultate corecte si la al doilea apel dacã se modificã declararea variabilei “aux” astfel:

```
char * aux=strdup(s);        // sau:  aux= (char*) malloc(strlen(s)+1);
```

Pentru realizarea unor noi operatii cu siruri se vor folosi pe cât posibil functiile existente. Exemple:

```
   // sterge n caractere de la adresa “d”
char * strdel ( char *d, int n) {
  if ( n < strlen(d))
    strcpy(d,d+n);
  return d;
}
// inserează sirul s la adresa d
void strins (char *d, char *s) {
  int ls=strlen(s);
  scopy (d+ls,d);                  // deplasare sigura la dreapta sir d
  strncpy(d,s,ls);                 // adauga sirul s la adresa d
}
```

Precizări la declararea functiile standard sau nestandard pe siruri :

- Argumentele sau rezultatele ce reprezintã lungimea unui sir sunt de tip ***size_t*** (echivalent de obicei cu ***unsigned int***) si nu ***int***, pentru a permite siruri de lungime mai mare.
- Argumentele ce reprezintã adrese de siruri care nu sunt modificate de functie se declarã astfel: `const char * str`

interpretat ca “pointer la un sir constant (nemodificabil)”. Exemplu:

```
size_t  strlen (const char * s);
```

Cuvântul cheie ***const*** în fata unei declaratii de pointer cere compilatorului sã verifice cã functia care are un astfel de argument nu modificã datele de la acea adresã. Exemplu de functie care produce un mesaj la compilare:

```
void trim (const char*s) {          // elimina toate spatiile dintr-un sir dat
   char *p=s;                       // avertisment la aceasta linie !
   while (*s) {
      while (*s==' ')            // sau:     while (isspace(*s))
        ++s;
      if (s==0) break;
     *p++=*s++;
   }
   *p=0;
}
```

Cum poate însã compilatorul C sã verifice cã un pointer “const char * ” transmis unei alte functii nu este folosit în acea functie pentru modificarea datelor ?

Verificarea este posibilã dacã functia apelatã declarã si ea argumentul pointer cu *const*. In exemplul urmãtor avem o functie care gãseste cuvântul k dintr-un sir fãrã sã modifice sirul primit, dar foloseste functia standard “strtok”. Compilatorul semnalează o conversie dubioasã de pointeri, de la “const char*” la “char*”.

```
char* kword (const char* s, int k) {
  char * p=s;          // mai bine:     p = strdup(s);
  p=strtok(p,” “);
  if (k==0 && p > 0) return p;
  while ( k-- && p!=NULL)
     p=strtok(0,” “);
  return p;
}
```

Solutia la problema anterioarã este fie copierea sirului primit într-un alt sir sau utilizarea unei alte functii în locul functiei “strtok”:

Toate functiile de bibliotecã cu pointeri la date nemodificabile folosesc declaratia *const*, pentru a permite verificări în alte functii scrise de utilizatori dar care folosesc functii de bibliotecã.

Extragerea de cuvinte dintr-un text

Un cuvânt sau un atom lexical (“token”) se poate defini în douã feluri:

- un sir de caractere separat de alte siruri prin unul sau câteva caractere cu rol de separator între cuvinte (de exemplu, spatii albe);
- un sir care poate contine numai anumite caractere si este separat de alti atomi prin oricare din caracterele interzise în sir.

In primul caz sunt putini separatori de cuvinte si acestia pot fi enumerati. Pentru extragerea de siruri separate prin spatii albe (‘ ‘,’\n’,’\t’,’\r’) se poate folosi o functie din familia “scanf” (“fscanf” pentru citire dintr-un fisier, “sscanf” pentru extragere dintr-un sir aflat în memorie). Intre siruri pot fi oricâte spatii albe, care sunt ignorate. Exemplu de functie care furnizează cuvântul k dintr-un sir dat s:

```
char* kword (const char* s, int k) {
   char word[256];
   while ( k >= 0) {
     sscanf(s,"%s",word);
     s=s+strlen(word);
     k--;
   }
   return word;       // sau:  return strdup(word)
}
```

Pentru extragere de cuvinte ce pot fi separate si prin alte caractere (‘,’ sau ’;’ de ex.) se poate folosi functia de biblioteca “strtok”, ca în exemplul urmãtor:

```
main ( ) {
   char linie[128], * cuv;      // adresa cuvant in linie
   char *sep=“.,;\t\n “         // sir de caractere separator
   gets(linie);                 // citire linie
   cuv=strtok (linie,sep);      // primul cuvant din linie
   while ( cuv !=NULL) {
      puts (cuv);                   // scrie cuvant
      cuv=strtok(0,sep);        // urmatorul cuvant din linie
   }
 }
```

Functia “strtok” are ca rezultat un pointer la urmãtorul cuvânt din linie si adaugã un octet zero la sfârsitul acestui cuvânt, dar nu mutã la altã adresã cuvintele din text. Acest pointer este o variabilã localã staticã în functia “strtok”, deci o variabilã care îsi pãstreazã valoarea între apeluri succesive.

Exemplu de implementare a functiei “strtok” cu variabilã staticã :

```
char *strtok (char * sir,char *separ) {
 static char *p;              // p este o variabila staticã !
 char * r;
 if (sir)  p=sir;             // la primul apel
 while (strchr(separ,*p) && *p )     // ignora separatori intre atomi
   p++;
 if (*p=='\0') return NULL;        // dacã s-a ajuns la sfârsitul sirului analizat
  r=p;                  // r = adresa unde incepe un atom
 while (strchr(separ,*p)==NULL && *p)   // caractere din atomul curent
   p++;
 if (p==r) return NULL;            // p = prima adresa dupa atom
 else {
  *p++='\0'; return r;           // termina atom cu octet zero
 }
}
```

Extragerea de cuvinte este o operatie frecventã, dar functia “strtok” trebuie folositã cu atentie deoarece modificã sirul primit si nu permite analiza în paralel a douã sau mai multe siruri. De aceea este preferabilã o altã functie, cum ar fi urmãtoarea:

```
char * stok (char * sir, char *tok) {     // depune urmatorul atom la adresa “tok’
  char * r, *p=sir ;
  // ignora separatori intre atomi
  while ( *p && *p ==' ')
     p++;
  if (*p=='\0') return 0;
  r=p;    // r= adresa de inceput token
    // caractere din token
  while (*p && *p !=' ')
     p++;
```

```
  if (p==r) return 0;
  else {
    strncpy(tok,r,p-r); tok[p-r]=0;
    return p;
  }
}
   // utilizare functie
int main () {
  char lin[]="  unu  doi   trei  patru ";
  char * p=lin; char t[256];
  while ( p=stok(p,t))       // de la adresa p extrage cuvant in t
     puts(t);                // afisare cuvant
}
```

In al doilea caz sunt mai multi separatori posibili decât caractere admise într-un atom; un exemplu este un sir de cifre zecimale sau un sir de litere (mari si mici) si separat de alte numere sau nume prin oricare alte caractere. Extragerea unui sir de cifre sau de litere trebuie realizatã de programator, care poate folosi functiile pentru determinarea tipului de caracter, declarate în fisierul antet <ctype.h>. Exemplu:

```
  // extragere cuvinte formate numai din litere
int main ( ) {
  char linie[80], nume[20], *adr=linie; int i;
  gets(linie);
  while (*adr) {
    // ignora alte caractere decât litere
    while (*adr && !isalpha(*adr)) ++adr;      // isalpha declarat in <ctype.h>
    if (*adr==0) break;          // daca sfarsit de linie
    for (i=0; isalpha(*adr); adr++, i++)
       nume[i]=*adr;             // extrage cuvant in “nume”
    nume[i]=0;                   // terminator de sir C
    puts (nume);                     // afiseaza un nume pe o linie
  }
}
```

Cãutarea si înlocuirea de siruri

Orice editor de texte permite cãutarea tuturor aparitiilor unui sir si, eventual, înlocuirea lor cu un alt sir, de lungime mai micã sau mai mare. De asemenea, existã comenzi ale sistemelor de operare pentru cãutarea unui sir în unul sau mai multe fisiere, cu diferite optiuni (comanda “Find” în MS-DOS si MS-Windows).

Cãutarea de cuvinte complete poate folosi functiile “strtok” si “strcmp”, iar cãutarea de subsiruri în orice context (ca pãrti de cuvinte) poate folosi functia “strstr”. In secventa urmãtoare se cautã si se înlocuiesc toate aparitiile sirului s1 prin sirul s2 într-o linie de text, memoratã la adresa “line”:

```
 while(p=strstr(line,s1)) {     // adresa lui s1 în line
```

```
   strdel (p,strlen(s1));       // sterge caractere de la adr p
   strins(p,s2);                // insertie de caractere la p
}
```

In exemplul anterior am presupus cã sirul nou s2 nu contine ca subsir pe s1, dar mai sigur este ca sã mãrim adresa din “p” dupã fiecare înlocuire:

```
 p=line;
 while(p=strstr (p,s1)) {     // cauta un s1 de la p
   strdel (p,strlen(s1));
   strins(p,s2);
   p=p+ strlen(s2);
 }
```

Uneori se cautã siruri care se “potrivesc” (“match”) cu o mascã (cu un sablon) care contine si caractere speciale pe lângã caractere obisnuite. Cea mai cunoscutã formã de mascã este cea care foloseste caracterele “wildcards” ‘*’ si ‘?’, cu semnificatia “subsir de orice lungime si orice caractere” si respectiv “orice caracter”. O mascã se poate potrivi cu multe siruri diferite (în continut dar si în lungime).

Aceastã mascã este folositã în principal la selectarea de fisiere dintr-un director fie în comenzi sistem, fie în functii de bibliotecã (“findfirst” si “findnext”, pentru cãutarea de fisiere).

Exemple de cãutare cu mascã (primul argument este masca de cãutare):

```
match (“a?c?”, “abcd”);      // da
match (“?c*”, “abcd”);           // da
match (“*ab*”, “abcd”);      // da
match (“*?*”, “abcd”);       // da
```

O functie pentru potrivirea unui sir dat cu o mascã este un exemplu de functie greu de validat, datoritã diversitãtii de situatii în care trebuie testatã (verificatã). Exemplu de functie recursivã pentru potrivire cu caractere “wildcard”:

```
int match (char* pat, char* str) {   // pat=pattern, str=string
  while (*str) {
    switch (*pat) {
      case '?':
        if (*str == '.') return 0;      // pentru nume de fisiere !
        break;
      case '*':
        do { ++pat; } while (*pat == '*');  // caractere * succesive
        if (!*pat) return 1;
        while (*str) if (match(pat, str++)) return 1;
        return 0;
      default:                          // orice caracter diferit de ? sau *
        if (toupper(*str) != toupper(*pat)) return 0;       // daca difera
        break;
```

```
    }
    ++pat, ++str;                    // avans in sir si in masca
  }
  while (*pat == '*') ++pat;         // caractere * succesive
  return !*pat;
}
```

In cazul cel mai general masca de cãutare este o expresie regulatã, cu caractere "wildcards" dar si cu alte caractere speciale, ce permit exprimarea unor conditii diverse asupra sirurilor cãutate. Exemple de expresii regulate:

Biblioteci de functii pentru folosirea de expresii regulate în limbajul C, dar si în alte limbaje (C++, Java, Perl, s.a.) sunt disponibile si pot fi folosite gratuit.

Argumente în linia de comandã

Functia "main" poate avea douã argumente, prin care se pot primi date transmise prin linia de comandã ce lanseazã programul în executie. Sistemul de operare analizeazã linia de comandã, extrage cuvintele din linie (siruri separate prin spatii albe), alocã memorie pentru aceste cuvinte si introduce adresele lor într-un vector de pointeri (alocat dinamic).

Primul argument al functiei "main" este dimensiunea vectorului de pointeri (de tip *int*), iar al doilea argument este adresa vectorului de pointeri (un pointer). Exemplu:

```
int main ( int argc, char * argv[]) {
  int i;
  for (i=1;i<n;i++)                / nu se afiseaza si argv[0]
    printf ("%s ", argv[i]);
}
```

Primul cuvânt, cu adresa în argv[0], este chiar numele programului executat (numele fisierului ce contine programul executabil), iar celelalte cuvinte din linie sunt date initiale pentru program: nume de fisiere folosite de program, optiuni de lucru diverse.

De obicei se extrag în ordine toate argumentelor din linia de comandã si deci putem renunta la indici în referirea la componentele vectorului de pointeri. Exemplu:

```
int main ( int argc, char ** argv) {
  while (argc --)
    printf ("%s ", *argv++);
}
```

Modul de interpretare al argumentelor din linia de comandã poate depinde de pozitia lor în lista de argumente si/sau de prezenta unor caractere prefix (minus de obicei pentru optiuni de lucru). S-a propus chiar un standard POSIX pentru unificarea

modului de interpretare al argumentelor din linia de comandã si existã (sub)programe care prelucreazã aceste argumente conform standardului.

11. Fisiere de date în C

Tipuri de fisiere

Un fisier este o colectie de date memorate pe un suport extern si care este identificatã printr-un nume. Continutul fisierelor poate fi foarte variat: texte, inclusiv programe sursã, numere sau alte informatii binare: programe executabile, numere înformat binar, imagini sau sunete codificate numeric s.a. Fisierele de date se folosesc fie pentru date initiale si pentru rezultate mai numeroase, fie pentru pãstrarea permanentã a unor date de interes pentru anumite aplicatii.

Fisierele sunt entitãti ale sistemului de operare si ca atare ele au nume care respectã conventiile sistemului, fãrã legãturã cu un limbaj de programare. Operatiile cu fisiere sunt realizate de cãtre sistemul de operare, iar compilatorul unui limbaj traduce functiile de acces la fisiere din limbaj în apeluri ale functiilor de sistem.

Programatorul se referã la un fisier printr-o variabilã; tipul acestei variabile depinde de limbajul folosit si chiar de functiile utilizate (în C). Asocierea dintre numele extern (un sir de caractere) si variabila din program se face la deschiderea unui fisier, printr-o functie standard.

De obicei prin "fisier" se subîntelege un fisier disc (pe suport magnetic sau optic), dar notiunea de fisier este mai generalã si include orice flux de date din exterior spre memorie sau dinspre memoria internã spre exterior. Cuvântul “stream”, tradus prin flux de date, este sinonim cu “file” (fisier), dar pune accent pe aspectul dinamic al transferului de date între memoria internã si o sursã sau o destinatie externã a datelor (orice dispozitiv periferic).

Pentru fisierele disc un nume de fisier poate include urmãtoarele:

- Numele unitãtii de disc sau partitiei disc (ex: A:, C:, D:, E:)
- "Calea" spre fisier, care este o succesiune de nume de fisiere catalog (director), separate printr-un caracter ('\' în MS-DOS si MS-Windows, sau '/' în Unix si Linux)
- Numele propriu-zis al fisierului (max 8 litere si cifre în MS-DOS)
- Extensia numelui, care indicã tipul fisierului (continutul sãu) si care poate avea între 0 si 3 caractere în MS-DOS).

Exemple de nume de fisiere disc:

```
A:bc.rar , c:\borlandc\bin\bc.exe
c:\work\p1.cpp , c:\work\p1.obj
```

Sistemele MS-DOS si MS-Windows nu fac deosebire între litere mari si litere mici, în cadrul numelor de fisiere, dar sistemele de tip Unix sau Linux fac deosebire între litere mari si litere mici.

Consola si imprimanta sunt considerate fisiere text, adicã:

- între aceste fisiere si memorie se transferã caractere ASCII
- se recunoaste caracterul sfârsit de fisier (Ctrl-Z în MS-DOS si MS-Windows)
- se poate recunoaste la citire un caracter terminator de linie ('\n').

Un fisier text pe disc contine numai caractere ASCII, grupate în linii de lungimi diferite, fiecare linie terminatã cu unul sau douã caractere terminator de linie (fisierele Unix/Linux folosesc un singur caracter terminator de linie '\n' , iar fisierele Windows si MS-DOS folosesc caracterele '\r' si '\n' (CR,LF) ca terminator de linie.

Un fisier text poate fi terminat printr-un caracter terminator de fisier (Ctrl-Z= \0x1A), dar nu este obligatoriu acest terminator. Sfârsitul unui fisier disc poate fi detectat si pe baza lungimii fisierului (numãr de octeti), memoratã pe disc.

Functiile de citire sau de scriere cu format din/în fisiere text realizeazã conversia automatã din format extern (sir de caractere) în format intern (binar virgulã fixã sau virgulã mobilã) la citire si conversia din format intern în format extern, la scriere pentru numere întregi sau reale.

Fisierele disc pot contine si numere în reprezentare internã (binarã) sau alte date ce nu reprezintã numere (de exemplu, fisiere cu imagini grafice, în diverse formate). Aceste fisiere se numesc fisiere binare, iar citirea si scrierea se fac fãrã conversie de format. Pentru fiecare tip de fisier binar este necesar un program care sã cunoascã si sã interpreteze corect datele din fisier (structura articolelor).

Este posibil ca un fisier binar sã continã numai caractere, dar functiile de citire si de scriere pentru aceste fisiere nu cunosc notiunea de linie; ele specificã un numãr de octeti care se citesc sau se scriu la un apel al functiei "fread" sau "fwrite".

Fisierele disc trebuie deschise si închise, dar fisierele consolã si imprimanta nu trebuie deschise si închise.

Functii pentru deschidere si închidere fisiere

In C sunt disponibile douã categorii de functii pentru acces la fisiere:

- Functii stil Unix, declarate în fisierul "io.h" si care se referã la fisiere prin numere întregi.
- Functii standard, declarate în fisierul "stdio.h" si care se referã la fisiere prin pointeri la o structurã predefinitã ("FILE").

In continuare vor fi prezentate numai functiile standard, din <stdio.h>.

Pentru a citi sau scrie dintr-un /într-un fisier disc, acesta trebuie mai întâi deschis folosind functia "fopen". La deschidere se precizeazã numele fisierului, tipul de fisier (text/binar) si modul de exploatare: numai citire, numai scriere (creare) sau citire si scriere (modificare).

La deschiderea unui fisier se initializeazã variabila pointer asociatã, iar celelalte functii (de acces si de închidere) se referã la fisier numai prin intermediul variabilei pointer. Exemplu:

```
#include <stdio.h>
void main ( ) {
  FILE * f;              // pentru referire la fisier
    // deschide un fisier text ptr citire
  f = fopen ("t.txt","rt");
```

```
  printf ( f == NULL? "Fisier negasit" : " Fisier gasit");
  if (f)                  // daca fisier existent
    fclose(f);            // inchide fisier
}
```

Functia "fopen" are rezultat NULL (0) dacã fisierul specificat nu este gãsit dupã cãutare în directorul curent sau pe calea specificatã.

Primul argument al functiei "fopen" este numele extern al fisierului scris cu respectarea conventiilor limbajului C: pentru separarea numelor de cataloage dintr-o cale se vor folosi douã caractere "\\", pentru a nu se considera o secventã de caractere "Escape" a limbajului. Exemple:

```
  char *numef = "C:\\WORK\\T.TXT";     // sau    c:/work/t.txt
  FILE * f;
  if ( (f=fopen(numef,"r")) == NULL) {
     printf("Eroare la deschidere fisier %s \n", numef);
     return;
  }
```

Al doilea argument al functiei "fopen" este un sir care poate contine între 1 si 3 caractere, dintre urmãtoarele caractere posibile:

"r","w","a" = mod de folosire ("read", "write", "append")
"+" dupã "r" sau "a" pentru citire si scriere din acelasi fisier
"t" sau "b" = tip fisier ("text", "binary"), implicit este "t"

Diferenta dintre “b” si “t” este aceea cã la citirea dintr-un fisier binar toti octetii sunt considerati ca date si sunt transferati în memorie, dar la citirea dintr-un fisier text anumiti octeti sunt interpretati ca terminator de linie (\0x0a) sau ca terminator de fisier (\0x1a). Nu este obligatoriu ca orice fisier text sã se termine cu un caracter special cu semnificatia “sfârsit de fisier” (ctrl-z , de exemplu) .

Pentru fisierele text sunt folosite modurile "w" pentru crearea unui nou fisier, "r" pentru citirea dintr-un fisier si "a" pentru adãugare la sfârsitul unui fisier existent. Modul “w+” poate fi folosit pentru citire dupã creare fisier.

Pentru fisierele binare se practicã actualizarea pe loc a fisierelor, fãrã inserarea de date între cele existente, deci modurile "r+","a+","w+". (literele 'r' si 'w' nu pot fi folosite simultan).

Inchiderea unui fisier disc este absolut necesarã pentru fisierele în care s-a scris ceva, dar poate lipsi dacã s-au fãcut doar citiri din fisier.

Este posibilã deschiderea repetatã pentru citire a unui fisier disc (fãrã o închidere prealabilã), pentru repozitionare pe început de fisier.

Deschiderea în modul “w” sterge orice fisier existent cu acelasi nume, fãrã avertizare, dar programatorul poate verifica existenta unui fisier în acelasi director înainte de a crea unul nou. Exemplu:

```
int main () {
  FILE *f ; char numef[100];          // nume fisier
```

```
  char rasp;                        // raspuns la intrebare
  puts("Nume fisier nou:"); gets(numef);
  f = fopen(numef,"r");             // cauta fisier (prin deschidere in citire)
  if ( f ) {                        // daca exista fisier cu acest nume
     printf ("Fisierul exista deja! Se sterge ? (d/n) ");
     rasp= getchar();               // citeste raspuns
     if ( rasp =='n' || rasp =='N')
     return;                        // terminare program
  }
  f=fopen(numef,"w");               // deschide fisier ptr creare
  fputs (numef,f);                  // scrie in fisier chiar numele sau
  fclose(f);
}
```

Fisierele standard de intrare-iesire (tastatura si ecranul consolei) au asociate variabile de tip pointer cu nume predefinit ("stdin" si "stdout"), care pot fi folosite în diferite functii, dar practic se folosesc numai in functia "fflush" care goleste zona tampon ("buffer") asociatã unui fisier.

Operatiile de stergere a unui fisier existent ("remove") si de schimbare a numelui unui fisier existent ("rename") nu necesitã deschiderea fisierelor.

Operatii uzuale cu fisiere text

Aici este vorba de fisiere text ce contin programe sursã sau documentatii si nu de fisiere text ce contin numere în format extern. Accesul la fisiere text se poate face fie la nivel de linie, fie la nivel de caracter, dar numai secvential. Deci nu se pot citi/scrie linii sau caractere decât în ordinea memorării lor în fisier si nu pe sãrite (aleator). Nu se pot face modificări într-un fisier text fãrã a crea un alt fisier, deoarece nu sunt de conceput deplasări de text în fisier.

Pentru citire/scriere din/în fisierele standard stdin/stdout se folosesc functii cu nume putin diferit si cu mai putine argumente, dar se pot folosi si functiile generale destinate fisierelor disc. Urmeazã câteva perechi de functii echivalente ca efect :

```
   // citire caracter
  int fgetc (FILE * f);          // sau getc (FILE*)
  int getchar();                    // echiv. cu  fgetc(stdin)
   // scriere caracter
  int fputc (int c, FILE * f);   // sau putc (int, FILE*)
  int putchar (int c);           // echivalent cu  fputc(c.stdout)
   // citire linie
  char * fgets( char * line, int max, FILE *f);
  char * gets (char * line);
   // scriere linie
  int fputs (char * line, FILE *f);
  int puts (char * line);
```

Functia "fgets" adaugã sirului citit un octet zero (în memorie) dar nu eliminã terminatorul de linie '\n', spre deosebire de functia pereche "gets". Functia "fputs" nu scrie în fisier octetul zero (necesar numai în memorie).

Modul de raportare al ajungerii la sfârsit de fisier este diferit:
- functia "fgetc" are rezultat negativ (constanta EOF= -1)
- functia "fgets" are rezultat NULL (0).

Exemplul urmãtor citeste un fisier text si scrie un alt fisier în care toate literele mari din textul citit sunt transformate în litere mari.

```
  // copiere fisier cu transformare in litere mari
int main (int argc, char * argv[]) {
  FILE * f1, * f2;  int ch;
  f1= fopen (argv[1],"r");   f2= fopen (argv[2],"w");
  if ( f1==0 || f2==0) {
     puts (" Eroare la deschidere fisiere \n");
     return;
  }
  while ( (ch=fgetc(f1)) != EOF)       // citeste  din f1
     fputc ( tolower(ch),f2);          // scrie in f2
  fclose(f1); fclose(f2);
}
```

Detectarea sfârsitului de fisier se poate face si cu ajutorul functiei "feof" (Find End of File), dar cu atentie deoarece rezultatul lui "feof"se modificã dupã încercarea de a citi dupã sfârsitul fisierului. In exemplul urmãtor, se va scrie în fisierul de iesire si –1, care este rezultatul ultimului apel al functiei "fgetc":

```
 while ( ! feof(f1))
    fputc(fgetc(f1),f2);
```

Solutia preferabilã pentru ciclul de citire-scriere caractere este urmãtoarea:

```
 while ( (ch=fgetc(f1)) != EOF)
    fputc ( ch, f2);
```

In principiu se poate citi integral un fisier text în memorie, dar în practicã se citeste o singurã linie sau un numãr de linii succesive, într-un ciclu repetat pânã se terminã fisierul (pentru a se putea prelucra fisiere oricât de mari).

Pentru actualizarea unui fisier text prin modificarea lungimii unor linii, stergerea sau insertia de linii se va scrie un alt fisier si nu se vor opera modificãrile direct pe fisierul existent. Exemplu:

```
  // inlocuieste in f1 sirul s1 prin sirul s2 si scrie rezultat in f2
void subst (FILE * f1, FILE * f2, char * s1, char * s2) {
 char linie[80], *p;
 while ( fgets (linie,80,f1) != NULL) { // citeste o linie din f1
```

```
    while(p=strstr (linie,s1)) {        // p= pozitia lui s1 în txt
      strdel (p,strlen(s1));          // sterge s1 de la adresa p
      strins(p,s2);             // inserez s2 la adresa p
    }
   fputs (linie,f2);           // scrie linia modificata in f2
 }
 fclose(f2);                // sau in afara functiei
}
```

Citire-scriere cu format

Datele numerice pot fi scrise în fisiere disc fie în format intern (mai compact), fie transformate în siruri de caractere (cifre zecimale, semn s.a). Formatul sir de caractere necesitã si caractere separator între numere, ocupã mai mult spatiu dar poate fi citit cu programe scrise în orice limbaj sau cu orice editor de texte sau cu alt program utilitar de vizualizare fisiere.

Functiile de citire-scriere cu conversie de format si editare sunt:

```
int fscanf (FILE * f, char * fmt, ...)
int fprintf (FILE * f, char * fmt, ...)
```

Pentru aceste functii se aplicã toate regulile de la functiile "scanf" si "printf". Un fisier text prelucrat cu functiile "fprintf" si "fscanf" contine mai multe câmpuri de date separate între ele prin unul sau mai multe spatii albe (blanc, tab, linie nouã). Continutul câmpului de date este scris si interpretat la citire conform specificatorului de format pentru acel câmp .

Exemplu de creare si citire fisier de numere.

```
   // creare - citire fisier text ce contine doar numere
int main () {
  FILE * f;  int x;                  // f = pointer la fisier
    // creare fisier de date
  f=fopen ("num.txt","w");        // deschide fisier
  for (x=1;x<=100;x++)
    fprintf (f,"%4d",x);         // scrie un numar
  fclose (f);                     // inchidere fisier
    // citire si afisare fisier creat
  f=fopen ("num.txt","r");
  while (fscanf (f,"%d",&x) == 1)       // pana la sfirsit  fisier
     printf ("%4d",x);                  // afisare numar citit
}
```

Fisiere text cu numere se folosesc pentru fisiere de date initiale cu care se verificã anumite programe, în faza de punere la punct. Rezultatele unui program pot fi puse într-un fisier fie pentru a fi prelucrate de un alt program, fie pentru arhivare sau pentru imprimare repetatã.

De observat ca majoritatea sistemelor de operare permit redirectarea fisierelor standard de intrare si de iesire, fãrã a modifica programele. Deci un program neinteractiv care foloseste functiile "scanf" si "printf" sau alte functii standard (gets, puts, getchar, putchar) poate sã-si citeascã datele dintr-un fisier sau sã scrie rezultatele într-un fisier prin specificarea acestor fisiere în linia de comanda. Exemple de utilizare a unui program de sortare:

```
date de la tastatura, rezultate afisate pe ecran :
        sort
date din "input", rezultate in "output" :
        sort <<input >>output
date de la tastatura, rezultate in "output" “
       sort >>output
date din "input",rezultate afisate pe ecran :
        sort <<input
```

Functii de acces secvential la fisiere binare

Un fisier binar este format în general din articole de lungime fixã, fãrã separatori între articole. Un articol poate contine un singur octet sau un numãr binar (pe 2,4 sau 8 octeti) sau o structurã cu date de diferite tipuri.

Functiile de acces pentru fisiere binare "fread" si "fwrite" pot citi sau scrie unul sau mai multe articole, la fiecare apelare. Transferul între memorie si suportul extern se face fãrã conversie sau editare (adãugare de caractere la scriere sau eliminare de caractere la citire).

Prototipuri functii :

```
size_t fread (void * buf, size_t size, size_t n, FILE* f);
size_t fwrite (void * buf, size_t size, size_t n, FILE* f);
```

De remarcat cã primul argument al functiilor "fread" si "fwrite" este o adresã de memorie (un pointer): adresa unde se citesc date din fisier sau de unde se iau datele scrise în fisier. Al doilea argument este numãrul de octeti pentru un articol, iar al treilea argument este numãrul de articole citite sau scrise. Numãrul de octeti cititi sau scrisi este egal cu produsul dintre lungimea unui articol si numãrul de articole.

Rezultatul functiilor "fread" si "fwrite" este numãrul de articole efectiv citite sau scrise si este diferit de argumentul 3 numai la sfârsit de fisier (la citire) sau în caz de eroare de citire/scriere.

Dacã stim lungimea unui fisier si dacã este loc în memoria RAM atunci putem citi un întreg fisier printr-un singur apel al functiei “fread” sau putem scrie integral un fisier cu un singur apel al functiei “fwrite”. Citirea mai multor date dintr-un fisier disc poate conduce la un timp mai bun fatã de repetarea unor citiri urmate de prelucrãri,

deoarece se pot elimina timpii de asteptare pentru pozitionarea capetelor de citire – scriere pe sectorul ce trebuie citit (rotatie disc plus comanda capetelor).

Programul urmãtor scrie mai multe numere întregi într-un fisier disc (unul câte unul) si apoi citeste continutul fisierului si afiseazã pe ecran numerele citite.

```
int main () {
  FILE * f; int x;  char * numef =“numere.bin”;
     // creare fisier
  f=fopen (numef,"wb");      // din directorul curent
  for (x=1; x<=100; x++)
     fwrite (&x,sizeof(float),1,f);
  fclose(f);
      // citire fisier pentru verificare
  printf("\n");
  f=fopen (numef,"rb");
  while (fread (&x,sizeof(float),1,f)==1)
      printf ("%4d ",x);
  fclose(f);
}
```

Lungimea fisierului "num.bin" este de 200 de octeti, câte 2 octeti pentru fiecare numãr întreg, în timp ce lungimea fisierului "num.txt" creat anterior cu functia "fprintf" este de 400 de octeti (câte 4 caractere ptr fiecare numãr). Pentru alte tipuri de numere diferenta poate fi mult mai mare.

De obicei articolele unui fisier au o anumitã structurã, în sensul cã fiecare articol contine mai multe câmpuri de lungimi si tipuri diferite. Pentru citirea sau scrierea unor astfel de articole în program trebuie sã existe (cel putin) o variabilã structurã care sã reflecte structura articolelor. Dacã se folosesc programe diferite pentru operatii cu un astfel de fisier, atunci este preferabilã descrierea structurii articolelor într-un fisier H.

Exemplu de definire a structurii articolelor unui fisier simplu cu date despre elevi:

```
  // fisier ELEV.H
typedef struct {
  char nume[25];
  float medie;
} Elev;
```

Functiile din exemplul urmãtor scriu sau citesc articole ce corespund unor variabile structurã :

```
#include “elev.h”
  // operatii cu un fisier de elevi (nume si medie)
  // creare fisier cu nume dat
void creare(char * numef) {
  FILE * f;  Elev s;
  f=fopen(numef,"wb");  assert (f != NULL);
  printf (" nume si medie ptr. fiecare student : \n\n");
```

```
  while (scanf ("%s %f ", s.nume, &s.medie) != EOF)
    fwrite(&s,sizeof(s),1,f);
  fclose (f);
}
   // afisare continut fisier pe ecran
void listare (char* numef) {
 FILE * f;  Elev e;
 f=fopen(numef,"rb");  assert (f != NULL);
 while (fread (&e,sizeof(e),1,f)==1)
   printf ("%-25s %6.2f \n",e.nume, e.medie);
 fclose (f);
}
   // adaugare articole la sfarsitul unui fisier existent
void adaugare (char * numef) {
  FILE * f;  Elev e;
  f=fopen(numef,"ab");  assert (f != NULL);
  printf (" nume si medie ptr. fiecare student : \n\n");
  while (scanf ("%s%f ",e.nume, &e.medie) != EOF)
    fwrite(&e,sizeof(e),1,f);
  fclose (f);
}
```

Functii pentru acces direct la date

Accesul direct la date dintr-un fisier este posibil numai pentru un fisier cu articole de lungime fixã si înseamnã posibilitatea de a citi sau scrie oriunde într-un fisier, printr-o pozitionare prealabilã înainte de citire sau scriere. Fisierele mari care necesitã regãsirea rapidã si actualizarea frecventã de articole vor contine numai articole de aceeasi lungime.

In C pozitionarea se face pe un anumit octet din fisier, iar functiile standard permit accesul direct la o anumitã adresã de octet din fisier. Functiile pentru acces direct din <stdio.h> permit operatiile urmãtoare:

- Pozitionarea pe un anumit octet din fisier ("fseek").
- Citirea pozitiei curente din fisier ("ftell").
- Memorarea pozitiei curente si pozitionare ("fgetpos", "fsetpos").

Pozitia curentã în fisier este un numãr de tip *long*, pentru a permite operatii cu fisiere foarte lungi.

Functia "fseek" are prototipul urmãtor :

```
int fseek (FILE * f, long bytes, int origin);
```

unde "bytes" este numãrul de octeti fatã de punctul de referintã "origin", care poate fi: 0 = începutul fisierului, 1 = pozitia curentã, 2 = sfârsitul fisierului.

Functia "fseek" este utilã în urmãtoarele situatii:

- Pentru repozitionare pe început de fisier dupã o cãutare si înainte de o altã cãutare secventialã în fisier (fãrã a închide si a redeschide fisierul)

- Pentru pozitionare pe începutul ultimului articol citit, în vederea scrierii noului continut (modificat) al acestui articol, deoarece orice operatie de citire sau scriere avanseazã automat pozitia curentã în fisier, pe urmãtorul articol.
- Pentru acces direct dupã continutul unui articol (dupã un câmp cheie), dupã ce s-a calculat sau s-a gãsit adresa unui articol cu cheie datã.

Intr-un fisier text pozitionarea este posibilã numai fatã de începutul fisierului, iar pozitia se obtine printr-un apel al functiie “ftell”.

Pozitionarea relativã la sfârsitul unui fisier nu este garantatã nici chiar pentru fisiere binare, astfel cã ar trebui evitatã. De asemenea, ar trebui evitatã pozitionarea fatã de pozitia curentã cu o valoare negativã, care nu functioneazã în toate implementãrile.

Modificarea continutului unui articol (fãrã modificarea lungimii sale) se face în mai multi pasi:
- Se cautã articolul ce trebuie modificat si se retine adresa lui în fisier (înainte sau dupã citirea sa);
- Se modificã în memorie articolul citit;
- Se readuce pozitia curentã pe începutul ultimului articol citit;
- Se scrie articolul modificat, peste continutul sãu anterior.

Exemplu de secventã pentru modificarea unui articol:

```
pos=ftell (f); fread (&e,sizeof(e),1,f );      // pozitia inainte de citire
   . . .   // modifica ceva in variabila “e”
fseek (f,pos,0);                               // repozitionare pe articolul citit
fwrite (&e,sizeof(e),1,f);                        // rescrie ultimul articol citit
```

Exemplu de functie care modificã continutul mai multor articole din fisierul de elevi creat anterior:

```
// modificare continut articole, dupa cautarea lor
void modificare (char * numef) {
 FILE * f; Elev e;  char nume[25];
 long pos;  int  eof;
 f=fopen(numef,"rb+");  assert (f != NULL);
 do {
   printf ("Nume cautat: "); end=scanf ("%s",nume);
   if (strcmp(nume,”.”)==0)  break;    // datele se terminã cu un punct
       // cauta "nume" in fisier
   fseek(f,0,0);                              // readucere pe inceput de fisier
   while ( (eof=fread (&e,sizeof(e),1,f)) ==1 )
     if (strcmp (e.nume, nume)==0) {
       pos= ftell(f)-sizeof(e);
      break;
     }
   if ( eof < 1) break;
   printf ("noua medie: "); scanf ("%f", &e.medie);
   fseek (f,pos,0);                           // pe inceput de articol gasit
```

```
    fwrite(&e,sizeof(e),1,f);                    // rescrie articol modificat
  } while (1);
  fclose (f);
}
```

De observat cã aproape toate programele care lucrează cu un fisier de articole trebuie sã continã o descriere a structurii acestor articole (o definitie de structurã).

"fflush" si alte functii de I/E

Nu orice apel al unei functii de citire sau de scriere are ca efect imediat un transfer de date între exterior si variabilele din program; citirea efectivã de pe suportul extern se face într-o zonã tampon asociatã fisierului, iar numãrul de octeti care se citesc depind de suport: o linie de la tastaturã, unul sau câteva sectoare disc dintr-un fisier disc, etc. Cele mai multe apeluri de functii de I/E au ca efect un transfer între zona tampon (anonimã) si variabilele din program.

Functia "fflush" are rolul de a goli zona tampon folositã de functiile de I/E, zonã altfel inaccesibilã programatorului C. "fflush" are ca argument variabila pointer asociatã unui fisier la deschidere, sau variabilele predefinite "stdin" si "stdout". Exemple de situatii în care este necesarã folosirea functiei "fflush":

a) Citirea unui caracter dupã citirea unui câmp sau unei linii cu "scanf" :

```
int main () {
  int n; char s[30]; char c;
  scanf ("%d",&n);      // sau  scanf("%s",s);
// fflush(stdin);         // pentru corectare
  c= getchar();          // sau  scanf ("%c", &c);
  printf ("%d \n",c);        // afiseaza codul lui c
}
```

Programul anterior va afisa 10 care este codul numeric (zecimal) al caracterului terminator de linie '\n', în loc sã afiseze codul caracterului "c".

Explicatia este aceea cã dupã o citire cu "scanf" în zona tampon rãmân unul sau câteva caractere separator de câmpuri ('\n','\t',' '), care trebuie scoase de acolo prin fflush(stdin) sau prin alte apeluri "scanf". Functia "scanf" opreste citirea unei valori din zona tampon ce contine o linie la primul caracter separator de câmpuri sau la un caracter ilegal în câmp (de ex. literã într-un câmp numeric).

De observat cã dupã citirea unei linii cu functiile "gets" sau "fgets" nu rãmâne nici un caracter în zona tampon si nu este necesar apelul lui "fflush".

b) In cazul repetãrii unei operatii de citire (cu "scanf") dupã o eroare de introducere în linia anterioarã (caracter ilegal pentru un anumit format de citire). Citirea liniei cu eroare se opreste la primul caracter ilegal si în zona tampon rãmân caracterele din linie dupã cel care a produs eroarea. Exemplu:

```
do {
    printf("x,y= ");
    err= scanf("%d%d",&x,&y);
    if (err==2) break;
    fflush(stdin);
  } while (err != 2);
```

c) In cazul scrierii într-un fisier disc, pentru a ne asigura cã toate datele din zona tampon vor fi scrise efectiv pe disc, înainte de închiderea fisierului. In exemplul urmãtor se foloseste “fflush” în loc de “fclose”:

```
int main () {
  FILE * f; int c ; char numef[]="TEST.DAT";
  char x[ ]="0123456789";
  f=fopen (numef,"w");
  for (c=0;c<10;c++)
      fputc (x[c],f);
   fflush(f);   // sau fclose(f);
   f=fopen (numef,"r");
   while ((c=fgetc(f)) != EOF)
      printf ("%c",c);
 }
```

In practicã se va folosi periodic “fflush” în cazul actualizãrii de duratã a unui fisier mare, pentru a evita pierderi de date la producerea unor incidente.

Este posibil ca sã existe diferente în detaliile de lucru ale functiilor standard de citire-scriere din diferite implementãri (biblioteci), deoarece standardul C nu precizeazã toate aceste detalii.

Fisiere din baze de date

Fisierele folosite în bazele de date diferã mult ca structurã de la o implementare la alta, dar au în comun urmãtoarele caracteristici:

- Structura articolelor este memoratã chiar în fisierul de date (la început, un antet);
- Se folosesc fisiere index auxiliare, pentru acces rapid la articole dupã continut.

Ca exemplu vom considera fisierele de tip DBF (DataBase File) folosite de programe mai vechi (dBase, FoxPro) sau mai noi (OpenOffice.org 2.0).

12. Tehnici de programare în C

Stil de programare

Comparând programele scrise de diversi autori în limbajul C se pot constata diferente importante atât în ceea ce priveste modul de redactare al textului sursã (utilizarea de acolade, utilizarea de litere mici si mari, s.a.), cât si în utilizarea elementelor limbajului (instructiuni, declaratii, functii, etc.).

O primã diferentã de abordare este alegerea între a folosi cât mai mult facilitãtile specifice oferite de limbajul C sau de a folosi constructii comune si altor limbaje (Pascal de ex.). Exemple de constructii specifice limbajului C de care se poate abuza sunt:

- Expresii complexe, incluzând prelucrãri, atribuiri si comparatii.
- Utilizarea de operatori specifici: atribuiri combinate cu alte operatii, operatorul conditional, s.a.
- Utilizarea instructiunilor "break" si “return”.
- Utilizarea de pointeri în locul unor vectori sau matrice.
- Utilizarea unor declaratii complexe de tipuri, în loc de a defini tipuri intermediare, mai simple. Exemplu:

```
   // vector de pointeri la functii void f(int,int)
void (*tp[M])(int,int);                    // greu de citit !
   // cu tip intermediar pointer la functie
 typedef void (*funPtr) (int,int);        // pointer la o functie cu 2 argum intregi
 funPtr tp[M];                            // vector cu M elemente de tip funPtr
```

O alegere oarecum echivalentã este între programe sursã cât mai compacte (cu cât mai putine instructiuni si declaratii) si programe cât mai explicite si mai usor de înteles. In general este preferabilã calitatea programelor de a fi usor de citit si de modificat si mai putin lungimea codului sursã si, eventual, lungimea codului obiect generat de compilator. Deci programe cât mai clare si nu programe cât mai scurte.

Exemplu de secventã pentru afisarea a n întregi câte m pe o linie :

```
for ( i=1;i<=n;i++) {
    printf ( "%5d%c",i, ( i%m==0 || i==n)? '\n':' ');
```

O variantã mai explicitã dar mai lungã pentru secventa anterioarã:

```
for ( i=1;i<=n;i++) {
  printf ("%6d ",i);
     if(i%m==0)
        printf("\n");
  }
printf("\n");
```

Conventii de scriere a programelor

Programele sunt destinate calculatorului si sunt analizate de către un program compilator. Acest compilator ignorã spatiile albe nesemnificative si trecerea de la o linie la alta.

Programele sunt citite si de către oameni, fie pentru a fi modificate sau extinse, fie pentru comunicarea unor noi algoritmi sub formã de programe. Pentru a fi mai usor de înteles de către oameni se recomandã folosirea unor conventii de trecere de pe o linie pe alta, de aliniere în cadrul fiecãrei linii, de utilizare a spatiilor albe si a comentariilor.

Respectarea unor conventii de scriere în majoritatea programelor poate contribui la reducerea diversitãtii programelor scrise de diversi autori si deci la facilitarea întelegerii si modificãrii lor de către alti programatori.

O serie de conventii au fost stabilite de autorii limbajului C si ai primului manual de C. De exemplu, numele de variabile si de functii încep cu o literã micã si contin mai mult litere mici (litere mari numai în nume compuse din mai multe cuvinte alãturate, cum sunt nume de functii din MS-Windows). Literele mari se folosesc în nume pentru constante simbolice. In ceea ce priveste numele unor noi tipuri de date pãrerile sunt împãrtite.

Una dintre conventii se referã la modul de scriere a acoladelor care încadreazã un bloc de instructiuni ce face parte dintr-o functie sau dintr-o instructiune *if, while, for* etc. Cele douã stiluri care pot fi întâlnite în diferite programe si cãrti sunt ilustrate de exemplele urmãtoare:

```
   // descompunere in factori primi (stil Kernighan & Ritchie)
int main () {
  int n, k, p ;
  printf ("\n n= "); scanf ("%d",&n);
  printf (“1”);                          // pentru simplificarea afisarii
  for (k=2; k<=n && n>1; k++) {
     p=0;              // puterea lui k in n
     while (n % k ==0) {           // cat timp n se imparte exact prin k
       p++; n=n / k;
     }
     if (p > 0)                          // nu scrie factori la puterea zero
       printf (" * %d^%d",k,p);
   }
}

 // descompunere in factori primi (stil Linux)
int main ()
{
 int n, k, p ;
 printf ("\n n= "); scanf ("%d",&n);
 printf (“1”);          // pentru simplificarea afisarii
 for (k=2; k<=n && n>1; k++)
 {
    p=0;               // puterea lui k in n
```

```
   while (n % k ==0)       // cat timp n se imparte exact prin k
   {
      p++; n=n / k;
   }
   if (p > 0)          // nu scrie factori la puterea zero
      printf (" * %d^%d",k,p);
 }
}
```

Uneori se recomandã utilizare de acolade chiar si pentru o singurã instructiune, anticipând adãugarea altor instructiuni în viitor la blocul respectiv. Exemplu:

```
if (p > 0) {        // scrie numai factori cu putere nenula
    printf (" * %d^%d",k,p);
}
```

Pentru alinierea spre dreapta la fiecare bloc inclus într-o structurã de control se pot folosi caractere Tab (‘\t’) sau spatii, dar evidentierea structurii de blocuri incluse este importantã pentru oamenii care citesc programe.

O serie de recomandări se referã la modul cum trebuie documentate programele folosind comentarii. Astfel fiecare functie C ar trebui precedatã de comentarii ce descriu rolul acelei functii, semnificatia argumentelor functiei, rezultatul functiei pentru terminare normalã si cu eroare, preconditii, plus alte date despre autor, data ultimei modificări, alte functii utilizate sau asemãnãtoare, etc. Preconditiile sunt conditii care trebuie satisfãcute de parametri efectivi primiti de functie (limite, valori interzise, s.a) si care pot fi verificate sau nu de functie. Exemplu:

```
 // Functie de conversie numar întreg pozitiv
 // din binar în sir de caractere ASCII terminat cu zero
 // “value” = numar intreg primit de functie (pozitiv)
 // “string” = adresa unde se pune sirul rezultat
 // “radix” = baza de numeratie (intre 2 si 16, inclusiv)
 // are ca rezultat adresa sir sau NULL in caz de eroare
 // trebuie completata pentru numere cu semn
char *itoa(int value, char *string, int radix) {
 char digits[] = "0123456789ABCDEF";
 char t[20], *tt=t, * s=string;
 if ( radix > 16 || radix < 0 || value < 0) return NULL;
 do {
  *tt++ = digits[ value % radix];
 } while ( (value = value / radix) != 0 );
 while ( tt != t)
   *string++= *(--tt);
 *string=0;
 return s;
}
```

Constructii idiomatice

Reducerea diversitãtii programelor si a timpului de dezvoltare a programelor se poate face prin utilizarea unor idiomuri consacrate de practica programãrii în C.

Cuvintele "idiom", "idiomatic" se referã la particularitãtile unui limbaj iar limbajul C excelează prin astfel de particularitãti.

Constructiile idiomatice în programare sunt denumite uneori sabloane sau tipare ("patterns"), pentru cã ele revin sub diverse forme în majoritatea programelor, indiferent de autorii lor. Folosirea unor constructii idiomatice permite programatorului sã se concentreze mai mult asupra algoritmului problemei si mai putin asupra mijloacelor de exprimare a acestui algoritm.

Un exemplu simplu de constructie idiomaticã C este initializarea la declarare a unei variabile sau a unui vector, în particular initializarea cu zerouri:

```
int counters[1000]= {0} ;
```

Specific limbajului C este utilizarea de expresii aritmetice sau de atribuire drept conditii în instructiuni *if, while, for, do* în absenta unui tip logic (boolean). Exemplu:

```
while (*d++ =*s++);      // copiaza sir de la s la d
```

In standardul C din 1999 s-a introdus un tip boolean, dar nu s-a modificat sintaxa instructiunilor astfel cã se pot folosi în continuare expresii aritmetice drept conditii verificate. Limbajul Java a preluat toate instructiunile din C dar cere ca instructiunile *if, do,...* sã foloseascã expresii logice si nu aritmetice.

Pentru a facilita citirea programelor si trecerea de la C la Java este bine ca toate conditiile sã aparã ca expresii de relatie si nu ca expresii aritmetice:

```
while (*s != 0)
     *d++=*s++;
```

Un exemplu de constructie specificã limbajului C este apelarea unei functii urmatã de verificarea rezultatului functiei, într-o aceeasi instructiune:

```
if ( f = fopen (fname,"r")) == NULL){
   printf ("Eroare la deschidere fisier %s \n", fname);
   exit(-1);
}
```

Utilizarea instructiunii *for* pentru cicluri cu numãrare, cu o ultimã expresie de incrementare, este o constructie tipicã limbajului C. Specific limbajului este si numerotarea de la zero a elementelor unui vector (matrice). Exemplu:

```
for (i=0;i<n;i++)
   printf (“%g “, x[i]);
```

Utilizarea de pointeri pentru prelucrarea sirurilor de caractere, cu incrementare adresei din sir dupã fiecare caracter prelucrat este un alt caz:

```
int strlen ( char * str) {      // lungime sir terminat cu zero
  int len=0;
  while ( *str++)
    len++;
  return len;
}
```

Utilizarea de pointeri ca argumente si rezultat al functiilor pe siruri (strcpy, strcat, s.a.), în functiile de citire-scriere (scanf, gets, read, write) si în functiile generice (qsort, lsearch, bsearch) este o caracteristicã a limbajului C si necesitã stãpânirea utilizãrii corecte a tipurilor pointer (principala sursã de erori în programare).

Un alt exemplu de sablon de programare este citirea unor nume dintr-un fisier de date sau de la consolã, alocarea dinamicã de memorie pentru siruri si memorarea adreselor acestor siruri într-un vector:

```
char buf[80], *a[1000]; int i=0;
while ( (scanf ("%s", buf) > 0))   // citire in zona “buf”
  a[i++]=  strdup( buf);                  // duplicare sir din buf la alta adresa
```

Alocarea dinamicã de memorie în C este o constructie idiomaticã, care foloseste operatorii ***sizeof*** si pentru conversie de tip (“cast”):

```
char** a = (char**) calloc (n, sizeof(char*)); // vector de n pointeri
```

Conversia de tip pentru variabile numerice si variabile pointer printr-un numãr nelimitat de operatori (un operator “cast” pentru fiecare tip) este de asemenea specificã limbajului C.

In scrierea programelor cu interfatã graficã sub Windows se folosesc multe sabloane de cod, unele chiar generate automat de cãtre mediul de dezvoltare.

Portabilitatea programelor

Un program C este portabil atunci când poate fi folosit (“portat”) pe orice calculator si sub orice sistem de operare, fãrã modificarea textului sursã.

Un program este portabil dacã :

- nu foloseste extensii ale standardului limbajului C, specifice unei anumite implementãri a limbajului (unui anumit compilator) si nici elemente de C++.
- nu foloseste functii specifice unui sistem de operare sau unui mediu de dezvoltare (functii nestandard).
- nu foloseste adrese de memorie sau alte particularitãti ale calculatorului.
- nu foloseste particularitãti ale mediului de dezvoltare (o anumitã lungime pentru numere întregi sau pentru pointeri, anumite tipuri de biblioteci etc.).

In general pot fi portabile programele de aplicatii care folosesc numai functii standard pentru intrări-iesiri (printf, scanf s.a.) si pentru alte servicii ale sistemului de operare gazdã (obtinere orã curentã, atribute fisiere etc.).

Programele care folosesc ecranul în mod grafic (cu ferestre, butoane, diverse forme si dimensiuni de caractere etc.) sau care necesitã pozitionarea pe ecran în mod text sunt dependente de sistemul de operare gazdã (Windows, Linux etc.).

Pentru mãrirea portabilitãtii programelor C standardul POSIX (Portable Operating System) propune noi functii unice în C pentru acces la servicii care ar trebui asigurate de orice sistem de operare compatibil POSIX.

Aflarea fisierelor dintr-un director si a atributelor acestora este un exemplu de operatii care depind de sistemul gazdã si nu se exprimã prin functii standard C, desi sunt necesare în multe programe utilitare: listare nume fisiere, arhivare fisiere, s.a. Mai exact, operatiile pot fi exprimate prin una sau douã functii, dar argumentele acestor functii (structuri sau pointeri la structuri) depind de sistemul gazdã.

De exemplu, programul urmãtor pentru afisarea numelor fisierelor dintr-un director este utilizabil numai cu biblioteci GNU “gcc” (cum este si Dev-Cpp) :

```
#include <stdio.h>
#include <dir.h>
int main ( ) {
  struct _finddata_t fb;
  long first;  int  done=0;
  first = _findfirst(path,&fb);
  while (!done) {
    puts (fb.name);
    done = _findnext(first,&fb);
  }
}
```

Acelasi program în varianta mediului BorlandC 3.1 :

```
#include <stdio.h>
#include <dir.h>
void  main ( ) {
   struct ffblk fb;
   char mask[10]="*.*";
   int done;
   done = findfirst(mask,&fb,0xFF);
   while (!done) {
      puts( fb.ff_name);
      done = findnext(&fb);
   }
}
```

Perechea de functii “findfirst”, findnext” realizeazã enumerarea fisierelor dintr-un director (al cãror numãr nu se cunoaste) si constituie elemente ale unui mecanism iterator (enumerator) folosit si în alte situatii de programare.

Erori uzuale în programe C

Majoritatea erorilor de programare provin din faptul cã ceea ce executã calculatorul este diferit de intentiile programatorului. Erorile care se manifestã la executia programelor C au ca efect erori de adresare a memoriei (semnalate de sistemul de operare) sau rezultate gresite. Mesaje de eroare la executie pot fi generate de programator în urma unor verificări prin *assert* sau prin instructiuni *if*.

Descoperirea diferentelor dintre intentiile programatorului si actiunile programului sãu se poate face prin depanarea programului. Depanarea se poate face prin introducerea de instructiuni suplimentare în program în faza de punere la punct (afisări de variabile, verificări cu *assert* s.a.) sau prin folosirea unui program "debugger" care asistã executia.

Existã câteva categorii de erori frecvente:

- Erori de algoritm sau de întelegere gresitã a problemei de rezolvat.
- Erori de exprimare a unui algoritm în limbajul de programare folosit.
- Erori de utilizare a functiilor standard sau specifice aplicatiei.
- Erori de preluare a datelor initiale (de citire date).

Utilizarea de variabile neinitializate este o sursã de erori atunci când compilatorul nu semnaleazã astfel de posibile erori (nu se pot verifica toate situatiile în care o variabilã poate primi o valoare). In particular, utilizarea de variabile pointer neinitializate ca adrese de siruri este o eroare uzualã.

Indirectarea prin variabile pointer cu valoarea NULL sau neinitializate poate produce erori de adresare care sã afecteze si sistemul de operare gazdã.

Erorile la depãsirea memoriei alocate pentru vectori (indici prea mari sau prea mici) nu sunt specifice limbajului C, dar nici nu pot fi detectate la executie decât prin instructiuni de verificare scrise de programator (în Pascal si în Java aceste verificări la indici de vectori se fac automat).

O serie de greseli, care trec de compilare, se datoreazã necunoasterii temeinice a limbajului sau neatentiei; în aceste cazuri limbajul "trãdeazã" intentiile programatorului.

Exemplul cel mai des citat este utilizarea operatorului de atribuire pentru comparatie la egalitate, probabil consecintã a obisnuintelor din alte limbaje:

```
if ( a = b) printf (" a=b \n");        // corect este    if ( a==b ) ...
```

Alte erori sunt cauzate de absenta acoladelor pentru grupuri de instructiuni, de absenta parantezelor în expresii pentru modificarea prioritãtii implicite de calcul, de utilizarea gresitã a tipurilor numerice si atribuirilor.

Operatiile cu siruri de caractere în C pot produce o serie de erori, mai ales cã exprimarea lor este diferitã fatã de alte limbaje: prin functii si nu prin operatori ai limbajului. Functiile pe siruri nu pot face nici o verificare asupra depãsirii memoriei alocate pentru siruri deoarece nu primesc aceastã informatie, ci numai adresele sirurilor.

Definirea si utilizarea de functii

Un program bine scris este o colectie de functii relativ mici, dintre care unele vor fi reutilizate si în alte aplicatii asemãnãtoare.

O functie nu trebuie sã depãseascã cam o paginã de text sursã (cca 50 linii) din mai multe motive: o functie nu trebuie sã realizeze roluri ce pot fi împãrtite între mai multe functii, o functie nu trebuie sã aibã prea multe argumente, o secventã prea lungã de cod sursã este mai greu de stãpânit.

Programele reale totalizeazã sute si mii de linii sursã, deci numãrul de functii din aceste programe va fi mare, iar functiile trebuie sã comunice. Pe de altã parte, transmiterea de rezultate prin argumente pointer în C nu este cea mai simplã si nici cea mai sigurã solutie pentru programatorii începãtori. Solutia argumentelor referintã din C++ a fost preluatã si de unele compilatoare C pentru cã este simplã si sigurã.

Stabilirea functiilor din componenta unui program este parte din activitatea de proiectare (elaborare) ce precede scrierea de cod. Vom schita prin exemple abordarea de sus în jos (‘top-down”) în proiectarea programelor, prin douã exemple nu foarte mari dar nici banale.

Primul exemplu este un preprocesor pentru directive “define” într-o formã mult simplificatã. Acest program citeste un fisier cu un text sursã C care poate contine directive “define” si produce un alt fisier C fãrã directive “define”, dar cu macro-substitutiile efectuate în text. O directivã “define” asociazã unui identificator un alt sir de caractere, care poate include si spatii. Exemplu:

```
#define byte unsigned char
```

Preprocesorul va înlocui în textul sursã care urmeazã acestei directive identificatorul “byte” prin sirul “unsigned char”, indiferent de câte ori apare.

Logica de lucru a programului propus poate fi descrisã astfel:

```
repeta pentru fiecare linie din fisier
    daca este directiva define retine intr-un tabel cele doua siruri
     daca nu este directiva define atunci
         repeta pentru toata linia
           cauta urmatorul identificator
            daca este in tabelul de directive define atunci
                  inlocuieste identificator prin sirul echivalent
```

De remarcat cã algoritmul descris este mult mai eficient decât un algoritm care ar folosi functia de bibliotecã “strstr” pentru a cãuta fiecare identificator în textul sursã, deoarece nu orice sir este un identificator; un identificator este un sir format numai din litere si/sau cifre si care începe obligatoriu cu o literã.

Rezultã ca ar fi utilã definirea unor functii pentru urmãtoarele operatii:

- introducere si respectiv cãutare în tabelul cu identificatori si siruri echivalente (doi vectori de pointeri în implementarea propusã);
- cãutarea urmãtorului identificator începând de la o adresã datã;
- înlocuirea unui subsir cu un alt sir la o adresã datã;

- functie care prelucrează o directivă define, prin extragerea celor două siruri;
- functie care ignorã spatii albe succesive de la o adresã datã.

Urmează implementarea a două dintre aceste functii:

```
  // cauta urmatorul identificator dintr-un sir dat s
  // rezultat adresa imediat urmatoare ; nume depus la "id"
char * nxtid ( char * adr, char * rez) {
  while ( *adr && ! isalpha(*adr) )              // ignora caracterele care nu sunt litere
     ++adr;
  if (*adr==0)                                   // daca sfarsit sir “adr”
     return 0;                                   // 0 daca nu mai sunt identificatori
  *rez++=*adr++;                                 //  prima litera
  while (*adr && (isalpha(*adr)|| isdigit(*adr)) )     // muta in rez litere si cifre
     *rez++=*adr++;
  *rez=0;                                        // terminator sir de la adresa “rez”
  return adr;                                    // adresa urmatoare cuvantului gasit
}
    // inlocuieste n1 caractere la adresa s1 prin n2 caractere de la adresa s2
void subst (char* s1, int n1, char * s2, int n2) {
  char t[256]; int m;
  // sterge n1 caractere de la adresa s1
  m=strlen(s1);
  if (m < n1) n1=m;              // daca sunt mai putin de n1 caractere in s1
  strcpy (s1, s1+n1);            // stergere prin deplasare la stanga caractere
  // introduce la s1 n2 caractere de la adresa s2
  strncpy(t,s2,n2); t[n2]=0;     // copiere n2 octeti de la s2 la t
  strcpy(s1,strcat(t,s1));       // copiere sir din t la s1 dupa concatenare
}
```

Programul principal este destul de lung în acest caz, dar definirea altor functii nu ar aduce avantaje (ele ar avea cam multe argumente sau ar folosi variabile globale):

```
int main () {
  char lin[128], * p, *q;                // in “lin” se citeste o linie
  char * def ="#define";                      // sir care incepe o directiva define
  char s1[128]={0}, s2[128]={0};         // s1 se va inlocui cu s2
  char *tid[20], *ts[20];                // un vector de identificatori si un vector de siruri
  int i,nd=0;                            // numar de definitii (de directive define)
  char id[256];                          // un identificator
  FILE * f;
  f=fopen("test.txt","r");               // un fisier de test
  assert (f 1= NULL);                    // se iese daca nu exista acest fisier
  while ( fgets (lin,128,f) ) {          // citeste o linie din fisier (terminata cu ‘\n’)
    lin[strlen(lin)-1]=0;                // elimina '\n' din sirul s2
    p=skip(p=lin);                           // ignora spatii la inceput de linie
    if (strncmp (def,p,7)==0) {              // daca e o directiva "define"
      pdef (lin,s1,s2);                      // extrage identificator in s1 si al doilea sir in
s2
      tid[nd]=strdup(s1);                // pune s1 in vectorul “tid”
```

```
      ts[nd]=strdup(s2);                  // pune s2 in vectorul “ts”
      nd++;                               // dimensiune vectori tid si ts
    }
    else {                                // daca nu e directiva define
      p=lin;                              // “lin” nu poate apare in stanga unei atribuiri

     while (p=nxtid(p,id)) {                  // extrage urmatorul identificator de la
adresa p
        i=get(tid,nd,id);                 // cauta identificator in vectorul “tid”
        if (i>=0) {                       // daca este identificator define
          strcpy(s2,ts[i]);               // s2 este sirul inlocuitor
          q=p-strlen(id);                 // q este adresa de inceput identificator
          subst(q,strlen(id),s2,strlen(s2)); // efectuare substitutie
        }
      }
      printf("%s\n",lin);                 // afisare linie dupa efectuare inlocuiri
    }
  }
}
```

Dintre avantajele utilizării functiilor pentru operatii distincte din acest program vom mentiona posibilitatea modificării definitiei unui identificator fără a afecta alte functii decât “nxtid”. O directivă “define” poate contine ca identificatori si nume de functii cu argumente si care includ spatii după parantează, ca în exemplul următor:

```
#define max( a,b )   (a > b ? a : b )
```

Un al doilea exemplu este un program care afiseaza functiile apelate dar nedefinite într-un fisier sursă C. Aici trebuie să găsim apelurile si respectiv definitiile de functii dintr-un fisier sursă, memorând numele functiilor definite. Algoritmul este următorul:

```
repeta pentru fiecare linie din fisier
   daca este definitie de functie atunci retine nume functie intr-un vector
    daca este apel de functie atunci
        cauta nume functie in vector de functii definite
        daca nu este in vector atunci
            afisare nume functie (apelata dar nedefinita anterior)
```

Definitiile si apelurile de functii C au în comun începutul - un identificator urmat de o paranteză deschisă ‘(‘ – dar diferă prin ceea ce urmează după paranteza închisă: dacă urmează o acoladă este o definitie, altfel considerăm că este un apel. Presupunem că nu există si declaratii de functii, care ar complica logica programului pentru a separa declaratiile de apeluri.

De observat că trebuie eliminate cuvintele cheie C ce pot fi urmate de paranteze din lista posibilelor nume de functii (“if”, “for”, “while”, “switch”,...).

Functiile care ar fi utile în acest program sunt:

- introducere si respectiv căutare în vectorul cu nume de functii (pointeri la nume);
- căutarea următorului identificator începând de la o adresă dată;

- cãutarea urmãtorului identificator nume de functie;

Cel putin douã dintre aceste functii au fost prezente si în programul anterior si vor fi probabil utile si în alte aplicatii care prelucrează fisiere sursã C.

Urmează functii specifice aplicatiei propuse:

```
// cauta urmatorul nume de functie (apel sau definire)
char* nxtfun (char * adr , char * rez) {
    char * p= adr;
    int i,found=0;
    char * kw[4]={"if","while","for","switch"};
    while (! found && (p= nxtid(p,rez))){
      i=get (kw,4,rez);
      if (i>=0) continue;
      while (isspace(*p))p++;        // ignora spatii albe
      if (*p!='(')
        continue;             // altceva decat nume de functie
      p= strchr(p+1,')')+1;        // gasit nume de functie
       found=1;
    }
    return p;
}
  // listare functii apelate dar nedefinite
void listf (char * s) {
  char * funs[100]; int i,nf=0;          //nume functii definite
  char * p=s;
  char fun[100];                         // un nume de functie
  while ( p= nxtfun(p,fun)){             // p= adresa unui antet de functie
    while (isspace(*p)) p++;             // ignora eventuale spatii
     if (*p=='{')                        // daca definitie
       nf=put(funs,nf,fun);              // pune definitie in tabel
     else {                              // daca apel functie
       i = get(funs,nf,fun);             // cauta in functii deja definite
       if (i<0)
         printf ("%s\n",fun);            // afisare nume functie nedefinita
     }
     p++;                                // peste '{' sau ';'
  }
}
  // verificare
int main (int argc, char* argv[]) {
  char lin[256]; FILE * f;
  assert (argc > 1);
  f= fopen (argv[1],"r");
  assert (f != NULL);
  while ( fgets (lin,256,f))
    listf (lin);
}
```

Luarea în considerare a prototipurilor de functii (declaratii de functii înainte de definire), ar necesita modificări numai în anumite module din program (“listf”) si eventual definirea unor noi functii.

Acesta este si avantajul principal al definirii si utilizării de functii: modificările necesare extinderii sau adaptării unei aplicatii la noi cerinte trebuie să afecteze cât mai putine module din program, pentru a introduce cât mai putine erori si numai în părti previzibile din program.

13. Dezvoltarea programelor mari în C

Particularitãti ale programelor mari

Aplicatiile reale conduc la programe mari, cu mai multe sute si chiar mii de linii sursã. Un astfel de program suferã numeroase modificãri (cel putin în faza de punere la punct), pentru adaptarea la cerintele mereu modificate ale beneficiarilor aplicatiei (pentru îmbunãtãtirea aspectului si modului de utilizare sau pentru extinderea cu noi functii sau pentru corectarea unor erori apãrute în exploatare).

Programarea la scarã mare este diferitã de scrierea unor programe mici, de scoalã, si pune probleme specifice de utilizare a limbajului, a unor tehnici si instrumente de dezvoltare a programelor, de comunicare între programatori si chiar de organizare si coordonare a colectivelor de programatori.

Principala metodã de stãpânire a complexitãtii programelor mari este împãrtirea lor în module relativ mici, cu functii si interfete bine precizate.

Un program mare este format dintr-un numãr oarecare de functii, numãr de ordinul zecilor sau sutelor de functii. Este bine ca aceste functii sã fie grupate în câteva fisiere sursã, astfel ca modificãri ale programului sã se facã prin editarea si recompilarea unui singur fisier sursã (sau a câteva fisiere) si nu a întregului program (se evitã recompilarea unor functii care nu au suferit modificãri). In plus, este posibilã dezvoltarea si testarea în paralel a unor functii din aplicatie de cãtre persoane diferite.

Inainte de a începe scrierea de cod este necesarã de obicei o etapã care contine de obicei urmãtoarele:

- întelegerea specificatiilor problemei de rezolvat si analiza unor produse software asemãnãtoare.
- stabilirea functiilor de bibliotecã care pot fi folosite si verificarea modului de utilizare a lor (pe exemple simple).
- determinarea structurii mari a programului: care sunt principalele functii din componenta programului si care sunt eventualele variabile externe.

Pentru a ilustra o parte din problemele legate de proiectarea si scrierea programelor mari vom folosi ca exemplu un program care sã realizeze efectul comenzii DIR din MS-DOS (“dir” si “ls” din Linux), deci sã afiseze numele si atributele fisierelor dintr-un director dat explicit sau implicit din directorul curent. O parte din aceste probleme sunt comune mai multor programe utilitare folosite în mod uzual.

Pentru început vom defini specificatiile programului, deci toate datele initiale (nume de fisiere si optiuni de afisare), eventual dupã analiza unor programe existente, cu acelasi rol. Programul va fi folosit în mod linie de comandã si va prelua datele necesare din linia de comandã.

O parte din optiunile de afisare au valori implicite; în mod normal se afiseazã toate fisierele din directorul curent, nu se afiseazã fisierele din subdirectoare si nu se afiseazã toate atributele fisierelor ci numai cele mai importante. Exemple de utilizare:

```
dir                    // toate fisierele din directorul curent, cu atribute
```

```
dir c:\work          // toate fisierele din directorul “work”
dir *.c              // toate fisierele de tip c din directorul curent
dir a:\pc lab*.txt   // fisiere de tip txt din a:\pc
dir /B *.obj         // fisiere de tip “obj”, fara atribute
```

Datele necesare programului sunt preluate din linia de comandã si poate fi necesarã includerea între ghilimele a sirului ce descrie calea si tipul fisierelor:

```
dir “c:\lcc\bin\*.*”
```

Programul va contine cel putin trei module principale : preluare date initiale (“input”), obtinere informatii despre fisierele cerute (“getfiles”) si prezentarea acestor informatii (“output”), plus un program principal.

Aceste module pot fi realizate ca fisiere sursã separate, pentru ca eventual sã se poatã face trecerea spre o variantã cu interfatã graficã, cu izolarea modificãrilor necesare acestei treceri si evitarea editării unui singur fisier sursã foarte mare (dacã tot programul se realizează ca un singur fisier).

Compilãri separate si fisiere proiect

Pe lângã aspectele ce tin de limbajul folosit, dezvoltarea si întretinerea programelor mari ridicã si probleme practice, de operare, ce depind de instrumentele software folosite (compilator mod linie de comandã sau mediu integrat IDE) si de sistemul de operare gazdã.

In urma compilării separate a unor fisiere sursã rezultã tot atâtea fisiere obiect (de tip OBJ), care trebuie sã fie legate împreunã într-un singur program executabil. In plus, este posibil ca aplicatia sã folosească biblioteci de functii nestandard, create de alti utilizatori sau create ca parte a aplicatiei.

Bibliotecile de functii sunt de douã categorii distincte:

- Biblioteci cu legare staticã, din care functiile sunt extrase în faza editării de legãturi si sunt atasate programului executabil creat de linkeditor. Diferenta dintre o bibliotecã staticã si un modul obiect este aceea ca un fisier obiect (OBJ) este atasat integral aplicatiei, dar din bibliotecã se extrag si se adaugã aplicatiei numai functiile (modulele obiect) apelate de aplicatie.
- Biblioteci cu legare dinamicã (numite DLL în sistemul Windows), din care functiile sunt extrase în faza de executie a programului, ca urmare a apelării lor. Astfel de biblioteci, folosite în comun de mai multe aplicatii, nu mãresc lungimea programelor de aplicatie, dar trebuie furnizate împreunã cu aplicatia. Un alt avantaj este acela cã o bibliotecã dinamicã poate fi actualizatã (pentru efectuarea de corecturi sau din motive de eficientã) fãrã a repeta construirea aplicatiei care o foloseste (editarea de legãturi). In MS-DOS nu se pot folosi biblioteci cu legare dinamicã.

In legãturã cu compilarea separatã a unor pãrti din programele mari apar douã probleme:

- Enumerarea modulelor obiect si bibliotecilor statice componente.

- Descrierea dependentelor dintre diverse fisiere (surse, obiect, executabile) astfel ca la modificarea unui fisier sã se realizeze automat comenzile necesare pentru actualizarea tuturor fisierelor dependente de cel modificat. Ideea este de gestiune automatã a operatiilor necesare întretinerii unui program mare, din care se modificã numai anumite pãrti.

Pentru dezvoltarea de programe C în mod linie de comandã solutiile celor douã probleme sunt:

- Enumerarea fisierelor obiect si bibliotecilor în comanda de linkeditare.
- Utilizarea unui program de tip "make" si a unor fisiere ce descriu dependente între fisiere si comenzi asociate ("makefile").

Atunci când se foloseste un mediu integrat pentru dezvoltare (IDE) solutia comunã celor douã probleme o constituie fisierele proiect. Desi au cam aceleasi functii si suportã cam aceleasi operatii, fisierele proiect nu au fost unificate si au forme diferite pentru medii IDE de la firme diferite sau din versiuni diferite ale unui IDE de la o aceeasi firmã (de ex. Borland C).

In forma sa cea mai simplã un fisier proiect contine câte o linie pentru fiecare fisier sursã sau obiect sau bibliotecã ce trebuie folosit în producerea unei aplicatii. Exemplu de fisier proiect din Borland C (2.0) :

```
input.c  getfiles.c  output.c  dirlist.c
```

In mediile de programare mai noi proiectele contin mai multe fisiere, unele generate automat de IDE (inclusiv fisierul pentru comanda "make").

Operatiile principale cu un fisier proiect sunt: crearea unui nou proiect, adãugarea sau stergerea unui fisier la un proiect si executia unui fisier proiect. Efectul executiei unui fisier proiect depinde de continutul sãu dar si de data ultimei modificãri a unui fisier din componenta proiectului. Altfel spus, pot exista dependente implicite între fisierele dintr-un proiect:

- Dacã data unui fisier obiect (OBJ) este ulterioarã datei unui fisier executabil, atunci se reface automat operatia de linkeditare, pentru crearea unui nou fisier executabil.
- Dacã data unui fisier sursã (C sau CPP) este ulterioarã datei unui fisier obiect, atunci se recompileazã fisierul sursã într-un nou fisier obiect, ceea ce va antrena si o nouã linkeditare pentru actualizarea programului executabil.

Fisiere antet

Functiile unei aplicatii pot folosi în comun urmãtoarele elemente de limbaj:

- tipuri de date definite de utilizatori (de obicei, tipuri structurã)
- constante simbolice
- variabile externe

Tipurile de date comune se definesc de obicei în fisiere antet (de tip H), care se includ în compilarea fisierelor sursã cu functii (de tip C sau CPP). Tot în aceste fisiere

se definesc constantele simbolice si se declarã functiile folosite în mai multe fisiere din componenta aplicatiei.

Exemplu de fragment dintr-un fisier antet folosit în programul “dirlist”:

```
struct file {
 char fname[13];   // nume fisier (8+3+'.'+0)
 long fsize;     // dimensiune fisier
 char ftime[26] ;  // data ultimei modificari
 short isdir;      // daca fisier director
};
#define MAXC 256   // dimensiunea unor siruri
#define MAXF 1000  // numar de fisiere estimat
```

Fisierul antet “dirlist.h” poate include fisiere antet standard comune (“stdio.h”, ”stdlib.h”), dar este posibil ca includerile de fisiere antet standard sã facã parte din fiecare fisier sursã al aplicatiei.

In general, comunicarea dintre functii se va realiza prin argumente si prin rezultatul asociat numelui functiei si nu prin variabile externe (globale). Existã totusi situatii în care definirea unor variabile externe, folosite de un numãr mare de functii, reduce numãrul de argumente, simplificã utilizarea functiilor si produce un cod mai eficient.

In programul “dirlist” astfel de variabile comune mai multor functii pot fi: calea cãtre directorul indicat, masca de selectie fisiere si lista de optiuni de afisare. Functia “getargs” din fisierul “input.c” preia aceste date din linia de comandã, dar ele sunt folosite de functii din celelalte douã fisiere “getfiles.c” si “output.c”. Variabilele externe se definesc într-unul din fisierele sursã ale aplicatiei, de exemplu în “dirlist.c” care contine functia “main”:

```
char path[MAXC], mask[MAXC], opt[MAXC];  // var comune
```

Domeniul implicit al unei variabile externe este fisierul în care variabila este definitã (mai precis, functiile care urmeazã definitiei). Pentru ca functii din fisiere sursã diferite sã se poatã referi la o aceeasi variabilã, definitã într-un singur fisier este necesarã declararea variabilei respective cu atributul ***extern***, în toate fisierele unde se fac referiri la ea. Exemplu :

```
extern char path[MAXC], mask[MAXC], opt[MAXC];
```

Directive preprocesor utile în programele mari

Directivele preprocesor C au o sintaxã si o prelucrare distinctã de instructiunile si declaratiile limbajului, dar sunt parte a standardului limbajului C. Directivele sunt interpretate într-o etapã preliminarã compilãrii (traducerii) textului C, de un preprocesor.

O directivã începe prin caracterul ‘#’ si se terminã la sfârsitul liniei curente (daca nu existã linii de continuare a liniei curente). Nu se foloseste caracterul ‘;’ pentru terminarea unei directive.

Cele mai importante directive preprocesor sunt :

```
  // inlocuieste toate aparitiile identificatorului “ident” prin sirul “text”
#define ident  text
  // defineste o macroinstructiune cu argumente
#define ident (a1,a2,...) text
  // include in compilare continutul fisierului sursa “fisier”
#include “fisier”
  // compilare conditionata de valoarea expresiei “expr”
#if  expr
  // compilare conditionata de definirea unui identificator (cu #define)
#if defined  ident
  // terminarea unui bloc introdus prin directiva #if
#endif
```

Directiva *define* are multiple utilizări în programele C :

a) - Definirea de constante simbolice de diferite tipuri (numerice, text)

b) - Definirea de macrouri cu aspect de functie, pentru compilarea mai eficientã a unor functii mici, apelate în mod repetat. Exemple:

```
# define max(A,B)  ( (A)>(B) ? (A):(B) )
#define random(num)(int) (((long)rand()*(num))/(RAND_MAX+1))
#define randomize()     srand((unsigned)time(NULL))
```

Macrourile pot contine si declaratii, se pot extinde pe mai multe linii si pot fi utile în reducerea lungimii programelor sursã si a efortului de programare.

In standardul din 1999 al limbajului C s-a preluat din C++ cuvântul cheie *inline* pentru declararea functiilor care vor fi compilate ca macroinstructiuni în loc de a folosi macrouri definite cu *define*.

c)- Definirea unor identificatori specifici fiecãrui fisier si care vor fi testati cu directiva *ifdef*. De exemplu, pentru a evita declaratiile *extern* în toate fisierele sursã, mai putin fisierul ce contine definitiile variabilelor externe, putem proceda astfel:

- Se defineste în fisierul sursã cu definitiile variabilelor externe un nume simbolic oarecare:

```
  // fisierul DIRLIST.C
#define MAIN
```

- In fisierul “dirlist.h” se plaseazã toate declaratiile de variabile externe, dar încadrate de directivele *if* si *endif*:

```
  // fisierul DIRLIST.H
#if !defined(MAIN)       // sau ifndef MAIN
 extern char path[MAXC], mask[MAXC], opt[MAXC];
#endif
```

Directiva *include* este urmatã de obicei de numele unui fisier antet (de tip H = header), fisier care grupeazã declaratii de tipuri, de constante, de functii si de variabile, necesare în mai multe fisiere sursã (C sau CPP). Fisierele antet nu ar trebui sã continã definitii de variabile sau de functii, pentru cã pot apare erori la includerea multiplã a unui fisier antet. Un fisier antet poate include alte fisiere antet.

Pentru a evita includerea multiplã a unui fisier antet (standard sau nestandard) se recomandã ca fiecare fisier antet sã înceapã cu o secventã de felul urmãtor:

```
#ifndef  HDR
#define HDR
  // continut fisier HDR.H ...
#endif
```

Fisierele antet standard ("stdio.h" s.a.) respectã aceastã recomandare.

O solutie alternativã este ca în fisierul ce face includerea sã avem o secventã de forma urmãtoare:

```
#ifndef STDIO_H
#include <stdio.h>
#define _STDIO_H
#endif
```

Directivele de compilare conditionatã de forma *if...endif* au si ele mai multe utilizãri ce pot fi rezumate la adaptarea codului sursã la diferite conditii specifice, cum ar fi:

- dependenta de modelul de memorie folosit (în sistemul MS-DOS)
- dependenta de sistemul de operare sub care se foloseste programul (de ex., anumite functii sau structuri de date care au forme diferite în sisteme diferite)
- dependenta de fisierul sursã în care se aflã (de exemplu "tcalc.h").

Directivele din grupul *if* au mai multe forme, iar un bloc *if ... endif* poate contine si o directiva *elseif*.

Proiectul initial

Majoritatea produselor software se pretează la dezvoltarea lor treptatã, pornind de la o versiune minimalã initialã, extinsã treptat cu noi functii. Prima formã, numitã si prototip, trebuie sã includã partea de interfatã cu utilizatorul final, pentru a putea fi prezentatã repede beneficiarilor, care sã-si precizeze cât mai devreme cerintele privind interfata cu operatorii aplicatiei.

Dezvoltarea în etape înseamnã însã si definirea progresivã a functiilor din componenta aplicatiei, fie de sus în jos ("top-down"), fie de jos în sus ("bottom-up"), fie combinat. Abordarea de sus în jos stabileste functiile importante si programul principal care apeleazã aceste functii. Dupã aceea se defineste fiecare functie, folosind eventual alte functii încã nedefinite, dar care vor fi scrise ulterior. In varianta initialã programul principal aratã astfel :

```
void  main(int argc, char * argv[]) {
   char  *files[MAXF];    // vector cu nume de fisiere
   int nf;                // numar de fisiere
   getargs (argc,argv);   // preluare date
   nf=listFiles(files);   // creare vector de fisiere
   printFiles(files,nf);  // afisare cu atribute
}
```

Abordarea de jos în sus porneste cu definirea unor functii mici, care vor fi apoi apelate în alte functii, s.a.m.d. pânã se ajunge la programul principal.

Pentru aflarea fisierelor de un anumit tip dintr-un director dat se pot folosi functiile nestandard “findffirst” si “findnext”, care depind de implementare.

Pentru determinarea atributelor unui fisier cu nume dat se poate folosi functia “stat” (file status) sau “fstat”, declarate în fisierul antet <sys/stat.h> împreunã cu tipul structurã folosit de functie (“struct stat”). Structura contine dimensiunea fisierului (“st_size”), data de creare (“st_ctime”), data ultimei modificãri si doi octeti cu atributele fisierului (“st_mode”): fisier normal sau director, dacã poate fi scris (sters) sau nu etc. Anumite atribute depind de sistemul de operare gazdã si pot lipsi în alte sisteme, dar functia “stat” si structura “stat” sunt aceleasi pentru diverse implementãri. Pentru determinarea atributelor, fisierul trebuie mai întâi deschis. Prototip “stat” :

```
int stat (char * filename, struct stat * statptr);
```

cu rezultat 0 dacã fisierul specificat în “filename” este gãsit si 1 dacã negãsit.

Functia “stat” poate sã primeascã numele complet, cu cale, al fisierului aflat într-un alt director decât programul care se executã.

Pentru extragerea unor biti din câmpul “st_mode” sunt prevãzute constante simbolice cu nume sugestive. Exemplu:

```
// verificã dacã “file” este fisier normal sau director
 err=stat (file, &finfo);              // pune atribute in finfo
 if (finfo.st_mode & S_IFDIR)
   printf ("Directory \n" );
 else
  printf ("Regular file \n" );
```

Functia “stat” si structura “stat” se pot folosi la fel în mai multe implementãri, desi nu sunt standardizate in ANSI C.

Pentru conversia datei si orei de creare a unui fisier (un numãr *long*) în caractere se foloseste una din functiile standard “ctime” sau “asctime”.

Utilizarea acestor functii necesitã includerea unor fisiere antet:

```
#include <io.h>              // “findfirst”, “findnext”
#include <sys/stat.h>        //  “stat”
#include <time.h>            //  “ctime”
```

```
#include <string.h>          // “strcpy”, “strcat”, strcmp” s.a.
```

Primul modul din programul nostru va fi modulul de preluare a datelor initiale: nume fisier director al cãrui continut se afiseazã (cu calea la director), nume/tip fisiere listate si optiuni de afisare. Aici se fac si verificări asupra utilizării corecte a programului si alte operatii de pregãtire a datelor pentru modulele urmãtoare. Vom porni cu o variantã în care nu se admit optiuni si se afiseazã numai fisiere din directorul curent, specificate printr-o mascã ce poate contine caractere ‘*’ si/sau ‘?’. Deci comanda de lansare a programului poate contine un singur argument (un sir mascã) sau nici unul; dacã nu se dã nici un argument se considerã masca “*.*”, deci se afiseazã toate fisierele.

Varianta initialã pentru primul modul poate fi urmãtoarea:

```
 // preluare argumente din linia de comanda
void getargs (int argc,char *argv[]) {
  char *p;
  if (argc < 2){                        // daca nu exista argument
    strcpy(mask,"*.*");  return;
  }
  p = strrchr(argv[1],'\\');            // ultimul caracter \
  if (p==0)
    strcpy(mask,argv[1]);
  else {
    printf("Numai fisiere din acest director \n");    exit(2);
  }
}
```

Urmãtorul modul, si cel mai important, este cel care obtine din sistem informatiile necesare pentru afisare: lista de fisiere si atributele fiecãrui fisier.
Varianta urmãtoare este pentru mediul Borland C:

```
int listFiles ( char* files[]) {
   struct ffblk finfo;
   int n, err;  char full[256];
   n=0;                                 // numar de fisiere gasite
   strcpy(full,path); strcat(full,mask);
   err= findfirst(full,&finfo,0xff);
   while (err >=0 ) {
       files[n++]= strdup(finfo.ff_name);
     err = findnext(&finfo);
   }
   return n;
}
```

Ultimul modul este cel care se ocupã de prezentarea listei de fisiere în functie de optiunile explicite sau implicite. In varianta initialã se afiseazã numele, lungimea si data de creare a fiecãrui fisier, cu exceptia fisierelor director pentru care nu se poate

obtine simplu dimensiunea totalã. La sfârsitul listei se afiseazã numãrul total de fisiere si dimensiunea lor totalã.

```
  // afisare lista fisiere
void printFiles ( char * f[], int nf) {
  long size, tsize=0L;       // dimensiune totala fisiere
  int i; FILE* fp; short isdir;
  struct stat fst;
  char tim[26], full[256];
  printf ("\n\n");
    // listare completa, cu dimensiune totala
  for (i=0;i<nf;i++) {
    strcpy(full,path); strcat(full,f[i]);
    fp=fopen(full,"r");
    stat (full, &fst);
    size= fst.st_size;   // dimensiune fisier
    tsize += size;
    isdir = fst.st_mode & S_IFDIR;
    strcpy(tim,ctime(&fst.st_ctime));
    tim[strlen(tim)-1]=0;
    if ( isdir)
       printf("%-12s <DIR>\t\t%s \n", f[i],tim );
    else
       printf("%-12s %8ld %s \n", f[i],size,tim);
  }
   printf ("\t%d Files \t %ld bytes \n", nf, tsize);
}
```

Formatul de afisare este apropiat de cel al comenzii DIR din MS-DOS dar nu identic, din cauza folosirii functiei “ctime” si a altor simplificări.

Extinderea programului

Programul nostru poate fi extins treptat, prin adãugarea de noi optiuni de afisare, fãrã modificări esentiale în versiunile precedente ale programului.

Preluarea optiunilor din linia de comandã poate fi relativ simplã dacã vom considera cã fiecare optiune este un sir separat, care începe cu ‘/’ (de obicei se admite gruparea mai multor optiuni într-un sir precedat de ‘/’). Optiunile pot fi scrise în orice ordine, înainte si/sau dupã numele directorului si mascã:

```
dirlist /B c:\games\*.* /OS
```

Optiunile comenzii DIR pot avea una sau douã litere, dar numãrul de litere nu conteazã dacã fiecare optiune se terminã cu spatiu alb.

Rezultatul prelucrării optiunilor din linia de comandã va fi un sir în care literele ce denumesc fiecare optiune sunt separate între ele printr-un caracter /.

```
void getargs (int argc, char *argv[] ) {
  char *p; char f[80];
  int i;
  opt[0]=0;
  if (argc <2){
    strcpy(mask,"*.*"); strcpy(path,".\\");
    return;
  }
  for (i=0;i<argc;i++){
    strcpy(f,argv[i]);   // numai ptr simplificare cod
    if (f[0]=='/') {     // daca optiune
      strcat(opt,f);  continue;
    }
   // argument care nu e optiune
    p = strrchr(f,'\\');
    if (p) {       // daca contine nume director
      strncpy(path,f, p-f+1);path[p-f+1]=0;
      strcpy(mask,p+1);
    }
    else {            // daca nu contine nume director
     strcpy(mask,f); strcpy(path,".\\");
    }
  }
}
```

Verificarea existentei unei optiuni se reduce la cãutarea sirului ce codificã optiunea în sirul “opt” care reuneste toate optiunile. Exemplu:

```
if (strstr (opt,”/b”)||strstr(opt,”/B”)) ...
```

Interpretarea unei optiuni poate fi mai simplã sau mai complicatã, functie de tipul optiunii. Optiunea /B (“brief”) este cea mai usor de tratat si o vom da ca exemplu. In ciclul principal din functia “printFiles” se va insera secventa urmãtoare:

```
    if (strstr(opt,"b")){
       // nu se afiseaza numele “.” si “..”
      if (strcmp(f[i],".")&& strcmp(f[i],".."))
         printf("%-12s \n", f[i]); // doar numele
     continue;      // urmatorul fisier din lista
    }
```

Pentru ordonarea listei de fisiere dupã un atribut (nume, extensie, mãrime, datã) este necesarã memorarea acestor atribute pentru toate fisierele. In acest scop este utilã definirea unei structuri mai mici ca structura “stat” care sã reuneascã numai atributele necesare la ordonare:

```
struct file {
```

```
 char fname[13];    // nume fisier redus la primele 8 car.
 long fsize;       // dimensiune fisier
 char ftime[26] ;    // data ultimei modificari
 char isdir;       // daca fisier director sau ordinar
};
```

Vom scrie o functie care sã determine atributele fisierelor si sã le memoreze într-un vector de structuri de tip “struct file”:

```
  // creare vector cu atribute fisiere
void fileAttr (char * files[], int nf, struct file fat[]) {
   struct stat fstat;
   FILE * fp;
   int i; char * p, *f, full[MAXC];
   for (i=0;i<nf;i++) {
     f=files[i];  // ptr simplificarea expresiilor
     strcpy(full,path); strcat(full,f);
     fp=fopen(full,"r");
     stat (full, &fstat);
     fat[i].isdir = fstat.st_mode & S_IFDIR;
     strcpy(fat[i].ftime, ctime (&fstat.st_ctime));
     fat[i].ftime[strlen(fat[i].ftime)-1]=0;
     if ( strcmp(f,".")==0 || strcmp(f,"..")==0) {
       strcpy(fat[i].fname,f);
       continue;
     }
     fat[i].fsize = fstat.st_size;  // dimensiune fisier
     strcpy (fat[i].fname, f);      // nume fisier
   }
}
```

Functia de afisare “printFiles” va primi acum vectorul de structuri “file” si dimensiunea sa si va suferi unele modificãri.

Vectorul de structuri va fi alocat în functia “main”, cu dimensiune fixã sau dinamic, deoarece se cunoaste acum numãrul exact de fisiere din director.

Modificãrile din functia “main” pentru apelul functiilor vor fi minore.

Ordonarea vectorului de structuri dupã orice câmp al structurilor este simplã dacã se foloseste functia de bibliotecã “qsort”. Pentru fiecare criteriu de sortare este necesarã o functie de comparare (cu prototip impus). Ca exemplu urmeazã douã astfel de functii si utilizarea lor în qsort:

```
  // comparare dupa nume
int cmpext(const void* a, const void * b) {
  struct file * af =(struct file*)a;
  struct file * bf =(struct file*)b;
  return strcmp(af→fname,bf→fname);
}
 // comparare dupa lungime
```

```
int cmpsize(const void* a, const void * b) {
  struct file * af =(struct file*)a;
  struct file * bf =(struct file*)b;
  return (int)(af→fsize - bf→fsize);
}
  // ordonare lista fisiere dupa lungime
void sortBySize (struct file f[], int nf) {
  qsort ( f, nf, sizeof(struct file), cmpsize);
}
```

Pentru ordonare dupã tipul fisierelor trebuie separatã extensia de nume.

Cel mai dificil de realizat este optiunea de afisarea recursivã a fisierelor din subdirectoarele directorului dat, deoarece necesitã eliminarea variabilei externe “path” si introducerea ei ca argument în functia recursivã “printFiles” si în celelalte functii care o folosesc : getargs si listFiles.

Imbunãtãtirea programului

Un program corect si complet poate fi perfectionat pentru:

- Reducerea posibilitãtilor de terminare anormalã, fãrã mesaje explicite.
- Reducerea timpului de executie si a memoriei ocupate.
- Imbunãtãtirea modului de prezentare a rezultatelor.
- Facilitarea unor extinderi sau modificãri ulterioare
- Facilitarea reutilizãrii unor pãrti din program în alte aplicatii.

In versiunea finalã a programului trebuie prevãzute toate situatiile în care ar putea apare erori si mesaje corespunzãtoare. Nu am verificat dacã programul primeste optiuni care nu au sens pentru el, nu am verificat existenta fisierelor la deschidere cu “fopen” sau la apelarea functiei “stat”. In general, fiecare apel de functie trebuie urmat imediat de verificarea rezultatului ei. Exemplu:

```
  if ( (fp=fopen(full,"r")) ==NULL){
    printf(“ Eroare la fopen: fisier %s”,full);
    exit(-1);
  }
  if (stat (full, &fstat)!= 0)
    printf (“ Eroare la functia stat: fisier %s”,full);
    exit (-1);
  }
```

Vectorul de pointeri la nume de fisiere are o dimensiune fixã MAXF, aleasã arbitrar si care ar putea sã fie insuficientã uneori. O solutie mai bunã este o alocare dinamicã initialã de memorie si modificarea functiei “listFiles” pentru extindere automatã prin realocare dinamicã:

```
   char  **files= (char**) malloc(MAXFILES*sizeof(char*));
```

Numãrul total de fisiere din directorul curent si din subdirectoare sale poate fi foarte mare, iar programul trebuie sã facã fatã oricãrui numãr.

In program existã si alte limite (la siruri de caractere) iar încadrarea în aceste limite trebuie verificatã sau se recurge la alocare si realocare dinamicã pentru eliminarea unor limitãri arbitrare.

Comparând cu modul de afisare realizat de comanda DIR programul nostru necesitã mai multe modificãri:

- Numãrul de octeti ocupat de un fisier si de toate fisierele poate avea multe cifre iar pentru a fi mai usor de citit trebuie separate grupe de câte 3 cifre prin virgule. Exemplu: 12,345,678 bytes.

Functia urmãtoare transformã un numãr lung într-un astfel de sir:

```
void format(long x, char * sx) {
  int r[10],i=0; char aux[4];
  *sx=0;              // pregatire strcat(sx,...)
  while ( x > 0) {
     r[++i]=x%1000; // un numar de max 3 cifre
     x=x/1000;
 }
 while ( i >0){
   printf("%d\n",r[i]);
   sprintf(aux,"%d",r[i--]);
   strcat(sx,aux); strcat(sx,",");
 }
 sx[strlen(sx)-1]=0;  // elimina ultima virgula
}
```

- Sirul furnizat de functia “ctime” este greu de citit si contine date inutile (ex. numele zilei din sãptãmânã), deci mai trebuie prelucrat într-o functie.

- In sistemul MS-Windows numele de fisiere nu sunt limitate la 8+3 ca în MS-DOS si deci va trebui prelucrat pentru reducere la 12 caractere. Programul NC (Norton Commander) nu retine primele 8 caractere din nume (care pot fi identice pentru mai multe nume) si formeazã un nume din primele 6 caractere ale numelui complet, caracterul ‘~’ si o cifrã (1,2,3...). Comanda DIR afiseazã si acest nume prescurtat si numele complet (sau o parte din el).

Functiile “findfirst” si “findnext” specifice sistemului MS-DOS fac automat aceastã reducere a numelui, dar alte functii nu o fac si trebuie realizatã în programul de listare.

O parte din functiile programului “dirlist” pot fi reutilizate si în alte programe: preluare optiuni si nume fisiere din linia de comandã, afisare numere întregi foarte mari s.a.

Concluzii

Un program complet pentru comanda DIR este mult mai mare decât schita de program prezentatã anterior, dar este mult mai mic si mai simplu decât alte programe necesare în practicã.

Problemele ridicate de acest program sunt oarecum tipice pentru multe alte programe reale si permite urmãtoarele concluzii:

- Necesitatea stãpânirii tuturor aspectelor limbajului folosit : operatii cu siruri de caractere, cu structuri si vectori de structuri, cu fisiere, alocare dinamicã, transmiterea de date între functii, scrierea de functii recursive etc.
- Necesitatea cunoasterii, cel putin la nivel de inventar, a functiilor disponibile în biblioteci si exersarea lor separatã, înainte de a fi folosite într-un program mare.
- Dezvoltarea progresivã a programelor, cu teste cât mai complete în fiecare etapã. Este bine sã pãstrãm mereu versiunile corecte anterioare, chiar incomplete, pentru a putea reveni la ele dacã prin extindere se introduc erori sau se dovedeste cã solutia de extindere nu a fost cea mai bunã.
- Activitatea de programare necesitã multã atentie si concentrare precum si stãpânirea detaliilor, mai ales într-un limbaj cum este limbajul C. La orice pas trebuie avute în vedere toate posibilitãtile existente si tratate.
- Comentarea rolului unor variabile sau instructiuni se va face chiar la scrierea lor în program si nu ulterior. Numãrul acestor comentarii va fi mult mai mare decât cel din exemplul prezentat, mai ales la fiecare antet de functie.

Aceste comentarii pot facilita adaptarea programului pentru un alt sistem de operare sau pentru o altã interfatã cu utilizatorii programului.

Informatii complete asupra functiilor de bibliotecã pot fi obtinute prin ajutor (Help) oferit de orice mediu IDE sau prin examinarea fisierelor antet, de tip H.

14. Programare generică în C

Colectii de date generice

O colectie de date (numită si structură de date) grupează mai multe componente, numite si elemente ale colectiei. Componentele unei colectii sunt fie valori individuale (numere, siruri de caractere, sau alte tipuri de date), fie perechi cheie-valoare, fie alte colectii sau referinte (pointeri) la date sau la colectii.

O multime poate contine valori numerice de diferite tipuri si lungimi sau siruri de caractere sau alte tipuri de date agregat (structuri), sau pointeri (adrese). Ideal ar fi ca operatiile cu un anumit tip de colectie să poată fi scrise ca functii generale, adaptabile pentru fiecare tip de date ce va face parte din colectie. Acest obiectiv este de dorit mai ales pentru operatii care necesită algoritmi mai complicati (operatii cu arbori binari echilibrati sau cu tabele de dispersie, de ex.), pentru a evita rescrierea functiilor pentru fiecare nou tip de date folosit.

Realizarea unei colectii generice în C se poate face în două moduri, dar nici unul complet satisfăcător:

- Prin utilizarea de tipuri generice (neprecizate) pentru elementele colectiei în subprogramele ce realizează operatii cu colectia. La utilizarea acestor subprograme adaptarea lor la un tip precis, cerut de o aplicatie, se face partial de către compilator (prin macro-substitutie) si partial de către programator (care trebuie să dispună de forma sursă pentru aceste subprograme).
- Prin utilizarea unor colectii de pointeri la un tip neprecizat (*void ** în C) si a unor argumente de acest tip în subprograme, urmând ca înlocuirea cu un alt tip de pointer (la date specifice aplicatiei) să se facă la executie. Utilizarea unor astfel de subprograme este mai dificilă, dar utilizatorul nu trebuie să intervină în textul sursă al subprogramelor.

Utilizarea de tipuri neprecizate

Primul exemplu arată cum se defineste o multime vector cu componente de un tip neprecizat în subprograme, dar precizat în programul care foloseste multimea :

```
    // multimi de elemente de tipul T
typedef  int T;              // tip componente multime
typedef  struct {
   T m[M];             // multime de intregi
   int n;              // dimensiune multime
} Set;
    // operatii cu o multime
int findS ( Set a, T x) {      // cauta pe x in multimea a
  int j=0;
  while ( j < a.n  &&  x != a.m[j] )
     ++j;
```

```
   if ( j==a.n) return 0;       // negasit
     else  return 1;         // gasit
}
int addS ( Set* pa, T x) {  // adauga pe x la multimea a
  if ( findS (*pa,x) )
      return 0;           // nu s-a modificat multimea a
  pa→m[pa→n++] = x;
  return 1;               // s-a modificat multimea a
}
```

Operatiile de citire-scriere a unor elemente din multime depind de asemenea de tipul T, dar ele fac parte în general din programul de aplicatie.

Functiile anterioare sunt corecte numai dacã tipul T este un tip numeric (aritmetic) pentru cã operatiile de comparare la egalitate si de atribuire depind în general de tipul T. Pentru a scrie operatii cu colectii care sã fie valabile pentru orice tip T avem mai multe posibilitãti:

a) Definirea unor operatori generalizati, modificati prin macro-substitutie :

```
#define EQ(a,b)  (a == b) // equals
#define LT(a,b)  (a < b)    // less than
#define AT(a,b)  (a = b)    // assign to
int findS ( Set a, T x) {// cauta pe x in multimea a
   int j=0;
   while ( j < a.n  &&  ! EQ(x,a.m[j]) )
      ++j;
   if ( j==a.n) return 0;       // negasit
     else  return 1;              // gasit
}
int addS (Set* pa, T x) {     // adauga pe x la o multime
  if ( findS (*pa,x) )
     return 0;              // nu s-a modificat multimea
  AT(pa→m[pa→n++],x);    // adaugare x la multime
  return 1;                 // s-a modificat multimea
}
```

Pentru o multime de siruri de caractere trebuie operate urmãtoarele modificãri în secventele anterioare :

```
#define EQ(a,b)  ( strcmp(a,b)==0) // equals
#define LT(a,b)  (strcmp(a,b) < 0)   // less than
#define AT(a,b)  ( strcpy(a,b) )        // assign to
typedef char * T;
```

b) Utilizarea unor functii de comparatie cu nume predefinite, care vor fi rescrise în functie de tipul T al elementelor multimii. Exemplu:

```
typedef  char * T;
```

```
    // comparare la egalitate siruri de caractere
int  comp  (T a, T b ) {
   return strcmp (a,b);
}
int findS ( Set a, T x) {// cauta pe x in multimea a
  int j=0;
  while ( j < a.n  &&  comp(x,a.m[j]) ==0 )
     ++j;
  if ( j==a.n) return 0;            // negasit
    else  return 1;                 // gasit
}
```

c) Transmiterea functiilor de comparare, atribuire, s.a ca argumente la functiile care le folosesc (fãrã a impune nume fixe acestor functii), la fel ca la apelul functiei "qsort". Exemplu:

```
typedef  char * T;                // definire tip T
  // tip functie de comparare
typedef  (int *) Fcmp ( T a, T b) ;
   // cauta pe x in multimea a
int findS ( Set a, T x, Fcmp cmp ) {
  int j=0;
  while ( j < a.n  &&  cmp(x,a.m[j]) ==0 )
     ++j;
  if ( j==a.n) return 0;              // negasit
    else  return 1;                   // gasit
}
```

Uneori tipul T al datelor folosite de o aplicatie este un tip agregat (o structurã C): o datã calendaristicã ce grupeazã numere pentru zi, lunã, an , descrierea unui arc dintr-un graf pentru care se memoreazã numerele nodurilor si costul arcului, s.a. Problema care se pune este dacã tipul T este chiar tipul structurã sau este un tip pointer la acea structurã. Ca si în cazul sirurilor de caractere este preferabil sã se lucreze cu pointeri (cu adrese de structuri) si nu structuri. In plus, atribuirea între pointeri se face la fel ca si atribuirea între numere (folosind operatorul de atribuire). Obiectele nu se mutã în memorie, ci doar adresele lor se mutã dintr-o colectie în alta.

In concluzie, tipul neprecizat T al elementelor unei colectii este de obicei fie un tip numeric, fie un tip pointer (inclusiv de tip *void* *).

Utilizarea de pointeri la "void"

O a doua solutie pentru o colectie generică este o colectie de pointeri la orice tip (*void* *), care vor fi înlocuiti cu pointeri la datele folosite în fiecare aplicatie. Si în acest caz functia de comparare trebuie transmisã ca argument functiilor de inserare sau de cãutare în colectie. Avantajul asupra solutiei cu tip neprecizat T este acela cã functiile pentru operatii cu colectii pot fi compilate si puse într-o bibliotecã si nu este necesar

codul sursã. Exemplu de operatii cu o multime de pointeri:

```
   // Multime ca vector de pointeri
// tipul multime
#define M 100
typedef void* Ptr;
typedef int (*Fcmp) (Ptr,Ptr) ;
typedef struct  {
   Ptr v[M];                    // un vector de pointeri la elementele multimii
   int n;                       // nr elem in multime
} * Set;
     // afisare date din multime
void printS ( Set a) {
  void print ( Ptr);    // declara functia apelata
  int i;
  for(i=0;i<a→n;i++)
       print (a→v[i]);  // depinde de tipul argumentului
  printf ("\n");
}
   // cautare in multime
int findS ( Set a, Ptr p, Fcmp comp ) {
  int i;
  for (i=0;i<a→n;i++)
    if (comp(p,a→v[i])==0)
      return 1;
  return 0;
}
   // adaugare la multime
int addS ( Set a, Ptr p, Fcmp comp) {
  if ( findS(a,p,comp))
    return 0;                   // multime nemodificata
  a→v[a→n++] = p;               // adaugare la multime
  return 1;                     // multime modificata
}
  // initializare multime
void initS (Set a) {
  a→n=0;
}
```

Dezavantajul unor colectii de pointeri apare în aplicatiile numerice: pentru fiecare numãr trebuie alocatã memorie la executie ca sã obtinem o adresã distinctã ce se memoreazã în colectie. Fiecare bloc de memorie alocat dinamic are un antet cu lungimea blocului (8 octeti în Borland C). Consumul de memorie este deci cu mult mai mare decât în cazul unui vector cu date de tip neprecizat. Exemplu de creare si afisare a unei multimi de întregi:

```
 // utilizare multime de pointeri
   // afisare numar intreg
```

```
void  print ( Ptr p) {
 printf ("%d ", *(int*)p );
}
   // comparare de intregi
int intcmp ( void* a, void* b) {
  return *(int*)a - *(int*)b;
}
void main () {
      // citire numere si creare multime
   Set a; int x; int * p;
   initS(a);
   printf ("Elem. multime: \n");
   while ( scanf ("%d", &x) > 0) {
     p= (int*) malloc (sizeof(int));
     *p=x;
     add(a,p,intcmp);
   }
   printS (a);
}
```

15. Diferente între limbajele C si C++

Diferente de sintaxã

Limbajul C++ este o extindere a limbajului C pentru programare orientatã pe obiecte. Limbajul C++ aduce o serie de inovatii fatã de limbajul C standard care nu sunt legate direct de aparitia claselor: alt fel de comentarii, noi operatori, noi tipuri de date (referinte), noi reguli sintactice s.a.

Cu ocazia adãugãrii facilitãtilor necesare POO s-au mai adus si alte îmbunãtãtiri limbajului C, astfel încât C++ este un "C ceva mai bun".

Cu mici exceptii, existã compatibilitate între cele douã limbaje, în sensul cã un program C este acceptat de compilatorul C++ si produce aceleasi rezultate la executie. Unele compilatoare tratează continutul unui fisier sursã în functie de extensia la numele fisierului: fisierele cu extensia CPP contin programe C++, iar fisiere cu orice altã extensie se considerã a fi scrise în C.

O parte dintre inovatiile aduse sunt importante si pentru cei care nu folosesc clase în programele lor.

In C++ se preferã altã definire pentru constante simbolice,în loc de directiva #define, care permite verificãri de tip din partea compilatorului. Exemplu:

```
const int NMAX=1000;
void main () {
  int n, x[NMAX], y[NMAX];
  . . .
```

In C++ declaratiile de variabile sunt tratate la fel cu instructiunile si deci pot apare oriunde într-un bloc, dar în C declaratiile trebuie sã preceadã prima instructiune executabilã dintr-un bloc. Se poate vorbi chiar un alt stil de programare, în care o variabilã este declaratã acolo unde este folositã prima datã. Exemplu:

```
   // suma valorilor dintr-un vector
  float sum (float x[], int n ) {
    float s=0;
    for ( int i=0; i<n; i++)
       s += x[i];
    return s;
  }
```

Domeniul de valabilitate al variabilei este blocul unde a fost declaratã variabila, dar instructiunea *for* prezintã un caz special. In versiunile mai noi ale limbajului C++ domeniul de valabilitate al variabilei declarate într-o instructiune *for* este limitat la instructiunile care vor fi repetate (din cadrul ciclului *for*). Din acest motiv secventa urmãtoare poate produce sau nu erori sintactice:

```
for (int i=0;i<6;i++) a[i]=i;
for (int i=0;i<6;i++) printf ("%d ", a[i];
```

In C++ se admite folosirea numelui unui tip structurã, fãrã a fi precedat de ***struct*** si fãrã a mai fi necesar ***typedef***. Exemplu:

```
struct nod {
   int val;
   nod * leg;          // in C: struct nod * leg
};
nod * lista;           // in C: struct nod * lista
```

Diferente la functii

In C++ toate functiile folosite trebuie declarate si nu se mai considerã cã o functie nedeclaratã este implicit de tipul ***int***. Dar o functie definitã fãrã un tip explicit este consideratã ca fiind de tip ***int***. Asa se explicã de ce functia ***main*** este deseori declaratã ca fiind de tip ***void***; absenta cuvântului ***void*** implicã tipul ***int*** pentru functia ***main*** si compilatorul verificã existenta unei instructiuni ***return*** cu expresie de tip întreg.

Absenta unei declaratii de functii (scrisã direct sau inclusã dinntr-un fisier H) este eroare gravã în C++ si nu doar avertisment ca în C (nu trece de compilare)

In C++ se pot declara valori implicite pentru parametri formali de la sfârsitul listei de parametri; aceste valori sunt folosite automat în absenta parametrilor efectivi corespunzãtori la un apel de functie. O astfel de functie poate fi apelatã deci cu un numãr variabil de parametri. Exemplu:

```
     // afisare vector, precedata de un titlu
void printv ( int v[ ], int n, char * titlu="") {
   // afiseaza sirul primit sau sirul nul
  printf ("\n %s \n", titlu);
  for (int i=0; i<n; i++)
    printf ("%d ", v[i]);
}
    ...
   // exemple de apeluri
printv ( x,nx );              // cu 2 parametri
printv (a,na," multimea A este");   // cu 3 parametri
```

In C++ functiile scurte pot fi declarate ***inline***, iar compilatorul înlocuieste apelul unei functii ***inline*** cu instructiunile din definitia functiei, eliminând secventele de transmitere a parametrilor. Functiile ***inline*** sunt tratate ca si macrourile definite cu ***define***. Orice functie poate fi declaratã ***inline***, dar compilatorul poate decide cã anumite functii nu pot fi tratate ***inline*** si sunt tratate ca functii obsnuite. De exemplu, functiile care contin cicluri nu pot fi ***inline***. Utilizarea unei functii ***inline*** nu se deosebeste de aceea a unei functii normale. Exemplu de functie ***inline***:

```
inline int max (int a, int b) { return a>b ? a : b; }
```

In C++ pot exista mai multe functii cu acelasi nume dar cu parametri diferiti (ca tip sau ca numãr). Se spune cã un nume este "supraîncãrcat" cu semnificatii ("function overloading"). Compilatorul poate stabili care din functiile cu acelasi nume a fost apelatã într-un loc analizând lista de parametri si tipul functiei. Exemple:

```
float abs (float f) { return fabs(f); }
long abs (long x) { return labs(x); }
printf ("%6d%12ld  %f \n", abs(-2),abs(-2L),abs(-2.5) );
```

Supradefinirea se practicã pentru functiile membre (din clase) si, în particular, pentru operatori definiti fie prin functii membre, fie prin functii prietene.

Operatori pentru alocare dinamicã

In C++ s-au introdus doi operatori noi, pentru alocarea dinamicã a memoriei ***new*** si pentru eliberarea memoriei dinamice ***delete***, destinati sã înlocuiascã functiile de alocare si eliberare (malloc, free, s.a.). Operatorul ***new*** are ca operand un nume de tip, urmat în general de o valoare initialã pentru variabila creatã (între paranteze rotunde); rezultatul lui ***new*** este o adresã (un pointer de tipul specificat) sau NULL daca nu existã suficientã memorie liberã. Exemple:

```
nod * pnod;                 // pointer la nod de lista
pnod = new nod;             // alocare fara initializare
assert (pnod != NULL);
int * p = new int(3);             // alocare cu initializare
```

Operatorul ***new*** are o formã putin modificatã la alocarea de memorie pentru vectori, pentru a specifica numãrul de componente. Exemplu:

```
int * v = new int [n];      // vector de n intregi
```

Operatorul ***delete*** are ca operand o variabilã pointer si are ca efect eliberarea blocului de memorie adresat de pointer, a cãrui mãrime rezultã din tipul variabilei pointer sau este indicatã explicit. Exemple:

```
int * v;
delete v;           // elibereaza sizeof(int) octeti
delete [ ] v;
delete [n] v;       // elibereaza n*sizeof(int) octeti
```

Operatorul de rezolutie "::" este necesar pentru a preciza domeniul de nume cãruia îi apartine un nume de variabilã sau de functie.

Fiecare clasã creeazã un domeniu separat pentru numele definite în acea clasã (pentru membri clasei). Deci un acelasi nume poate fi folosit pentru o variabila externã (definitã în afara claselor), pentru o variabilã localã unei functii sau pentru o variabilã membrã a unei clase (structuri). Exemplu:

```
int end;              // variabila externa
void cit () {
  int end=0;          // variabila locala
...
  if (::end) { ...}   // variabila externa
}
class A {
 public:
   int end;           // variabila membru a clasei A
   void print();
 ...
};
// exemple de utilizare in "main"
 end=1;          // sau
 A::end=0;
 f.seekg (0, ios::end);   // din clasa predefinita "ios"
 ...
```

Utilizarea operatorului de rezolutie este necesarã si la definirea metodelor unei clase în afara clasei, pentru a preciza compilatorului cã este definitia unei metode si nu definitia unei functii externe. Exemplu:

```
  // definitie metoda din clasa A
void A:: print () { ... }
  // definitie functie externa
void print () { ... }
```

Tipuri referintã

In C++ s-au introdus tipuri referintã, folosite în primul rând pentru parametri modificabili sau de dimensiuni mari. Si functiile care au ca rezultat un obiect mare pot fi declarate de un tip referintã, pentru a obtine un cod mai performant. Caracterul ampersand (&) folosit dupã tipul si înaintea numelui unui parametru formal (sau unei functii) aratã compilatorului cã pentru acel parametru se primeste adresa si nu valoarea argumentului efectiv. Exemplu:

```
  // schimba intre ele doua valori
void schimb (int & x, int & y) {
   int t = x;
```

```
    x = y; y = t;
  }
   // ordonare vector
  void sort ( int a[], int n ) {
    ...
    if ( a[i] > a[i+1])
      schimb ( a[i], a[i+1]);
    ...
  }
```

Spre deosebire de un parametru pointer, un parametru referintã este folosit de utilizator în interiorul functiei la fel ca un parametru transmis prin valoare, dar compilatorul va genera automat indirectarea prin pointerul transmis (în programul sursã nu se foloseste explicit operatorul de indirectare '*').

Referintele simplificã utilizarea unor parametri modificabili de tip pointer, eliminând necesitatea unui pointer la pointer. Exemplu de definire în C++ a functiei "strtoi" si de utilizare a functiei cu argument referintã:

```
int strtoi (char *start, char * & stop) {
  char * p=start;
  while (*p !=0 && isspace(*p))        // ignora spatii
       p++;
  start=p;
  while (*p != 0 && isdigit(*p) )
      p++;
     stop= p+1;
     return atoi(start);
  }
   // utilizare
  void main () {
      char * s =" 1 12   123  1234  ";
      char * p=s ; int x;
      while ( x=strtoi(p,p))
        printf( "%d \n",x);
  }
```

Sintaxa declararii unui tip referintã este urmãtoarea:

tip & nume

unde "nume" poate fi:

- numele unui parametru formal
- numele unei functii (urmat de lista argumentelor formale)
- numele unei variabile (mai rar)

Efectul caracterului '&' în declaratia anterioarã este urmãtorul: compilatorul creeazã o variabilã "nume" si o variabilã pointer la variabila "nume", initializeazã variabila pointer cu adresa asociatã lui "nume" si retine cã orice referire ulterioarã la "nume" va fi tradusã printr-o indirectare prin variabila pointer anonimã creatã.

O functie poate avea ca rezultat o referintã la un vector dar nu poate avea ca rezultat un vector.

O functie nu poate avea ca rezultat o referintã la o variabila localã, asa cum nu poate avea ca rezultat un pointer la o variabila localã. Exemplu:

```
typedef int Vec [M];
  // adunarea a 2 vectori - gresit !
Vec& suma (Vec a, Vec b, int n) {
  Vec c;
  for (int i=0; i<n;i++)
     c[i]=a[i]+b[i];
  return c;       // eroare !!!
}
```

Fluxuri de intrare-iesire

In C++ s-a introdus o altã posibilitate de exprimare a operatiilor de citire-scriere, pe lângã functiile standard de intrare-iesire din limbajul C. In acest scop se folosesc câteva clase predefinite pentru "fluxuri de I/E" (declarate în fisierele antet <iostream.h> si <fstream.h>).

Un flux de date ("stream") este un obiect care contine datele si metodele necesare operatiilor cu acel flux. Pentru operatii de I/E la consolã sunt definite variabile de tip flux, numite "cin" (console input), "cout" (console output).

Operatiile de citire sau scriere cu un flux pot fi exprimate prin metode ale claselor flux sau prin doi operatori cu rol de extractor din flux (>>) sau insertor în flux (<<). Atunci când primul operand este de un tip flux, interpretarea acestor operatori nu mai este cea de deplasare binarã ci este extragerea de date din flux (>>) sau introducerea de date în flux (<<).

Operatorii << si >> implicã o conversie automatã a datelor între forma internã (binarã) si forma externã (sir de caractere). Formatul de conversie poate fi controlat prin cuvinte cheie cu rol de "modificator". Exemplu de scriere si citire cu format implicit:

```
#include <iostream.h>
void main ( ) {
  int n; float f; char s[20];
  cout << " n= ";  cin >> n;
  cout << " f= ";  cin >> f;
  cout << " un sir: "; cin >> s;  cout << s << "\n";
}
```

Intr-o expresie ce contine operatorul << primul operand trebuie sã fie "cout" (sau o altã variabilã de un tip "ostream"), iar al doilea operand poate sã fie de orice tip aritmetic sau de tip "char*" pentru afisarea sirurilor de caractere. Rezultatul expresiei

fiind de tipul primului operand, este posibilã o expresie cu mai multi operanzi (ca la atribuirea multiplã). Exemplu:

```
cout << "x= " << x << "\n";
```

este o prescurtare a secventei de operatii:

```
cout << "x= "; cout << x; cout << "\n";
```

In mod similar, într-o expresie ce contine operatori >> primul operand trebuie sã fie "cin" sau de un alt tip "istream", iar ceilalti operanzi pot fi de orice tip aritmetic sau pointer la caractere. Exemplu:

```
cin >> x >> y;
```

este echivalent cu secventa:

```
cin >> x; cin >> y;
```

Operatorii << si >> pot fi încãrcati si cu alte interpretãri, pentru scrierea sau citirea unor variabile de orice tip clasã, cu conditia supradefinirii lor .

Este posibil si un control al formatului de scriere prin utilizarea unor “modificatori”.

Tipuri clasã

Tipurile clasã reprezintã o extindere a tipurilor structurã si pot include ca membri variabile si functii. Pentru definirea unei clase se poate folosi unul din cuvintele cheie *class*, *struct* sau *union*, cu efecte diferite asupra atributelor de accesibilitate ale membrilor clasei:

- O clasã definitã prin *class* are implicit toti membri invizibili din afara clasei (de tip *private*).
- O clasa definitã prin *struct* sau *union* are implicit toti membri publici, vizibili din afara clasei.

In practicã avem nevoie ca datele clasei sã fie ascunse (locale) si ca functiile clasei sã poatã fi apelate de oriunde (publice). Pentru a stabili selectiv nivelul de acces se folosesc cuvintele cheie *public*, *private* si *protected*, ca etichete de sectiuni cu aceste atribute, în cadrul unei clase. In mod uzual, o clasã are douã sectiuni: sectiunea de date (*private*) si sectiunea de metode (*public*).

Functiile unei clase, numite si metode ale clasei, pot fi definite complet în cadrul definitiei clasei sau pot fi numai declarate în clasã si definite în afara ei

Exemplul urmãtor contine o variantã de definire a unei clase pentru un vector extensibil de numere întregi:

```
class intArray {            // clasa vector de intregi
  // date clasei (private)
   int * arr;               // adresa vector (alocat dinamic)
   int d,dmax,inc;          // dimensiune curenta si maxima
   void extend();           // implicit private, definita ulterior
public:
  intArray (int max=10, int incr=10) {     // constructor
    dmax=max; inc=incr; d=0;
```

```
    arr= new int[dmax];
   }
  ~intArray () { delete [ ] arr;}            // destructor
  int get (int i)
   { assert (i >= 0 && i < dmax);
     return arr[i];
   }
  void add (int elem) {       // adauga un element la vector
    if ( d==dmax)
      extend();
    arr[d++]=elem;
  }
  int size() { return d; }       // dimensiune curenta vector
};
 // extindere vector
   void intArray::extend () {
    int * oldarr=arr;
    dmax+=inc;
    arr = new int[dmax];
    for (int i=0;i<d;i++)
      arr[i]= oldarr[i];
    delete [ ] oldarr;
  }
```

Pentru clasele folosite în mai multe aplicatii, cum este clasa “intArray”, se recomandã ca toate functiile clasei sã fie definite în afara clasei, într-un fisier sursã separat; eventual se compileazã si se introduc într-o bibliotecã. Definitia clasei se pune într-un fisier antet separat, care va fi inclus de toate fisierele sursã ce folosesc tipul respectiv. In acest fel este separatã descrierea clasei de implementarea clasei si de utilizãrile clasei în diverse aplicatii. Exemplu:

```
  // fisier INTARRAY.H
class intArray {
 private:
   int * arr;              // adresa vector (alocat dinamic)
   int d,dmax,inc;         // dimensiune curenta si maxima
   void extend();          // implicit private, definita ulterior
 public:
   intArray (int max=10, int incr=10);  // constructor
   ~intArray ();           // destructor
   int get (int ) ;        // extrage element
   void add (int );        // adauga element
   int size();            // dimensiune vector
};
   // fisier INTARRAY.CPP
 #include “intArray.h”
  intArray::intArray (int max=10, int incr=10){
    dmax=max; inc=incr; d=0;
    arr= new int[dmax];
```

```
  }
 intArray::~intArray () { delete [] arr;}
 intArray::int get (int i)
  { assert (i >= 0 && i < dmax);
    return arr[i];
  }
 void intArray::add (int elem) {
   if ( d==dmax)
     extend();
   arr[d++]=elem;
 }
 int intArray::size() {
   return d;
 }
   // fisier care foloseste clasa intArray : TEST.CPP
#include “intArray.h”
#include <iostream.h>
void main () {
 intArray a(3,1);          // iniial 3 elemente, increment 1
 for (int i=1;i<=10;i++)
    a.add(i);
 for (i=0;i< a.size();i++)
   cout << a.get(i) << ' ';
 cout << endl;
}
```

Orice clasã are (cel putin) o functie constructor (publicã) apelatã implicit la definirea de variabile de tipul clasei; un constructor alocã memorie si initializeazã variabilele clasei. Functiile constructor au toate numele clasei si pot diferi prin lista de argumente. O functie constructor nu are tip.

O functie destructor este necesarã numai pentru clase cu date alocate dinamic (în constructor).

Sintaxa pentru apelul unei metode (nestatice) extinde referirea la membri unei structuri si se interpretează ca apel de functie pentru un obiect dat prin numele sãu.

Supradefinirea operatorilor

Pentru variabilele de un tip clasã (structurã) se pot folosi numai doi operatori, fãrã a mai fi definiti. Acestia sunt operatorul de atribuire ('=') si operatorul de obtinere a adresei variabilei ('&'). La atribuirea între variabile de un tip clasã se copiazã numai datele clasei .

Alte operatii cu obiecte se definesc prin functii si/sau operatori specifici clasei respective.

Operatorii limbajului C pot fi supradefiniti, adicã pot fi asociati si cu alte operatii aplicate unor variabile de tip clasã. Aceastã facilitate este utilã în cazul claselor de definesc noi tipuri de date.

Un operator este considerat în C++ ca o functie cu un nume special, dar supus tuturor regulilor referitoare la functii. Numele unei functii operator constã din cuvântul *operator* urmat de unul sau douã caractere speciale, prin care se foloseste operatorul. In exemplul urmãtor se defineste o clasã pentru siruri de caractere, cu un operator de concatenare siruri (‘+’).

```
class string {
 char * start; // adresa sir terminat cu zero
public:
 string ( char * s);    // un constructor
 string () { start=new char[80]; *start='\0';}
 ~string() {delete start; }
 string& operator + (string& sir);
 void show (void) { cout << start << '\n';}
};
  // functii ale clasei 'string'
string::string ( char * s) {
 int lung= strlen(s);
 start=new char[lung+1];
 strcpy (start,s);
}
string& string::operator + (string& str) {
 int lung=strlen(start)+strlen(str.start);
 char * nou=new char[lung+1];
 strcpy (nou,start);  strcat (nou,str.start);
 delete start; start=nou;
 return * this;          // this este adresa obiectului curent
}
 // teste
main () {
 string s1 ("zori "), s2 ("de "), s ;
 s= s1+s2; s.show();          // concatenare siruri
 string s3 ("zi");
 s= s1+s2+s3; s.show();
}
```

16. Programare orientatã pe obiecte în C++

Clase si obiecte

Programarea cu obiecte (POO) este un alt mod de abordare a programãrii decât programarea proceduralã (în limbaje ca C si Pascal), cu avantaje în dezvoltarea programelor mari.

In programarea proceduralã sarcina programatorului este de a specifica actiuni de prelucrare, sub forma de proceduri (functii, subprograme). Un program C sau Pascal este o colectie de functii (si/sau proceduri). Datele prelucrate se transmit ca argumente (parametri) de la o functie la alta. Pentru anumite operatii uzuale existã functii predefinite (de bibliotecã).

In programarea orientatã pe obiecte programatorul lucreazã cu obiecte: declarã obiectele necesare aplicatiei (si atributele lor), iar prelucrãrile sunt exprimate ca actiuni asupra obiectelor (prin apeluri de metode). Sunt prevãzute mai multe tipuri de obiecte predefinite (clase de bibliotecã), dar orice programator poate defini noi tipuri de obiecte (noi clase). Un obiect contine în principal date.

O clasã este un tip de date asemãnãtor unei structuri din C (“struct”), dar care contine si functii pentru operatiile asociate datelor continute in clasã. De exemplu, o clasã pentru date calendaristice “Date” contine ca date trei numere întregi ce reprezintã ziua, luna si anul, iar ca metode ale clasei are functii pentru compararea a douã date, pentru calculul diferentei a douã date, pentru afisarea unei date într-un anumit format (sau pentru crearea unui sir de caractere cu zi, lunã, an în vederea afisãrii), s.a.

Un obiect corespunde unei variabile de un tip structurã; de exemplu un obiect de tip “Date” contine o datã concretã, cum ar fi {25, 12, 2004}. Asupra unui astfel de obiect sunt permise numai operatiile prevãzute de metodele clasei “Date”.

O clasã corespunde unei notiuni abstracte cum ar fi “orice datã calendaristicã”, iar un obiect este un caz concret (o realizare a conceptului sau o instantiere a clasei).

Vom prezenta câteva exemple (în limbajul C++) care sã ilustreze comparativ modul de abordare al programãrii procedurale si al programãrii orientate pe obiecte.

Primul exemplu comparã douã date calendaristice si afiseazã relatia dintre ele.

```
    /* operatii cu structura "date" in C */
typedef struct {
  int zi, luna, an ;
} date ;
  /* transforma data in sir de car */
char * toString (date d) {
    char * str = (char*) malloc(14);         // sir de forma zz-ll-aaaa
    sprintf(str," %02d-%02d-%04d ",d.zi,d.luna,d.an);
    return str;
}
  /* comparare date calendaristice */
int cmpdat (date d1,date d2 ) {
```

```
   int cmpan, cmplun;
   cmpan = d1.an-d2.an;              // compara ani
   if (cmpan) return cmpan;      // un nr. negativ sau pozitiv
   cmplun = d1.luna-d2.luna;    // compara luni
   if (cmplun) return cmplun;     // un nr. negativ sau pozitiv
   return d1.zi- d2.zi;              // zero la egalitate de date
}
 /* utilizare functii */
void main () {
  date d1,d2;  int res;
  char oper = '=';                  // operator afisat intre date
  printf ( "Data de forma zi luna an: ");
  scanf("%d%d%d", &d1.zi,&d1.luna,&d1.an);
  printf ( "Data de forma zi luna an: ");
  scanf("%d%d%d", &d2.zi,&d2.luna,&d2.an);
  res = cmpdat(d1,d2);          // rezultat comparatie
  if (res < 0) oper='<';   if (res > 0) oper='>';
  printf ("%s %c %s \n", toString(d1), oper, toString(d2));
}
```

Urmează varianta C++, cu clasă în loc de structură pentru data calendaristică:

```
class Date {            // clasa pentru date calendaristice
 private:
    int zi,luna,an;     // variabile ale clasei (datele clasei)
 public:
  Date (int z, int l, int a) {       // constructor ptr obiectele clasei
    zi=z; luna=l ; an=a;
  }
  int compare ( Date d) {       // compara ob. curent cu obiectul d
    int cmpan = an-d.an;         // compara ani
    if (cmpan) return cmpan;     // un nr. negativ sau pozitiv
    int cmplun = luna-d.luna;    // compara luni
    if (cmplun) return cmplun;    // un nr. negativ sau pozitiv
    return zi- d.zi;                 // zero la egalitate de date
  }
  char * toString () {              // produce un sir cu datele clasei
    char * str = new char[11];  // sir de forma zz-ll-aaaa
    sprintf(str," %02d-%02d-%04d ",zi,luna,an);
    return str;
  }
};
  // utilizare
void main () {
  int z,l,a; char op = '=';             // op este operatorul afisat (=,<,>)
  cout << "Data de forma zi luna an: ";
  cin >> z >> l >> a;                     // citeste 3 intregi de la consola
  Date d1 (z,l,a);                       // un obiect d1
  cout << "Data de forma zi luna an: ";
  cin >> z >> l >> a;                     // citeste 3 intregi
```

```
  Date d2 (z,l,a);                    // alt obiect d2
  int res = d1.compare(d2);           // rezultat comparatie  obiecte
  if (res < 0) op='<';                // alege cod operator functie de rezultat comp.
  if (res > 0) op='>';
  cout << d1.toString() << op << d2.toString() << '\n';     // afisare  d1 op d2
}
```

Al doilea exemplu prezintã comparativ modul de exprimare a operatiilor de citire-scriere din (în) fisiere text, cu functii si cu obiecte.

Pentru operatii cu fisiere de date în C existã o serie de functii predefinite, cum ar fi fopen, fclose, fgets, fputs s.a. Programatorul unei aplicatii cu fisiere apeleazã aceste functii transmitând ca unul din argumente numele (identificatorul) fisierului care face obiectul actiunii:

```
 // copiere fisier in C
void copyFile ( char * src, char * dst) {
   char line[1000], * adr;
   FILE * f1 =fopen(src,"r");      // deschide fisier sursa
   FILE * f2 =fopen(dst,"w");         // deschide fisier destinatie
   do {
     adr= fgets(line,1000,f1);   // citeste o linie din f1
     if (adr==NULL) break;      // la sfarsit de fisier fgets are rezultat NULL
     fputs (line,f2);               // scrie linie in f2
   } while (adr);
   fclose(f2);                     // eventual si fclose(f1)
}
```

In C++ sunt definite câteva clase pentru fisiere de date (numite fluxuri de date), iar operatiile se exprimã prin apeluri de metode ale acestor clase. Fiecare fisier prelucrat de program este reprezentat printr-un obiect:

```
  // copiere fisier in C++
void copyFile ( char * src, char * dst) {
   const M=1000;    char line[M];
   ifstream f1;               // f1 este obiect din clasa "ifstream"
   ofstream f2;               // f2 este obiect din clasa "ofstream"
   f1.open (src); f2.open(dst);       // metoda open din clasele ifstream, ofstream
   while ( ! f1.eof()) {              // apel metoda eof din clasa ifstream
     f1.getline(line,M);               // apel metoda getline din clasa ifstream
     f2.write(line,strlen(line); f2.put('\n');   // metode write, put din clasa ofstream
   }
   f2.close();                // metoda close din clasa ifstream
}
```

Deschiderea unui flux se poate face chiar la definirea obiectului flux de I/E :

```
ifstream f1(src); ofstream f2(dst);     // alt mod de construire obiecte flux de I/E
while ( ! f1.eof()) { ... }
```

La definirea unui obiect (unei variabile de un tip clasã) se apeleazã automat o functie constructor, care face initializari ale variabilelor clasei. Deschiderea unui fisier este consideratã ca operatie de initializare si realizatã (optional) de un constructor al clasei (pot fi mai multi constructori, cu diferite argumente, dar cu acelasi nume) .

Un alt exemplu se referã la extragerea de cuvinte (simboluri = tokens) dintr-un sir de caractere în care cuvintele sunt separate prin caractere separator specificate de programator. In C se foloseste functia de biblioteca "strtok" pentru extragerea de cuvinte, ca în exemplul urmãtor:

```
   // functie de creare vector de pointeri la cuvinte
int tarray (char * text, char * separ, char* vec[]) {
  int i=0;  char * p;
  p=strtok (text,separ);       // primul apel (primul cuvant)
  while ( p != NULL) {         // p=NULL la sfarsit text
    vec[i++]= p;               // pune adresa cuvant in vector
    p=strtok(0,separ);         // alte apeluri, ptr urmatoarele cuvinte
  }
  return i;          // numar de cuvinte gasite
}
```

In Java existã clasa "StringTokenizer" cu metodele "nextToken" (sau "next") si "hasMoreTokens" ("hasNext") pentru aceastã operatie de împãrtire în cuvinte. Dacã definim în C++ o clasã "StrTok", similarã cu "StringTokenizer", atunci functia de creare a vectorului de cuvinte aratã astfel:

```
  // functie de creare vector de pointeri la cuvinte
int tarray (char * text, char * separ, char* vec[ ]) {
  int i=0;
  StrTok st(text,separ);       // constructor cu 2 argumente ptr obiectul "st"
  while ( st.hasNext())        // apel metoda "hasNext" (daca mai sunt cuvinte)
    vec[i++]= st.next();       // apel metoda "next" ptr urmatorul cuvant
  return i;                    // numar de cuvinte gasite
}
```

In exemplele anterioare si în cele ce vor urma se poate observa mutarea accentului de pe actiuni (functii) pe obiecte (date) în programarea orientatã pe obiecte. Numele de clase sunt substantive, uneori derivate din verbul ce defineste principala actiune asociatã obiectelor respective. In Java existã obiecte comparator (de un subtip al tipului “Comparator”) folosite în compararea altor obiecte, clasa "Enumerator" folositã la enumerarea elementelor unei colectii, clasa "StringTokenizer" folositã la extragerea cuvintelor dintr-un sir s.a.

Chiar si sintaxa apelãrii metodelor aratã cã o metodã (de ex. "write") se apeleazã pentru un anumit obiect (de tip "ofstream"), deci orice actiune (metodã) existã numai în contextul unei clase si se poate folosi numai pentru obiecte din clasa respectivã. In POO metodele sunt subordonate obiectelor, obiecte care sunt date.

Utilizarea de obiecte

Din punct de vedere sintactic, o clasã este o structurã extinsã, care poate contine ca membri atât variabile cât si functii. Fiecare definire a unui tip structurã introduce un nou tip de date, care poate fi folosit în declararea de variabile, de functii si argumente de functii. La fel, fiecare definitie de clasã introduce un nou tip de date. Operatiile posibile cu variabile de un tip clasã sunt cele definite prin metodele publice ale clasei.

In Java este definitã o clasa "Vector" (redenumitã si "ArrayList") pentru un vector extensibil de obiecte generice (de tip "Object"). Putem sã ne definim o clasã vector si în C++, sau mai multe clase vector pentru diferite tipuri de elemente componente.

Fie o clasa “IntArray” pentru un vector extensibil de numere întregi. Un obiect de acest tip este o variabilã a cãrei declaratie depinde de functiile constructor definite în clasã. Dacã existã (si) un constructor fãrã argumente (sau toate argumentele au valori implicite) atunci putem scrie astfel:

```
IntArray  a,b;          // dar nu si :    IntArray a();
```

Dacã existã un constructor cu argument întreg care specificã dimensiunea initialã a vectorului, atunci putem scrie declaratii de forma:

```
IntArray  a(100), b(200);
```

Un constructor al clasei este apelat automat în C++ la declararea unei variabile de un tip clasã si de cãtre operatorul "new", care alocã memorie pentru un nou obiect. Exemple de alocare dinamicã a unor obiecte de tip vector :

```
IntArray * pa = new IntArray (100);
IntArray * p = new IntArray();    //  sau:   p = new IntArray;
```

Functia urmãtoare creeazã un obiect pe baza unui vector C si poate fi rescrisã sub forma unui constructor al clasei "IntArray":

```
void create (int v[ ], int nv, IntArray a) {
     for(int i=0;i<nv;i++)
        a.add (v[i]);
}
```

Am presupus cã existã în clasa "IntArray" o metodã "add" care adaugã un întreg la sfârsitul unui vector, cu eventuala mãrire de capacitate pentru vector.

Pentru a ilustra si alte utilizãri ale unui tip clasã vom prezenta si o altã variantã a functiei anterioare:

```
IntArray create (int v[ ], int nv) {
  IntArray* ap = new IntArray(nv);       // pointer la un obiect
  for (int i=0;i<nv;i++)
    ap->add(v[i]);
  return *ap;                            // rezultat este obiectul de la adresa ap
}
```

Operatorul “new” din C++ alocã memorie (ca si functia “malloc”) si apeleazã constructorul clasei ce corespunde listei de argumente dintre paranteze (care va initializa memoria alocatã de “new”).

O metodã a unei clase este membru al acelei clase si se apeleazã altfel decât o functie definitã în afara oricãrei clase. O metodã (nestaticã) se apeleazã pentru un anumit obiect; de exemplu, metoda "add" este apelatã pentru obiectul “a”.

Metodele declarate statice pot fi folosite chiar dacã nu existã obiecte din clasa respectivã, dar trebuie precedate de numele clasei (pentru cã putem avea metode cu acelasi nume în clase diferite). Metodele statice se folosesc mai rar în C++, dar mai frecvent în Java, unde nu existã functii exterioare claselor.

In C++ putem defini si folosi variabile de un tip clasã, sau pointeri la un tip clasã, sau referinte la un tip clasã. O referintã este tot un pointer, dar folosit ca si cum ar fi o variabilã de tipul respectiv (indirectarea prin pointer se face automat de compilator). Variabilele referintã trebuie initializate la declarare, în C++.

Exemple:

```
Date d (1,12,1918);                         // obiect de tip Date
Date * dp = new Date (1,12,1918);         // pointer la un obiect Date
Date & dr = d;   Date&  ddr = * new Date(1,1,2005);  // referinte la ob. Date
cout << d.toString() << “ “ << dp->toString() <<” “ << ddr.toString() << ‘\n’;
```

In C++ o functie (sau o metodã) poate avea ca rezultat un obiect, sau o referintã la un obiect, sau un pointer la un obiect.

In Java si în C# orice variabilã de un tip clasã este implicit o referintã la un obiect din clasa respectivã; de exemplu, o variabilã “Date” nu este un nume pentru un obiect ci contine adresa unui obiect de tip “Date” (o referintã la un obiect). Argumentele de functii de un tip clasã sunt si ele implicit referinte, în Java.

In C++ se practicã argumente de functii de tip referintã la obiecte din douã motive: obiectul transmis prin referintã poate fi modificat de functie si transmiterea de adrese la functii în loc de obiecte este mai eficientã ca memorie si ca timp. Exemplu:

```
  // incrementare datã (ziua urmatoare)
void nextDay (Date& d) {
   if ( lastDay (d.zi,d.luna) ) {      // daca e ultima zi din luna
        d.zi=1; d.luna++;          // ar trebui verificat si daca e ultima  luna !
   }
    else
        d.zi++;        // ziua urmatoare din aceeasi luna
}
```

Definirea de noi clase

In C++ un tip clasã se defineste la fel cu un tip structurã, cu câteva particularitãti:

- Cuvântul cheie "struct" este înlocuit de obicei prin cuvântul cheie "class", desi se poate folosi chiar si "struct" pentru definirea de clase.

- Functiile membre ale unei clase pot fi definite în cadrul definitiei clasei sau în afara clasei, dar atunci trebuie declarate în clasã.

- Nu toti membri unei clase sunt accesibili din afara clasei; nivelul de accesibilitate este declarat prin cuvintele cheie "public", "private" si "protected", folosite ca etichete pentru secvente de declaratii care au toate acest atribut. De obicei variabilele clasei sunt "private" sau "protected", iar functiile (metodele) sunt marcate cu "public".

Multimea metodelor publice constituie interfata clasei cu exteriorul, sau interfata expusã de clasa respectivã cãtre utilizatorii potentiali (functii sau metode externe).

Exemplu de definire a unei clase:

```
class StrTok {
private:                // poate lipsi, este implicit
  char * str;             // sirul analizat
  char * token;        // adr cuv urmator
  char * sep;          // sirul caracterelor separatori de cuvinte
public:
  StrTok (char * str, char * delim="\t \n\r") {        // constructor
    this->str= strdup(str);
    sep=delim;
    token=strtok(this->str,sep);
  }
  char * next () {             // o metoda
    if (token == NULL)
       return NULL;
    char* aux= token;
    token=strtok(0,sep);
    return aux;
  }
  int hasNext () {              // alta metoda
    return token !=NULL;
  }
};
```

In C++ metodele pot fi definite si în afara clasei, iar metodele mai lungi sau care contin cicluri ar trebui definite în afara clasei. In plus, definitia unei clase poate fi introdusã într-un fisier antet separat (.H), inclus în definitia unor clase derivate si aplicatiilor care folosesc obiecte de acest tip clasa. Exemplu:

```
  // vectori de intregi (fisier "IntArray.h")
class IntArray {
protected:             // pentru a permite def. de clase derivate
   int * vec;           // adresa vector
   int d,dmax,inc;      // dimens. efectiva, maxima si increment
public:
  IntArray (int max=10, int incr=10) ;   // constructor
  ~IntArray();                    // destructor
  void add(int);                  // adaugare intreg la vector
  int size();                     // dimensiune efectiva vector
```

```
  int get(int);                    // valoare element dintr-o pozitie
  int indexOf (int);               // indicele unui numar dat in vector (-1 daca nu exista)
  void print();                    // afisare vector
  void remove ();                  // elimina ultimul element
};
```

La definirea metodelor trebuie specificatã si clasa de care apartin, ca parte din numele metodei. Exemple:

```
 int IntArray:: size() {       // metoda “size” din clasa “IntArray”
   return d;
 }
 void IntArray:: add (int x) {   // metoda “remove” din clasa “IntArray”
    if ( d==dmax)          // daca vector plin
       extend();           // extindere vector (metoda “private” a clasei)
     vec[d++]=x;           // adaugare la sfârsitul vectorului
 }
 void IntArray::print () {      // metoda “print” din clasa “IntArray”
   for (int i=0;i<d ;i++)
     cout << vec[i] << ' ';
    cout << endl;
 }
```

Datele clasei sunt de obicei inaccesibile direct pentru functii din afara clasei, pentru a evita modificări nedorite ale acestor date. Se spune cã datele sunt încapsulate în obiecte (un obiect este o capsulã) sau cã datele sunt “ascunse”. Variabilele "private" accesibile pentru citire si scriere prin metode publice se mai numesc si proprietãti.

Orice clasã are una sau mai multe functii constructor, având numele clasei. De obicei se defineste explicit efectul functiei constructor, dar absenta unui constructor nu este o eroare sintacticã, pentru cã este generat automat de compilator un constructor cu efect nul. Constructorul permite initializarea automatã a datelor dintr-un obiect simultan cu crearea (cu definirea) obiectului, evitindu-se astfel erori de utilizare a unor obiecte neinitializate. Exemplu:

```
   // constructor de obiecte IntArray
 IntArray (int max=10, int incr=10) {
   dmax=max; inc=incr; d=0;
    vec= new int[dmax];
 }
```

De remarcat cã functia constructor nu are tip, pentru cã ea nu este apelatã explicit într-un program; compilatorul genereazã apeluri ale functiei constructor la definirea unor variabile de un tip clasã, la crearea dinamicã de obiecte (cu operatorul ***new***) si în alte situatii (transmitere de parametri, de rezultate intermediare s.a.)

In C++, functiile constructor folosesc parametri cu valori implicite, care simplificã declararea unor obiecte si reduce numãrul de functii constructor ce trebuie definite.

Obiectele unei clase, ca orice alte variabile, au o duratã de viatã si o clasã de alocare, rezultate din locul unde au fost definite. Deci putem avea obiecte locale unei functii, care existã în memorie numai cât este activã functia, si obiecte externe functiilor, mentinute pe toatã durata executiei unui program. In plus, pot exista si obiecte alocate la cerere, fãrã nume, si adresate printr-un pointer; aceste obiecte durează pânã la distrugerea lor explicitã, prin eliberarea memoriei alocate.

Functia complementara unui constructor, numita "destructor", este apelatã automat la disparitia unui obiect: la terminarea programului pentru obiectele statice, la iesirea dintr-o functie pentru obiectele locale (si parametri functiei), la folosirea operatorului ***delete*** pentru obiecte dinamice. Functia destructor nu are tip, iar numele ei derivã din numele clasei, precedat de caracterul tildã ('~').

Functia destructor este definitã explicit în C++ mult mai rar ca un constructor, deoarece efectul nul al destructorului implicit este suficient pentru obiectele care nu contin date alocate dinamic (în Java nu sunt necesare functii destructor).

Pentru clase ce contin date alocate dinamic, constructorul trebuie sã realizeze si alocarea de memorie necesarã, iar destructorul sã elibereze memoria alocatã de constructor. Exemplu:

```
   // destructor clasa IntArray
 ~IntArray () { delete [ ] vec;}
```

Un apel de metodã de forma

```
   a.add (x);          // apeleaza metoda “add” pentru obiectul “a”
```

este tradus de compilatorul C++ într-un apel de functie C de forma:

```
   add ( &a, x);
```

în care primul argument este adresa obiectului pentru care se apelează functia clasei.

In cadrul unei metode acest argument implicit de tip pointer (adresa obiectului) este accesibil prin cuvântul cheie "this", utilizat ca orice alt pointer. Un exemplu de utilizare este într-un constructor (sau într-o metodã) care are argumente formale cu acelasi nume ca variabile ale clasei:

```
 IntArray (int dmax=10, int inc =10) {
   this->dmax= dmax;
   this->inc= inc; d=0;
   vec= new int[dmax];
 }
```

Exista si alte situatii de programare în care utilizarea variabilei “this” nu poate fi evitată (de exemplu în definirea unor operatori).

Operatori supradefiniti

Prin clase se definesc noi tipuri de date, iar operatiile cu variabile de aceste tipuri sunt realizate prin functii (metode ale clasei sau functii “prieten” în C++). De exemplu adunarea a douã variabile de un tip “Complex” se va face astfel:

```
Complex  a,b,c;
a.add (b);          //  add este metoda a clasei Complex  ( a += b)
c= add (a,b);      // add este functie prieten a clasei Complex  ( c = a+b)
```

In C++ (si în C#) este permisã supradefinirea operatorilor limbajului C, pentru a fi posibilã utilizarea lor cu variabile de tipul clasei în care au fost supradefiniti. Dacã supradefinim operatorii "+" si "=+" în clasa "Complex" atunci putem scrie:

```
Complex  a,b,c;
a += b;              //  operator metoda a clasei Complex
c= a + b;           // operator functie prieten a clasei Complex
```

Se foloseste cuvântul "supradefinire" ("override") si nu "redefinire", pentru cã rãmân valabile definitiile anterioare ale operatorului respectiv, la care se adaugã o nouã definitie (o nouã interpretare, în functie de tipul operanzilor).

Operatorii binari "<<" si ">>" sunt supradefiniti pentru clasele "ostream", "ofstream" si respectiv "istream", "ifstream", pentru a exprima actiunile de "insertie valoare într-un flux de iesire" si respectiv "extragere valoare dintr-un flux de intrare". Ei sunt operatori binari: primul operand este de tip "ostream" ("istream"), iar al doilea operand este de orice tip predefinit (char, int, float, double, char *, etc.). Pentru a fi utilizati si pentru operanzi de un tip clasã (de ex. "Complex" sau "Date"), ei trebuie supradefiniti în aceste clase.

Un operator este privit în C++ ca o functie cu un nume special: numele este format din cuvântul cheie "operator" si caracterul sau caracterele ce reprezintã operatorul. O expresie de forma:

```
a + b               // a si b sunt obiecte de un tip clasa C
```

este consideratã a fi echivalentã cu unul din apelurile urmãtoare:

```
a.operator+ (b)       // operator realizat ca metoda a clasei C
operator+ (a,b)      // operator realizat ca functie separata de clasa C
```

Pentru cã functia operator trebuie sã aibã acces la datele locale clasei ("private"), ea este declaratã în clasa C ca functie prieten (cuvântul cheie "friend").

Un bun exemplu de clasã care necesitã diversi operatori este clasa "Complex" pentru numere complexe. Pentru început vom defini doar operatorul de comparatie la egalitate a douã obiecte de tip complex, ca metodã a clasei :

```
class Complex {
  private:
    double re, im;
  public:
 Complex (double re=0, double im=0) {          // un constructor
    this-> re=re; this->im=im;
 }
 int operator == (Complex z) {                      // comparatie la egalitate
```

```
   return re==z.re && im==z.im;
 }
  void print ( ) { cout<<'('<<re<<','<<im<<')'<<'\n';}
} ;
```

Operatorul "==" poate fi definit si ca functie prieten, iar utilizarea sa va fi la fel ca în cazul definirii ca metodã a clasei. Exemplu:

```
friend int operator == (Complex a, Complex b) {
   return a.re==b.re && a.im==b.im;
}
```

Operatorul de adunare numere complexe "+" definit ca metodã a clasei va modifica obiectul pentru care se apeleazã, deci primul operand; vom prefera de aceea definirea lui printr-o functie prieten :

```
friend Complex operator + (Complex a, Complex b) {
   return Complex (a.re+b.re, a.im+b.im);
}
```

In schimb, operatorul "+=" care modificã primul operand va fi definit printr-o metodã a clasei, cu nume de operator. Exemplu:

```
Complex operator += (Complex  z) {         // in clasa Complex
   re += z.re; im += z.im;
 return * this;
}
```

Operatorul de insertie intr-un flux a unui obiect de tipul Complex nu poate fi definit decât ca functie prieten, deoarece primul operand nu este de tip "Complex" (un operator binar definit ca metodã a clasei "Complex" trebuie sã aibã primul operand de tip "Complex", deoarece acesta este obiectul pentru care se executã metoda operator).

```
ostream& operator << (ostream& s, Complex & z) {
   s << '(' << z.re << ',' << z.im << ')' << '\n';
   return s;
}
```

Rezultatul referintã al operatorului "<<" permite expresii de forma:

```
      cout << z1 << " " << z2 << "\n";        // z1, z2 de tip Complex
```

Se pot supradefini si unii operatori unari, inclusiv operatorul pentru conversie de tip ("cast"). In exemplul urmãtor se redefineste in clasa "IntArray" operatorul de selectie (de indexare), pentru a fi folosit în locul metodei "get":

```
   // definitie din clasa IntArray
int operator[ ] (int k) {                    // valoare element din pozitia i
   assert ( k < d);
   return vec[k];
}
```

Definitia anterioarã permite expresii de forma:

```
cout << a[0];   int x = a[0];
```

dar nu permite o expresie de forma:

```
a[0]= 77;
```

Pentru a utiliza rezultatul operatorului (functiei) în partea stânga a unei atribuiri vom modifica definitia astfel ca functia operator sã aibã rezultat referintã:

```
int& operator[ ] (int k) {                    // valoare element din pozitia i
   assert ( k < d);
   return vec[k];
}
```

Clase cu date alocate dinamic

Copierea de obiecte are loc în urmãtoarele situatii:

- La atribuirea între obiecte de acelasi tip (operatorul de atribuire poate fi folosit între obiecte, indiferent de tipul lor). Exemplu:

```
Date a, b(1,5,2000);        a=b;
```

- La construirea unui obiect pe baza datelor dintr-un alt obiect:

```
Date  b(1,5,2000);  Date a(b);
```

- La transmiterea rezultatului unei functii prin instructiunea “return”. Exemplu:

```
Date fun ( Date d ) {
     // modifica continutul obiectului d …
     return d;      // transmite o copie a obiectului d
}
```

- La înlocuirea unui argument formal cu un argument efectiv. Exemplu:

```
Date q (z,l,a);    fun(q);       // se inlocuieste d prin q (copiere q in d)
```

- La evaluarea unor expresii cu mai multi operanzi ce sunt obiecte. Exemplu:

```
Complex a,b,c,d; …  d = a+b+c;
```

Pentru aceste situatii compilatorul C++ generează automat un constructor prin copiere, cu acelasi efect ca atribuirea între obiecte: copierea bit cu bit a datelor.

Copierea obiectelor care contin pointeri la date alocate dinamic creează urmãtoarea problemã: prin copierea valorii unui pointer se ajunge la situatia în care douã obiecte diferite folosesc o aceeasi zonã de date, adresatã de acel pointer.

Fie două obiecte “a” si “b” de tipul “IntArray”; după atribuirea a=b se ajunge ca cele două obiecte să conțină adresa aceluiasi vector (numită “vec”), iar modificări în vectorul din obiectul “b” se vor reflecta si în obiectul “a”. Exemplu:

```
IntArray a , b;
a.add(1); b.add (2);
a = b;
a.print ();       // scrie 2
b.set(0,-5);    // pune in pozitia 0 din b valoarea -5
a.print();        // scrie -5
```

Copierea bit cu bit a datelor dintr-un obiect într-un alt obiect se numeste si copiere superficială (“shallow copy”); copierea profundă (“deep copy”) foloseste fiecare pointer dintr-un obiect pentru a copia datele adresate prin acel pointer, astfel ca obiecte distincte să folosească zone de date diferite (cu acelasi continut imediat după copiere). In C++ copierea profundă se face prin redefinirea operatorului de atribuire si prin definirea unui constructor prin copiere (“copy constructor”). Exemple:

```
  // constructor prin copiere pentru obiecte vector de intregi
IntArray:: IntArray ( IntArray & a) {
    dmax= a.dmax; d = a.d;   // copiere variabile intregi
    vec= new int [dmax];   // aloca memorie pentru vectorul din noul obiect
    for (int i=0;i<d;i++) // copiere elemente vector
       vec[i]=a.vec[i];
 }
  // operator de atribuire supradefinit pentru clasa IntArray
IntArray & operator = (IntArray & v) {       if (dmax != v.dmax) {           delete [ ] vec;
dmax=v.dmax;          vec= new int [dmax];   // aloca memorie pentru vectorul din
noul obiect
     }
     d=v.d;                          // numar de elemnte in vector
     for (int i=0;i<d;i++)              // copiere elemente vector
       vec[i]=v.vec[i];
     return * this;                  // rezultatul este obiectul modificat prin atribuire
    }
```

In Java si în C# operatia de copiere a unui obiect se numeste clonare si este realizată printr-o functie; atribuirea directă între obiecte nu este posibilă doarece variabilele de un tip clasă sunt referinte la obiecte si nu sunt nume pentru obiecte.

Clase derivate

In practică pot apare clase care au mai multe metode comune (cu acelasi efect) dar si metode care diferă (cu nume diferite sau cu acelasi nume). Un exemplu este o clasă "IntArray" pentru obiecte ce reprezintă vectori extensibili (realocati dinamic) si o clasă "ArraySet" pentru multimi de întregi realizate ca vectori.

Diferenta dintre cele douã clase este aceea cã un obiect de tipul "ArraySet" nu poate avea douã sau mai multe elemente egale (egale dupã un anumit criteriu). Evitarea duplicatelor se poate face în metoda de adãugare a unui element la o multime “add” care va avea implementãri diferite în clasele "IntArray" si "AraySet".

Intr-o astfel de situatie se va declara clasa "ArraySet" ca o subclasã derivatã din "Array" (In Java si în C# se spune cã "ArraySet" extinde clasa "IntArray").

```
  // multime de intregi realizata ca vector
class ArraySet : public IntArray {
public:
  ArraySet (int max=10, int incr=10): IntArray (max,incr) { }
  int contains (int x) {            // o metoda noua
    return indexOf(x) >=0;     // metoda indexOf mostenita de la IntArray
  }
  void add (int x) {        // o metoda redefinita
    if ( ! contains(x))
      IntArray::add(x);          // apel metoda cu acelasi nume din clasa IntArray
  }
};
   // exemplu de utilizare
void main () {
    ArraySet a;
     for (int x=0; x< 100;x++)
         a.add (x%4);      // adauga numai valorile 0,1,2,3
    a.print();                 // utilizare metoda mostenita de la superclasa
}
```

Uneori vrem sã refolosim o parte din metodele unei clase existente, cu adãugarea sau modificarea altor metode. De exemplu, putem adãuga clasei ArraySet operatii de reuniune, intersectie si diferentã de multimi.

O clasã D, derivatã dintr-o alta clasã B (clasa de bazã), mosteneste de la clasa B datele si functiile publice (cu câteva exceptii), dar poate sã adauge date si/sau functii proprii si sã redefineascã oricare dintre metodele mostenite, în functie de cerintele clasei derivate D. Nu se mostenesc functiile constructor si destructor.

O clasã derivatã poate servi drept clasã de bazã pentru derivarea altor clase. Clasa de bazã se mai numeste si superclasã, iar clasa derivatã se mai numeste si subclasã. Derivarea este un proces de specializare a claselor; clasa vector este mai generalã decât clasa multime-vector pentru cã un vector poate contine si elemente egale. O multime-vector este un caz particular de vector. La fel, o multime ordonatã este un caz particular de multime-vector si o vom defini ca o clasã derivatã din “ArraySet”.

La construirea unui obiect dintr-o clasã derivatã D se apeleazã mai întâi constructorul clasei de baza B si apoi constructorul clasei derivate D.

La distrugerea unui obiect dintr-o clasã derivatã D se apeleazã mai întâi destructorul clasei derivate D si apoi destructorul clasei de baza B.

In C++ constructorul unei clase derivate are o sintaxã specialã care sã permitã transmiterea de date la constructorul clasei de baza, executat înaintea sa:

```
D (tip x): B (x) { ... }
```

Relatia dintre o clasã derivatã D si clasa din care este derivatã B este o relatie de forma "D este un fel de B" (“is a kind of “). Deci o multime vector este un fel de vector. La fel, o stivã vector este un fel de vector (un caz particular de vector).

Pe lângã reutilizarea metodelor din superclasã în subcasã, derivarea creeazã tipuri compatibile si ierarhii de tipuri. Tipul unei clase derivate este subtip al tipului clasei din care derivã, asa cum tipul "int" poate fi considerat ca un subtip al tipului "long", iar tipul "float" ca un subtip al tipului “double”.

Tipurile clasã obtinute prin derivare, ca si tipurile numerice din C, nu sunt independente între ele, formând o familie (o ierarhie) de tipuri compatibile. Un avantaj este acela cã functiile matematice din C (sqrt, exp, cos, etc.) au o singurã formã, cu argumente formale de tip “double”, dar pot fi apelate cu argumente efective de orice tip numeric. Conversia de la un tip mai general la un subtip se face automat în C si C++, dar în Java si C# numai folosind operatori de conversie “cast”. Exemplu:

```
int x; double d=3.1415 ;    x = (int) d;     // în Java
```

In C++ este permis ca un obiect de un tip mai general sã primeascã prin atribuire (sau sã fie înlocuit, ca argument de functie) un obiect de un subtip al sãu. Exemplu:

```
IntArray a;  ArraySet s;
a = s ;                  // “upcasting” = conversie in sus
```

Justificarea este, în acest caz, cã orice multime vector poate fi consideratã ca un vector general. Trecerea inversã, de la un vector oarecare la o multime, nu este însã posibilã decât dacã existã în subclasa “ArraySet” un constructor cu argument de tip “IntArray”, în care se asigurã unicitatea elementelor din multimea nou creatã.

Reguli similare se aplicã la conversia de pointeri cãtre tipuri compatibile :

```
IntArray * ap; ArraySet * sp;
ap = sp;         // conversie implicitã (in sus)
sp = (ArraySet*) ap;      // conversie explicitã (in jos) = downcasting
```

Desi este posibilã sintactic, conversia în jos (la un subtip) poate conduce la efecte nedorite, dacã nu este folositã corect: dacã adãugãm vectorului adresat de “ap” mai multe elemente egale, acestea nu vor fi eliminate dupã conversia de pointeri :

```
IntArray * ap;    ArraySet * sp;
for (int x=1;x<10;x++)
   ap->add( x%4);           // metoda add” din “IntArray”
sp = (ArraySet *) ap;
sp->print();        // scrie 1,2,3,0,1,2,3,0,1
```

Conversia în jos de pointeri (sau de referinte) este totusi mult folositã în programarea orientatã pe obiecte, asa cum se va arãta mai departe. De exemplu, putem scrie o functie de ordonare cu argument “IntArray *” care sã poatã fi folositã si pentru a ordona o multime de tip “ArraySet” sau un alt caz particular de vector.

In general, dacã D este o subclasã a clasei B, atunci conversia de la tipul D* (sau D&) la tipul B* (sau B&) se face automat, dar conversia inversã, de la B* la D* este posibilã numai folosind operatorul de conversie ("cast"). Explicatia este aceea cã un obiect de tip D contine toate datele unui obiect de tip B, deci folosind o adresã de obiect D putem accesa toate datele obiectului B. Reciproca nu este adevãratã, deoarece în subclasa D pot exista date si metode inexistente în superclasa B si care nu sunt accesibile printr-un pointer la tipul B. Folosirea operatorului de conversie obligã programatorul sã confirme cã printr-o variabilã de tip “B*” se referã de fapt la un obiect D si nu la un obiect de tip B.

Ierarhii de tipuri si polimorfism

Posibilitatea de a înlocui o variabilã de tip B* prin orice variabilã de un tip D* (clasa D fiind derivatã direct sau indirect din B) a condus la crearea unor familii de clase, de forma unui arbore care are ca rãdãcinã o clasã de bazã cu foarte putine functii si fãrã date, din care sunt derivate direct sau indirect toate celelalte clase din familie. In felul acesta se creeazã o ierarhie de tipuri compatibile.

Dacã se defineste un tip clasã foarte general (cum este tipul "Object" din Java sau din C#) si toate celelale clase sunt derivate direct sau indirect din clasa "Object", atunci o colectie de pointeri la obiecte de tip "Object" este o colectie generică, pentru cã poate memora adrese de obiecte de orice alt tip (subtipuri ale tipului "Object"). In C++ nu sunt permisi vectori cu componente de un tip referintã (dar în Java se folosesc frecvent vectori de referinte).

O colectie de pointeri la un tip mai general poate fi folositã pentru a memora adresele unor obiecte de orice subtip, asa cum o colectie de pointeri “void *” poate fi folositã pentru a memora adrese de întregi, de reali, de siruri, de structuri, etc. La extragerea din colectie se face conversie la subtipul cunoscut de programator, pentru a putea folosi metodele subclasei.

Vom folosi ca exemplu o colectie vector, cu dimensiune fixã pentru simplificare:

```
      // clasa de baza a ierarhiei
class Object {
public:
  void print( ) {cout << this;}                    // scrie adresa acestui obiect
  int equals(Object * obj) { return this == obj;}      // compara adrese obiecte
} ;

    // vector de obiecte
class Array : public Object {
protected:              // pentru acces in clase derivate
  Object ** vp;         // adresa vector de pointeri la obiecte
```

```
  int dmax;              // dimensiune vector
  int d;                 // prima pozitie libera si numar de elemente
public:
  Array (int size) { vp=new Object * [dmax=size]; d=0; }
  void print () {                          // afisare vector
     for (int i=0; i< d; i++) {      // repeta ptr fiecare element din vector
       vp[i]->print();                 // apelul metodei “print” pentru obiectul de la adresa
vp[i]
       cout <<"\n";
     }
 }
  int size() { return d;}                  // dimensiune vector
  void add (Object * ptr) { vp[d++]= ptr; }
  int contains (Object * ptr) {            // daca vectorul contine obiectul cu adr. ptr
    for (int i=0;i<d;i++)
      if ( vp[i]->equals(ptr))      // compara obiectele de la adrele vp[I] si ptr (nu
adresele lor)
        return 1;
    return 0;
  }
};
```

Pentru verificare vom defini douã clase derivate din clasa “Object”:

```
   // obiecte numere intregi
class Int: public Object {
  int val;
public:
  Int (int vi) {val=vi;}
  void print () { cout << val << ' ';}
  int equals (Object * obp) {
    Int * p = (Int*) obp;
    return val == p->val;
  }
  int getVal () { return val;}
};
     // obiecte cu date calendaristice
class Date: public Object {
 int zi,luna,an;
public:
 Date (int zi, int luna, int an) {
   this->zi=zi; this->luna=luna; this->an=an;
 }
 int equals ( Object * obp) {
   Date * p = (Date*) obp;
   return zi==p->zi && luna == p->luna && an==p->an;
 }
 void print() {
    cout << zi << '-' << luna << '-' << an << '\n';
 }
```

```
};
```

Vom folosi clasa “Array” pentru obiecte de douã tipuri diferite, ambele derivate din tipul “Object”: clasele “Int” si “Date” :

```
void main () {
  Array a(10);
  Date * d2 = new Date (31,12,2004);
  Date * d1 = new Date (1,1,2005);
  a.add (d1);  a.add(d2);
  a.print();
  a.add (new Int(5));   a.add (new Int(7));
  a.print();
}
```

Utilizarea clasei “Array” aratã cã metoda “print” afiseazã adresele obiectelor reunite într-un obiect “Array” (si nu datele din aceste obiecte), iar metoda “contains” nu dã rezultate corecte. Explicatia este aceea cã metoda “print” din clasa “Array” apeleazã mereu metoda “print” din clasa “Object”, indiferent de tipul real al obiectelor.

O solutie ar fi ca sã determinãm la executie tipul obiectelor memorate si sã apelãm metoda de afisare corespunzãtoare acestui tip. In acest scop am putea adãuga clasei “Object” si subclaselor sale o metodã “getType” având ca rezultat tipul obiectelor (un nume sau un numãr unic pentru fiecare clasã).

Solutia aplicatã în C++ si în toate limbajele orientate pe obiecte care au urmat foloseste functii virtuale (sau polimorfice) în locul determinării tipului la executie.

O metodã declaratã ***virtual*** în clasa de bazã si apelatã prin intermediul unui pointer sau printr-o referintã la clasa de bazã, produce executia unei secvente diferite, în functie de tipul pointerului sau referintei (de tipul obiectului referit). Exemplu:

```
class Object {
public:
  virtual void print( ) {cout << this;}                    // scrie adresa acestui obiect
  virtual int equals(Object * obj) { return this == obj;}      // compara adrese obiecte
} ;
// efectul apelului unei functii virtuale
  void main () {
     Date d (1,1,2005); Date & dr=d;  Date * dp= &d;
     Object obj= d; Object& obr= dr;  Object* obp=  dp;
     obj.print();         // apeleaza functia Object::print()
     obr.print ();       // apeleaza functia Date::print()
     obp->print();     // apeleaza functia Date::print()
  }
```

In clasele derivate nu mai trebuie folosit cuvântul ***virtual*** la redefinirea metodelor virtuale, deci definitiile anterioare ale claselor “Date” si “Int” nu se modificã.

Compilatorul tratează diferit metodele virtuale de celelalte metode: fiecare obiect dintr-o clasã cu metode virtuale contine, pe lângã date, si adresa unui tabel de metode

virtuale (unul singur pentru o clasã), care este un vector de pointeri. Un apel de metodã virtualã este tradus printr-un apel indirect prin adresa tabelului de metode virtuale, iar un apel de metodã nevirtualã este tradus printr-un apel la o adresã fixã, stabilitã de compilator pentru metoda respectivã.

Altfel spus, adresa functiei apelate este stabilitã la compilare pentru orice metodã nevirtualã ("legare timpurie"), dar este stabilitã la executie pentru orice metodã virtualã ("legare târzie"). Functia virtualã este apelatã printr-un pointer (referintã); acel pointer trimite cãtre un obiect, iar acel obiect trimite cãtre tabelul de metode virtuale al clasei de care apartine si deci la implementarea specificã clasei respective.

Functiile polimorfice (virtuale) au sens numai într-o ierarhie de clase (de tipuri).

Metodele virtuale împreunã cu colectiile de pointeri la tipul generic "Object" permit realizarea de colectii generice, care pot reuni obiecte de orice tip si chiar de tipuri diferite într-o aceeasi colectie. Putem deci sã avem o singurã clasã "Array" pentru vectori de pointeri la obiecte, indiferent de tipul acestor obiecte, cu conditia ca el sã fie subtip al tipului "Object". Clasa anterioarã "Int" aratã cã pentru a memora date de un tip primitiv într-un astfel de vector generic trebuie sã includem tipul primitiv într-o clasã derivatã din "Object" (operatie numitã si "boxing"). In Java si C# existã clase anvelopã pentru toate tipurile de date primitive (char, int, long, float, double, s.a.) .

Clase abstracte

O superclasã poate fi atât de generalã încât sã nu se poatã preciza implementarea unor metode, care pot fi însã declarate ca tip, parametri si efect (ca mod de utilizare). Astfel de metode se numesc metode abstracte, iar o clasã cu metode abstracte este o clasã abstractã, care nu poate genera obiecte (nu se poate instantia).

Rolul unei clase abstracte este numai acela de a servi ca bazã pentru derivarea altor clase, deci ca tip comun mai multor subtipuri.

In C++ o metodã abstractã este o metodã virtualã purã, declaratã astfel:

```
// metode virtuale pure (abstracte)
virtual void print () =0;          // nu este o atribuire, este un nod de declarare
virtual int equals (Object *) =0;
```

O metodã abstractã poate rãmâne abstractã într-o subclasã, sau poate fi definitã, cu acelasi tip si cu aceleasi argumente ca în superclasã.

O metodã abstractã este altceva decât o metodã cu definitie nulã (fãrã efect) :

```
// metode virtuale cu definitie nulã (neabstracte)
virtual void print () { }
virtual int equals (Object *) { }
```

O metodã abstractã din superclasã trebuie implementatã într-o subclasã pentru a crea obiecte, iar aceastã obligatie poate fi verificatã de cãtre compilator. Compilatorul nu poate însã verifica dacã o metodã cu definitie nulã în superclasã a fost sau nu redefinitã în subclase.

In concluzie, o altã definitie posibilã pentru clasa “Object” ar fi urmãtoarea:

```
   // clasa abstractã pentru obiecte generice
class Object  {
public:
  virtual void print() =0;
  virtual int compare (Object* obj) =0;   // cu rezultat negativ, pozitiv sau zero
} ;
```

In C++ nu se face deosebire între o clasã care contine numai metode abstracte si o clasã care mai contine si date sau metode implementate. In C# si în Java se foloseste notiunea de “interfatã” pentru o clasã care contine numai metode abstracte.

O interfatã stabileste un mod de utilizare comun pentru toate subclasele sale (clase care implementeazã interfata) sau un contract pe care clasele compatibile cu interfata se obligã sã-l respecte (prin definirea tuturor metodelor abstracte din interfatã).

In programarea orientatã pe obiecte se poate face o separare netã între interfatã si implementare, evidentã în cazul tipurilor abstracte de date. Un tip abstract de date este definit prin operatiile posibile cu obiecte de acel tip si nu presupune nimic despre modul de implementare.

Un exemplu este tipul abstract “multime”, definit ca o colectie de elemente distincte cu operatii de adãugare element la multime, verificare apartenentã la o multime s.a. Conceptul matematic de multime poate fi implementat printr-un vector cu elemente distincte sau printr-un arbore binar sau printr-un tabel de dispersie etc. In Java existã o interfatã “Set” pentru operatii cu orice multime si câteva clase instantiabile care implementeazã interfata : “TreeSet” si “HashSet”. La acestea se pot adãuga si alte implementãri de multimi (vector, listã înlãntuitã), dar cu aceleasi operatii.

Se recomandã ca o aplicatie cu multimi sã foloseascã variabile si argumente de functii de tipul general “Set” si numai la crearea de obiecte multime se va preciza tipul de multime folosit. In acest fel se poate modifica usor implementarea multimii, fãrã a interveni în multe puncte din aplicatie. In plus, se pot scrie algoritmi generici, deci functii care pot prelucra orice multime, indiferent de implementare.

Agregare si delegare

Reutilizarea functionalitãtii unei clase C într-o altã clasa E se poate face si prin includerea unui obiect de tip C în clasa E; metodele clasei E vor apela metodele clasei C prin intermediul obiectului continut. Relatia dintre clasele E si C este de tipul "E contine un C" (“has a”). Exemplu:

```
  // clasa multime-vector, prin compunere
class ArraySet {
  IntArray vec;
public:
  ArraySet (int max=10, int incr=10): vec(max,incr) { }
  int contains (int x) {
```

```
    return vec.indexOf(x) >=0;
  }
  void add (int x) {
   if ( ! contains(x))
     vec.add(x);
  }
  void print () {
    vec.print();
  }
};
```

A se observa modul de initializare a obiectului continut de către constructorul clasei compuse. Metodele clasei compuse (sau o parte din ele) sunt implementate prin apelarea unor metode din clasa obiectului continut; se numeste cã operatiile “print”, “add” s.a. sunt delegate (transmise spre executie) obiectului vector, sau cã obiectul vector este delegat sã realizeze operatiile cerute unui obiect multime-vector.

Prin compunere (agregare), clasa compusã poate avea metode diferite ca nume si ca argumente de clasa continutã, chiar dacã se refolosesc operatii implementate în metodele clasei continute. Altfel spus, derivarea păstreazã interfata clasei de bazã (cu eventuale modificări), iar compunerea poate prezenta o interfatã complet diferitã pentru clasa compusa. Exemplu:

```
 // clasa stiva de intregi
class Stack {
  IntArray st;
public:
  Stack (int ni=10, int inci=10): st(ni,inci) {};
 void push (int x) {      // pune x pe stiva
   st.add(x);
 }
 int pop () {              // scoate din varful stivei
   int sz=st.size();
   assert ( sz >0);
   int x= st[sz-1];
   st.remove();       // metoda din clasa IntArray
   return x;
 }
 int empty () { return st.size()==0;}       // verifica daca stiva e goala
} ;
```

Definitia anterioarã corespunde afirmatiei cã "o stivã este un obiect care contine un obiect vector". O clasã stivã poate fi definitã si ca subclasã a clasei vector “IntArray” considerând cã "o stivã este un fel de vector" (un caz particular de vector). Totusi, o stivã nu foloseste aproape nimic din interfata unui vector, deci ar fi preferabilã compozitia în acest caz.

De observat cã în ambele cazuri un obiect stivã va contine un obiect vector, pentru cã un obiect stivã contine datele clasei sale, date mostenite dela superclasã si care sunt adresa si dimensiunea vectorului (la fel ca si în cazul agregãrii).

Relatia stabilitã prin derivare este o relatie staticã între clase (nu mai poate fi modificatã la executie), dar relatia stabilitã prin agregare este o relatie dintre obiecte si poate fi o relatie dinamicã, dacã variabila continutã în clasa compusã este de un tip mai general (clasã abstractã sau interfatã). Variabila continutã (obiect sau pointer la obiect) poate primi la executie, prin atribuire, variabile de diferite subtipuri.

Un alt exemplu de reutilizare a functionalitãtii unei clase prin compunere poate fi o clasã pentru dictionare implementate printr-un singur vector extensibil. Metodele clasei dictionar pot folosi metode ale clasei vector: adãugare la vector (cu extindere, la nevoie), afisare vector, cãutare în vector s.a.

Cele mai importante metode de lucru cu un dictionar sunt:

```
Object * put (Object* key, Object* value); // pune o pereche in dictionar
Object* get (Object * key);              // obtine valoarea asociata unei chei date
```

Un dictionar este o colectie de perechi cheie-valoare, în care cheia si valoarea pot avea orice tip dar cheile trebuie sã fie distincte (la comparatie cu metoda “equals”). O solutie posibilã foloseste o clasã auxiliarã “Pair” pentru o pereche de obiecte si un vector în care se introduc pointeri la obiecte “Pair”.

Exemplul urmãtor este o transcriere în C++ a unei situatii uzuale din Java, când se defineste o colectie ordonatã de obiecte generice comparabile. Ordonarea foloseste operatia de comparare, dar aceasta depinde de tipul obiectelor din colectie: altfel se comparã date calendaristice si altfel se comparã siruri de caractere sau numere. Este deci necesar ca la crearea unui obiect vector ordonat sã se transmitã si obiectul comparator specific tipului de obiecte introduse în vector. Clasa pentru vectori ordonati contine o variabilã pointer la tipul abstract “Comparator”, initializatã de constructor cu obiectul concret comparator (dintr-o subclasã a clasei “Comparator”). Obiectul concret comparator folosit de obiectul vector este stabilit la executie, iar relatia dintre aceste obiecte este o relatie dinamicã.

```
  // clasa ptr vectori ordonati de obiecte oarecare
class SortedArray: public Array {
private:
  Comparator* comp;      // comparator de obiecte memorate
  void sort();
public:
  SortedArray (int size, Comparator* cmp): Array(size) { comp=cmp;}
  void add (Object * p) {
    Array::add(p);
    sort();
  }
};
```

```
  // ordonare vector de obiecte
void SortedArray::sort () {
    for (int j=1;j<d;j++)
      for (int i=0;i<d-1;i++)
        if ( comp->compare(vp[i],vp[i+1]) > 0) {
          Object* b=vp[i];
          vp[i]=vp[i+1];
          vp[i+1]=b;
        }
}
```

Deoarece “Comparator” este o clasã abstractã (o interfatã, de fapt), nu putem avea o variabilã “Comparator” (nu pot exista obiecte de acest tip), dar putem avea o variabilã pointer la tipul “Comparator”, ca loc unde vom memora un pointer la un obiect de un subtip al tipului “Comparator”. Exemplu de comparator pentru obiecte de tip “Int”:

```
   // clasa abstracta ptr obiecte comparator
class Comparator {
 public:
  virtual int compare (Object*,Object*)=0;
};
   // comparator pentru obiecte Int
class IntComp: public Comparator {
public:
  int compare (Object* p, Object* q) {
    Int* ip = (Int*)p;
    Int* iq = (Int*)q;
    return ip->getVal() - iq->getVal();
  }
};
```

Exemplu de creare a unui obiect vector ordonat:

```
SortedArray a (100, new IntComp());
```

Este posibilã si chiar folositã combinarea derivării cu agregarea, ca metode de reutilizare a metodelor unor clase. Am putea avea, de exemplu, o clasã abstractã “Stack” (pentru stive abstracte), cu metode “push” si “pop”, iar una din subclasele sale instantiabile ar contine un obiect vector de obiecte (stive realizate ca vectori).

Clase sablon (“Templates”)

In programarea cu obiecte mai existã si o altã solutie pentru programarea generică a unor colectii în care să putem memora date de orice tip (tip primitiv sau tip clasã). Această solutie se bazează pe clase “sablon” (“template”), clase pentru care se poate preciza la instantiere tipul datelor continute. La definirea clasei, tipul datelor folosite

apare ca un parametru formal (pot fi si mai multi parametri tip de date) si de aceea se mai numesc si clase cu tipuri parametrizate (cu parametri tipuri de date).

In C++, cuvântul cheie "template" precede o definitie de functie sau de clasã în care unul sau mai multe tipuri de date sunt neprecizate. Tipurile neprecizate sunt date ca parametri, precedati fiecare de cuvântul cheie "class" si încadrati între paranteze unghiulare ‘<’ si ‘>’.

Exemplu de functie sablon pentru determinarea valorii minime dintre douã variabile de orice tip T, neprecizat la definirea functiei:

```
  // definire functie sablon cu un parametru "class"
template <class T> T min (T a, T b) {        // T poate fi orice tip definit anterior
  return a < b? a: b;
}
```

Cuvântul "class" aratã cã T este un parametru ce desemneazã un tip de date si nu o valoare, dar nu este obligatoriu ca utilizarea functiei "min" sã foloseascã un parametru efectiv de un tip clasã (în Java nu mai este necesar “class”). In functia “min” tipul care va înlocui tipul neprecizat T trebuie sã cunoasca operatorul '<' cu rol de comparatie la mai mic. Exemplul urmãtor aratã cum se poate folosi functia sablon "min" cu câteva tipuri de date primitive:

```
  // utilizari ale functiei sablon
void main () {
  double x=2.5, y=2.35 ;
  cout << min (x,y);     // min (double,double)
  cout << min (3,2);     // min (int,int)
  cout << min (&x,&y);   // min (double*,double*)
}
```

O clasã sablon este o clasã în a cãrei definire se folosesc tipuri de date neprecizate pentru variabile si/sau pentru functii membre ale clasei. Toate tipurile neprecizate trebuie declarate într-un preambul al definitiei clasei, care începe prin cuvântul cheie "template" si este urmat, între paranteze ascutite, de o lista de nume de tipuri precedate fiecare de cuvântul "class". Exemplu:

```
    // o clasa vector cu componente de orice tip T
template <class T> class Vector {
  T* vec;              // adresa vector
  int d, dmax;       // dimensiune vector
public:
  Vector (int sz=10) { vec= new T [dmax=sz]; d=0;}
  void add (T elem) {
    assert ( d<dmax); vec[n++]=elem;
  }
  T& operator [ ] (int i) {
    assert (i<d); return vec[i];
  }
```

```
  void print ();   // definita in afara clasei
};
```

La declararea unei variabile de tip clasã sablon trebuie precizat numele tipului efectiv utilizat, între paranteze ascutite, dupã numele clasei. Exemplu:

```
   // utilizari ale clasei sablonvoid  main () {  Vector<int> a (20);       // a este vector
de intregi  Vector<Int> x;                  // x este vector de obiecte Int
  for (int i=0;i<10;i++) {
     a.add (i);   x.add ( * new Int(i) );
  }
}
```

Pentru metodele definite în afara clasei trebuie folosit cuvântul "template".Exemplu:

```
template <class T>   void Vector<T> :: print () {     for (int i=0;i<n;i++)
        cout << vec[i] << ' ';              // daca se poate folosi << ptr acel tip
  }
```

Pe baza definitiei clasei sablon si a tipului parametrilor efectivi de la instantierea clasei, compilatorul înlocuieste tipul T prin tipul parametrilor efectivi. Pentru fiecare tip de parametru efectiv se generează o altă clasă , asa cum se face expandarea unei macroinstructiuni (definitã prin "define"). Definitia clasei este folositã de compilator ca un "sablon" (tipar, model) pentru a genera mai multe definitii de clase "normale". De exemplu, declaratia

```
         Vector<int> a;
```

va genera o definitie de forma urmãtoare:

```
class Vector_int {         // numele generat ptr clasa depinde de compilator    int * vec;
int n,nmax;public:   int& operator [ ] (int i) { ... }
  void add (int elem) { ... }
  void print();
};
```

De observat cã o clasã colectie sablon contine obiecte si nu pointeri la obiecte, ca în cazul colectiilor generice bazate pe ierarhii declase.

Biblioteca STL (Standard Template Library) contine clase colectie sablon si clase iterator asociate, precum si o serie de algoritmi generali pentru clase colectie, sub formã de functii sablon. Clasele colectie (container) STL sunt împãrtite în douã:

- Secvente de obiecte ("sequences"), deci diferite tipuri de liste:
  vector, list, deque , stack, queue, priority_queue
- Asocieri (dictionare) si multimi :
  set, multiset , map, multimap

Clasa "vector" corespunde unui vector extensibil, clasa "list" corespunde unei liste înlãntuite; clasele "stack" (stiva), "queue" (coada) si "priority_queue" (coada cu prioritãti) sunt construite pe baza claselor secventã de baza, fiind liste particulare.

Pentru toate clasele secventã se poate folosi operatorul [] pentru selectia elementului dintr-o pozitie datã. Exemplu de folosire a clasei “vector”:

```
#include <vector>#include <string>#include <iostream>using namespace std;int main () {  vector<string > vs (10);        // clasa “string” este predefinita in STL
  char cb[30];                        // aici se citeste un sir
  while (cin >> cb) {
    string str= *new string(cb);
    vs.push_back (str);           // adauga la sfârsitul sirului
  }
    // afisare vector
  for (int i=0;i<vs.size();i++)   // “size” este metoda a clasei “vector”
     cout << vs[i] <<',';           // operatorul [ ] supradefinit in clasa “vector”
  cout <<endl;
}
```

Orice secventã are definit un obiect iterator, folosit la fel ca un pointer pentru a parcurge elementele colectiei. Exemplu:

```
   // afisare vector, cu iterator    vector<string>::iterator it;          // declarare obiect iterator ptr vector de siruri   for (it=vs.begin(); it!=vs.end();it++)        // “begin”, “end” metode ale clasei vector
     cout << *it << ' ,' ;                    // afisare obiect de la adresa continutã în it
  cout << endl;
```

Alte tehnici de programare orientate pe obiecte

Principalele caracteristici ale POO au fost considerate initial a fi încapsularea, mostenirea si polimorfismul. Ulterior au apãrut si alte teme specifice POO, cum ar fi separarea interfetei de implementare, compunere si delegare, relatii între clase si obiecte, colectii generice de obiecte, comunicarea prin evenimente între obiecte de tip observator-observat, spatii de nume, refactorizarea programelor cu obiecte, fabrici de obiecte si alte scheme de proiectare cu clase (“design patterns”).

Pentru un program cu clase existã întotdeauna mai multe solutii, diferite între ele prin numãrul si tipul obiectelor folosite, dar si prin relatiile dintre obiecte si clase. Se considerã drept criteriu principal de calitate a unui program cu obiecte usurinta si siguranta de extindere si de adaptare a aplicatiei la noi cerinte, fãrã a introduce erori. Aceastã cerintã se numeste si “proiectare în perspectiva schimbãrii”.

Alte criterii sunt posibilitatea de reutilizare a unor solutii (scheme) de proiectare în alte aplicatii si performantele aplicatiei. Schemele de proiectare (“design patterns”) sunt solutii verificate sau solutii optime (“best practices”) de realizare a unor cerinte sau situatii de programare comune mai multor aplicatii.

Cel care dezvoltã o aplicatie cu obiecte are la dispozitie biblioteci de clase, iar pentru anumite pãrti din aplicatii (cum ar fi interfata graficã cu utilizatorii) poate folosi infrastructuri de clase (“framework”), care impun un anumit stil de proiectare.

Din punct de vedere practic, timpul de punere la punct a unei aplicatii cu obiecte poate fi mult redus prin utilizarea unui mediu integrat de dezvoltare (IDE), sau a unui mediu vizual pentru dezvoltarea rapidã de aplicatii cu interfatã graficã.

Aici nu am mentionat nimic despre programarea interfetelor grafice deoarece în C++ este dependentã de sistemul de operare gazdã si este relativ complicatã.

Pentru reprezentarea graficã a relatiilor dintre clasele si obiectele unei aplicatii sau a unei pãrti dintr-o aplicatie s-au propus simboluri unice, în cadrul standardului UML (Unified Modelling Language), dar încã se folosesc în literatura de specialitate multe reprezentãri nestandard, mai simple si mai sugestive.