Programarea Calculatoarelor
Curs 9

Iulian Năstac

Cap. Programarea procedurală

- **Programarea procedurală** este o caracteristică a limbajelor de programare de nivel înalt. Dacă se doreşte executarea unui anumit set de instrucţiuni, cu date diferite sau în locuri diferite, acestea se grupează într-o subrutină care va fi apelată printr-un salt ori de câte ori este nevoie. Acest salt este cu revenire la instrucţiunea următoare celei din care s-a făcut saltul şi de aceea el diferă de salturile realizate cu instrucţiunea *goto.*

- Acest tip de secvenţă cu organizarea de mai sus poartă diverse denumiri în cadrul limbajelor de programare: *subprogram*, *subrutină*, *procedură*, *funcţie*, etc.

Recapitulare

În limbajul C folosirea funcţiilor în cadrul unui program implică:

- declararea funcţiilor;

- definirea funcţiilor.

Recapitulare

Logica scrierii unui program în “C” este următoarea:

- includerile de fişiere header;
- declaraţiile de variabile şi constante globale;
- declaraţiile de funcţii;
- definirea funcţiei *main*;
- definirea tuturor funcţiilor declarate.

3. Sfera de influenţă a funcţiilor

Definiţie: *sfera de influenţă* a unui limbaj înseamnă totalitatea regulilor ce stabilesc modul în care o secvenţă de cod influenţează sau are acces la o altă secvenţă de cod sau de date.

În limbajul C, în cadrul *sferei de influenţă* sunt strict respectate următoarele **principii**:

1. Fiecare funcţie conţine un bloc de cod discret propriu acelei funcţii şi nici o instrucţiune din altă funcţie nu poate să aibă acces la el decât printr-un apel al funcţiei. Nu se poate folosi ***goto*** pentru salt între funcţii.

2. Dacă o funcţie nu utilizează variabile sau date globale ea nu poate afecta alte părţi ale programului. Codul şi datele dintr-o funcţie nu pot interacţiona direct cu codul şi datele din altă funcție.

3. În *C toate funcţiile au acelaşi nivel al sferei de influenţă.* C nu este tehnic vorbind un limbaj structurat în blocuri, deoarece nu poate fi definită o funcţie în altă funcţie (dar o funcţie o poate apela pe alta).

Principii (continuare)

4. Variabilele care sunt definite într-o funcţie sunt numite *variabile locale*. O variabilă locală este creată atunci când se intră în acea funcţie şi este distrusă la ieşire. Deci variabilele locale nu îşi păstrează valoarea între apelările funcţiei. Singura excepţie sunt variabilele declarate cu *static* care nu se distrug la părăsirea funcţiei dar sunt limitate ca sferă de influenţă la interiorul funcţiei.

Principii (continuare)

5. Vrem să apelăm o funcţie dintr-o altă funcţie printr-un argument ce este o variabilă.

Exemplu:

```
tip f1 (    )
{ tip x;
   ...
   f2(x);
   ...
}
```

Funcţiei *f2* îi este transmisă o copie a valorii argumentului. Ceea ce se întâmplă în interiorul funcţiei *f2* nu are efect asupra variabilei folosite pentru apelare (adică *x* din funcţia *f1*).

6. Funcţii diferite pot avea variabile locale cu nume identice ceea ce nu duce la influenţarea reciprocă a funcţiilor.

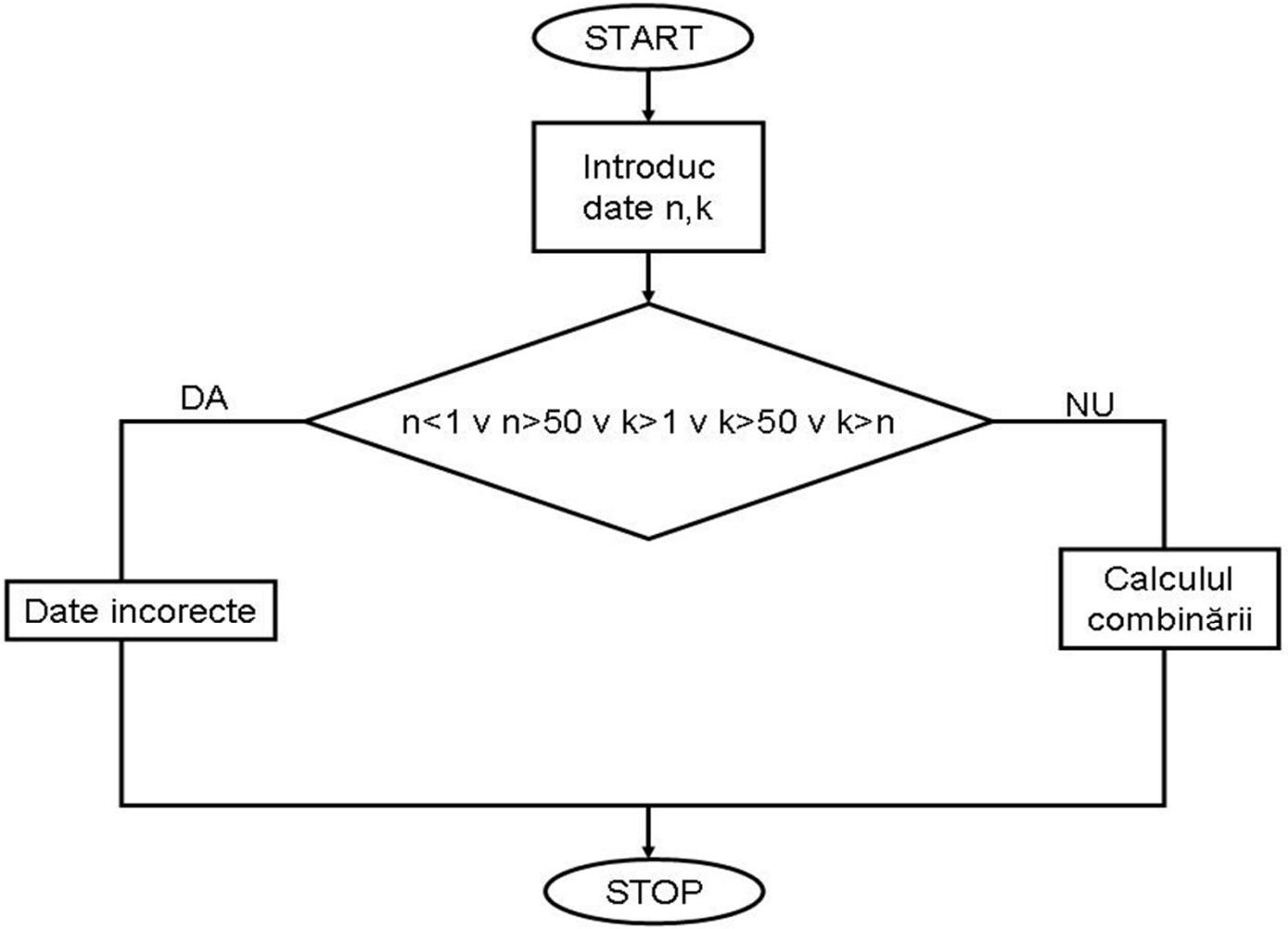
Exemplu:

Vom scrie un program care cere introducerea variabilelor n şi k de tip întreg de la tastatură şi verifică dacă acestea aparţin intervalului [1,50]. Programul calculează şi furnizează rezultatul:

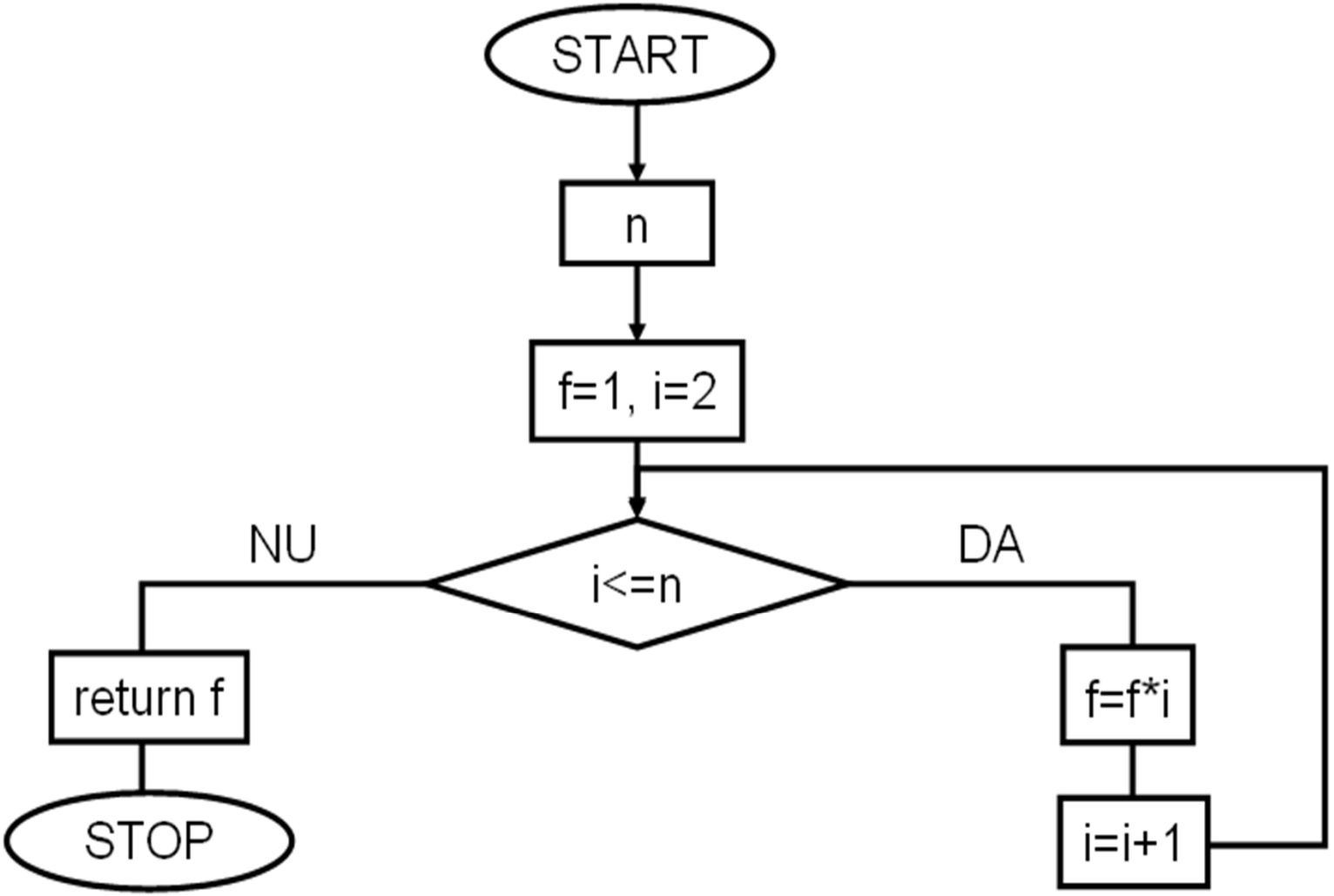
$$C_n^k = \frac{n!}{k!(n-k)!}$$

Se impune condiţia k<n.

Schema logică a unui program detaliază, în general, doar funcţia *main*.



Schema logică a funcţiei *fact* este:



```
#include <stdio.h>
double fact(int n);
int main()
{ long int k,n; /* pentru conversia rezultatului la long int */
  printf("\n introduceti pe n=");
  scanf("%d",&n);
  printf("\n introduceti pe k=");
  scanf("%d",&k);
  if (n<1||n>50||k<1||k>50||k>n)
          printf("\n Date incorecte");
  else printf("\n Rezultatul = %g", fact(n)/(fact(k)*fact(n-k)));
}
double fact(int n)
{ double f;
  int i;
  for (i=2,f=1.0;i<=n;i++)
          f*=i;
  return f;
}
```

4. Apelarea prin argumentele unei funcţii

În limbajul C apelarea unei funcţii se poate face (relativ la natura argumentelor sale) în două feluri:

- prin valoare;

- prin referinţă.

a. Apelarea prin valoare

În acest caz, care este şi cel mai uzual, se copiază valoarea unui argument într-un parametru formal al subrutinei. Modificările efectuate asupra parametrului utilizat nu au efect asupra argumentului efectiv al funcţiei.

b. Apelarea prin referinţă

Atunci când parametrul funcţiei este un pointer, în parametrul formal se copiază de fapt adresa unui argument. În interiorul subrutinei adresa este folosită pentru acces la argumentul folosit efectiv la apelare. Aceasta înseamnă că modificările efectuate asupra parametrului afectează argumentul.

Exemplu de apelare prin valoare:

```
#include <stdio.h>

void apelare_valoare(int x)
{       printf("In functia apelare_valoare x = %d inainte de a adauga 10.\n", x);
        x += 10;
        printf(" In functia apelare_valoare x = %d dupa ce adauga 10.\n", x);
}

int main()
{       int a=10;

        printf("a = %d inainte de apelul functiei apelare_valoare.\n", a);

         apelare_valoare(a);

        printf("a = %d dupa apelul functiei apelare_valoare.\n", a);
        return 0;
}
```

Exemplu de apelare prin referinţă:

```
#include <stdio.h>
...
void preluare(float *p1,float *p2);
...
void main()
{  float x,y;
   ...
   preluare(&x,&y);
   ...
}
void preluare(float *p1,float *p2)
{ printf("\n Introduceti primul numar: ");
  scanf("%f",p1);
  printf("\n Introduceti al doilea numar: ");
  scanf("%f",p2);
}
```

Exemplul 2 de apelare prin referinţă :

```
#include <stdio.h>

void apelare_referinta(int *y)
{       printf("In functia apelare_referinta y = %d inainte de a adauga 10.\n", *y);
        (*y) += 10;
        printf("In functia apelare_referinta y = %d dupa ce am adaugat 10.\n", *y);
}

int main()
{       int b=10;

        printf("b = %d Inaintea utilizarii functiei apelare_referinta.\n", b);
        apelare_referinta(&b);
        printf("b = %d Dupa utilizarea functiei apelare_referinta.\n", b);

        return 0;
}
```

5. Valori returnate de o funcţie

Pentru a returna o valoare, funcţia trebuie să utilizeze instrucţiunea ***return***. Această instrucţiune prezintă trei aspecte importante:

- Forţează ieşirea imediată din funcţia în care se află execuţia programului.
- Poate returna o valoare de tipul funcţiei.
- Determină execuţia programului să se reîntoarcă la instrucţiunea imediat următoare aceleia din care a fost apelată funcţia.

Observaţii:

1. Într-o funcţie pot exista una sau mai multe instrucţiuni *return*. Valoarea sau variabila care urmează după instrucţiunea *return* trebuie să fie de tipul funcţiei care o conţine.

2. De multe ori funcţiilor de tip *void* le lipseşte instrucţiunea *return*. Cu toate acestea o funcţie de tip *void* poate conţine una sau mai multe instrucţiuni simple *return*, dar fără a fi însoţite de o valoare de întoarcere.

Observaţii (continuare):

3. În general funcţia *main* nu returnează nimic şi se poate declara cu *void* (compilatoarele vechi).
4. O funcţie nu poate fi ţinta unei atribuiri.
   Astfel:

   f(x,y)=100;

   este o instrucţiune incorectă. De asemenea parametrii unei funcţii nu se iniţializează.
5. Tipul unei funcţii nu are legătură cu tipul atributelor sale.
6. Uneori, declarația de returnare nu este necesară în funcții specifice (în cazul în care argumentele sunt apelate prin referință)….. chiar dacă ne așteptăm ca ceva să fie returnat de la o astfel de funcție.

6. Funcţii recursive

În principiu o funcţie poate să se apeleze pe ea însăşi, ceea ce implică automat un fenomen de recursivitate.

Definiţie: o funcţie este recursivă dacă o instrucţiune din corpul ei apelează chiar acea funcţie.

Două variante (recursivă şi nerecursivă) ale funcţiei de calculare a factorialului unui număr întreg:

Varianta recursivă

```
int fact(int n)
{int f;
 if (n==1) return (1);
 f=fact(n-1)*n;
return(f);
}
```

Varianta nerecursivă

```
int fact(int n)
{ int i,f;
 for(f=1,i=2;i<=n;i++)
      f*=i;
return f;
}
```

Observaţii:

1. Rutinele recursive nu reduc semnificativ mărimea codului şi nici nu îmbunătăţesc utilizarea memoriei.

2. Multe versiuni recursive de funcţii sunt mai lente decât echivalentul lor iterativ.

3. Folosirea abuzivă a recursivităţii poate determina depăşirea limitei memoriei sistemului de calcul.

4. Recursivitatea are însă şi un avantaj: creează versiuni mai simple şi mai clare ale unor algoritmi (de exemplu sortarea).

5. Funcţiile recursive sunt întâlnite şi sub denumirea de *funcţii telescopice* (deoarece se întind şi se strâng).

7. Eficienţa funcţiilor

Funcţiile sunt esenţiale pentru a scrie eficient programele, mai puţin cele simple. Însă, în unele aplicaţii specializate este mai bine să se elimine o funcţie şi să se înlocuiască cu un *cod inline*.

Codul *inline* efectuează aceleaşi acţiuni ca şi ale funcţiei dar fără dezavantajul produs de o apelare de funcţie (printre care se numără şi micşorarea vitezei).

8. Declararea funcţiilor cu număr variabil de parametri

În C pot fi specificate funcţii cu un număr variabil de parametri.

Ca exemplu, funcţia:

func(int a, int b, ...);

va avea cel puţin doi parametrii de tip întreg şi un număr necunoscut (inclusiv 0) de parametrii după cei deja declaraţi.

Observaţii:

- Utilizarea sub forma ***func(...)*** este incorectă, deoarece funcţia trebuie să aibă cel puţin un parametru definit explicit.

- Funcţiile *printf* şi *scanf* sunt funcţii cu număr variabil de parametrii.

Clase de memorie

- La compilarea unui text sursă, compilatorul alocă memorie pentru variabilele programului într-una din următoarele zone:
  - pe stivă;
  - în regiştrii calculatorului;
  - în alte zone de memorie.

Variabile

- Locale
- Globale

Observaţie:

Variabilele globale au o definiţie şi eventual una sau mai multe declaraţii de variabilă externă.

Există două cazuri care pot fi discutate:

a. Programul este alcătuit dintr-un singur fişier

În acest caz variabila globală este scrisă în afara corpului oricărei funcţii a programului (de obicei la începutul fişierului sursă). Ea este valabilă din locul unde este scrisă şi până la sfârşitul fişierului sursă respectiv.

b. Programul se compune din mai multe fişiere

O variabilă globală poate fi utilizată într-un fişier sursă în care nu este definită, dacă este declarată ca variabilă externă (cu **extern**) în acel fişier.

Variabile locale

- **Variabilele locale au valabilitate locală, în unitatea (sau zona) în care au fost declarate.**

Variabile locale pot fi:

- automatice
- statice
- registru

Variabile ***automatice*** (declarate implicit sau cu *auto*)

Subcazuri:

- declarate înaintea primei instrucţiuni dintr-o funcţie; sunt puse pe stivă şi se distrug (curăţirea stivei la părăsirea funcţiei);
- declarate la începutul unei instrucţiuni compuse (bloc) (se alocă pe stivă şi se elimină când programul trece la instrucţiunea următoare celei compuse).

Variabile ***statice***

Se declară cu *static* (nu sunt pe stivă şi nu se distrug la părăsirea blocului, dar sunt apelate numai în interiorul său).

Subcazuri:

- declarate în corpul unei funcţii, sunt valabile numai în funcţia respectivă;
- declarate în afara corpurilor funcţiilor, sunt valabile numai în fişierul sursă în care sunt definite.

Variabile ***register***

Se declară folosind cuvântul cheie **register**

Tradiţional se aplică doar variabilelor *int* şi *char*. Totuşi standardul ANSI C permite folosirea pentru orice tip de variabilă. Iniţial *register* cerea compilatorului să păstreze valoarea variabilei într-un registru procesor. Ca urmare operaţiile se executau mult mai repede.

Subcazuri:

- ANSI C prevede că accesul la obiect se face cât mai rapid posibil;
- ANSI C++ prevede *register* ca o indicaţie dată compilatorului că obiectul astfel declarat va fi utilizat din plin.

Variabilele *register* pot fi de tip:

- *întreg* sau *caracter* sunt stocate în registre;

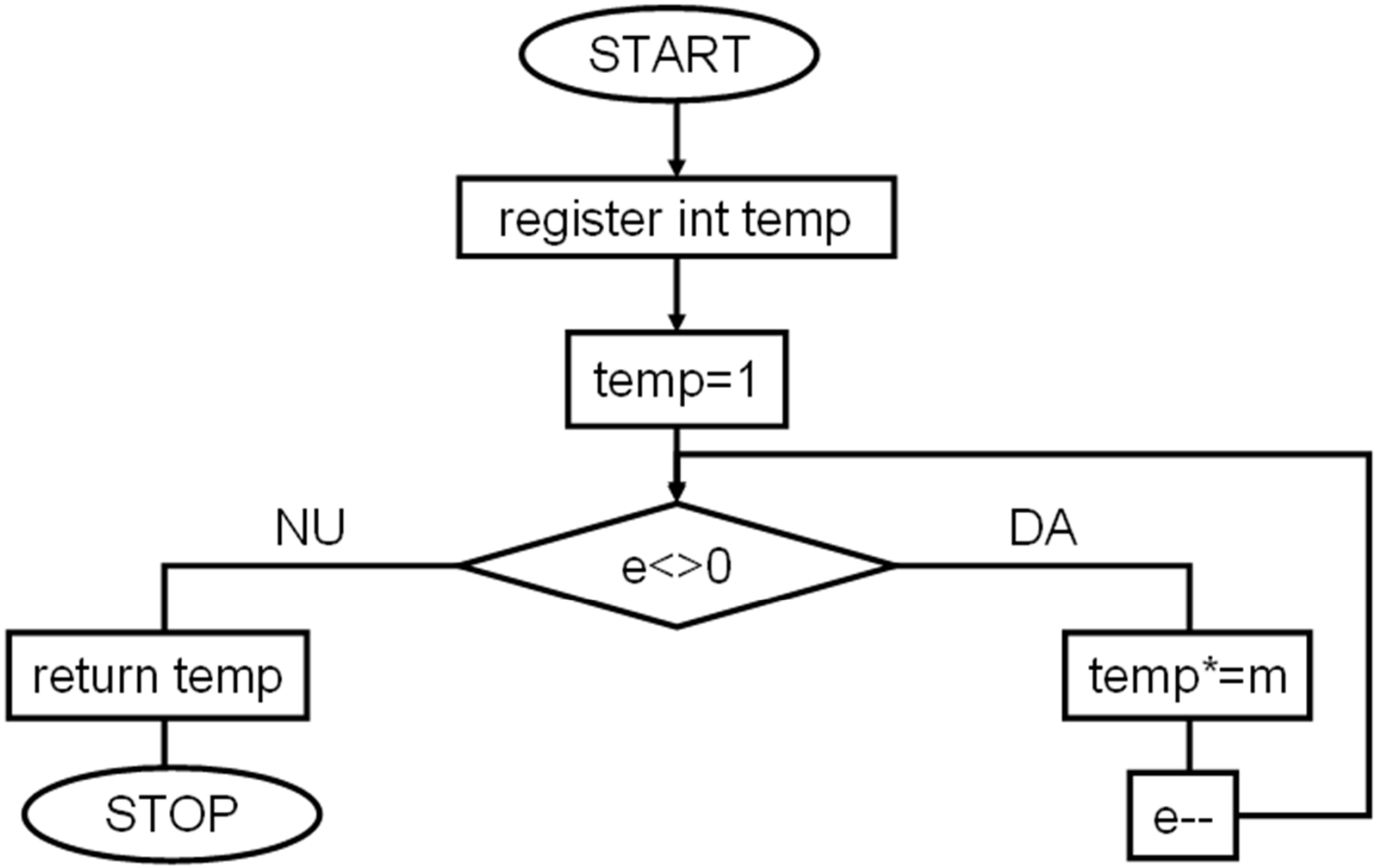
- mai mare (*matriceale*) nu sunt stocate în registre, dar ele beneficiază de un tratament preferenţial din partea compilatorului.

Observaţii:

1. Variabilele ***register*** pot fi tratate diferit astfel încât să corespundă modului de instalare a compilatorului de C/C++ şi mediului său de operare;
2. Tehnic vorbind, este permis unui compilator să ignore ***register*** şi să trateze normal variabilele specificate (caz rar în practică);
3. ***register*** se aplică doar variabilelor locale şi parametrilor formali ai unei funcţii.

Exemplu:

```
int power(register int m, register int e)
{ register int temp;
  temp=1;
 for ( ; e; e--)
        temp=temp*m;
return temp;
}
```


START
register int temp
temp=1
NU
e<>0
DA
return temp
temp*=m
STOP
e--

Alte observaţii
(continuare variabile registru):

1. Faptul că variabilele ***register*** sunt optimizate ca viteză le face ideale pentru controlul sau utilizarea în bucle;
2. Numărul permis de variabile ***register*** este limitat pentru compilatoarele de C şi C++. Dacă se declară prea multe variabile ***register****,* compilatorul le tratează automat ca variabile normale.
3. Deoarece variabilele ***register*** pot fi stocate în registrele procesorului, aceste variabile nu au adrese. Deci nu se poate găsi adresa variabilei ***register*** cu ajutorul operatorului ***&***.
4. Efectul sesizabil al lui ***register*** este doar pentru ***char*** şi ***int***. Nu se poate îmbunătăţi substanţial viteza pentru celelalte tipuri de variabile.

Observații privind variabilele:

- Variabilele globale (precum și cele statice) sunt inițializate cu 0 în mod automat.

- Variabilele locale iau valori arbitrare, ce depind de configurația biților din stiva unde sunt efectiv alocate.

Inițializarea variabilelor

- Prin inițializare se acordă o valoare inițială pentru o ***variablă***, de obicei imediat la declararea sa.

- Sintaxa:

  clasa tip nume = expresie;

Observații:

- Evitați utilizarea variabilelor locale (auto) care nu sunt inițializate. Fără inițializare, aceste variabile au valori arbitrare, fenomen ce se datorează faptului că memoria stivă, ce le conține, nu șterge efectiv valorile vechilor variabile dezalocate (adica nu pune valoarea 0 în zonele dezalocate).

- **Atenție: parametrii unei funcții nu se inițializează!**

Cap. **Matrici (Tablouri)**

- Tehnica tablourilor reprezintă o metodă comodă de stocare a matricilor în memoria unui sistem de calcul.
- **În programarea C,** o matrice (sau tablou) este o colecţie de variabile de acelaşi tip, oricare dintre elemente putând fi apelate cu acelaşi nume**.**
- C-ul utilizează principiul "Row Major", adică elementele, de pe fiecare rând al unei matrici, sunt stocate succesiv în memorie (la adrese succesive raportate la dimensiunea tipului).

Declararea unei matrici

- Sintaxa:

  tip Nume_matr [lim_1] [lim_2] … [lim_n];

unde:

- *tip* → tipul elementelor matricei
- *lim_i* → limita superioară a indicelui al i-lea (ce poate avea valorile:0, 1, 2,… , lim_i – 1)

Observaţii:

- Valorile *lim_i* (unde *i* = 0, 1, …, *n*) sunt expresii constante întregi pozitive.
- La elementele tabloului se poate face referire folosind variabile cu indici.
- În *C* numele unui tablou este un simbol care are ca valoare adresa primului său element.
- La întâlnirea unei declaraţii de tablou, compilatorul alocă o zonă de memorie necesară pentru a păstra valorile elementelor sale.
- Tablourile şi pointerii prezintă o strânsă legătură în C (a se vedea capitolul viitor dedicat pointerilor).

Exemplu:

- **int a[3][4];**

  se declară o matrice de numere întregi de 3 linii și 4 coloane. Indexul de linie pornește de la 0 și ajunge până la 2.

	Coloana 0	Coloana 1	Coloana 2	Coloana 3
rândul 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
rândul 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
rândul 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Legătura dintre pointeri şi matrici

- Numele unui tablou este un pointer deoarece el are ca valoare adresa primului său element.

- Există o diferenţă între numele unui tablou şi o variabilă de tip pointer. Astfel, unei variabile de tip pointer i se pot atribui valori la execuţie, în timp ce acest lucru nu este posibil să se realizeze pentru numele unui tablou (acesta are tot timpul ca valoare adresa primului său element).

- Se obişnuieşte să se spună că numele unui tablou este un ***pointer constant***.

Matrici unidimensionale (vectori)

- Vectorii reprezintă matricele (tablourile) cu o singură dimensiune.

- Format:

tip nume_vect [lim];

Observaţii:

1. În C toate matricele au ca indice pentru primul element valoarea zero (0). La declararea:

char p[10];

se creează o matrice de 10 elemente de tip caracter de la *p[0]* la *p[9]*.

2. Cantitatea de memorie necesară pentru înregistrarea unei matrice este direct proporţională cu tipul şi mărimea sa. Mărimea totală în octeţi se calculează:

*total octeţi=sizeof(tip de bază)*mărimea matricei*

3. *C* nu controlează limitele unei matrice. Se pot depăşi ambele margini ale unei matrice şi scrie în locul altor variabile sau peste codul programului. Rămâne în sarcina programatorului să controleze limitele acolo unde este necesar.