VII. Apelul recursiv al funcţiilor

1. **Conceptul de recursivitate**
2. **Recursivitatea directă**
3. **Înregistrarea de activare**
4. **Relația dintre recursivitate şi iterație**
5. **Exemple de programe recursive**

Conceptul de recursivitate

Definiţii

- Un obiect sau un fenomen este definit în mod **recursiv** dacă în definiția sa se face referire la el însuşi.
- Conceptul de **recursivitate** oferă posibilitatea definirii unei infinități de obiecte printr-un număr finit de relații.
- **Recursivitatea** a fost introdusă în programare în 1960, în limbajul **Algol**.
- *O funcție este recursivă atunci când executarea ei implică cel puțin încă un apel către ea însăşi.*
- *Recursivitate directă* – apelul recursiv se face chiar din funcția invocată.
- *Recursivitate indirectă (mutuală)* – apelul recursiv se realizează prin intermediul mai multor funcții care se apelează circular.

Conceptul de recursivitate

Exemple de funcţii recursive

$$fact(n) = \begin{cases} 1 & \text{dacă } n = 0 \\ n * fact(n-1) & \text{dacă } n > 0 \end{cases}$$

fib: $N \rightarrow N$ (Fibonacci)

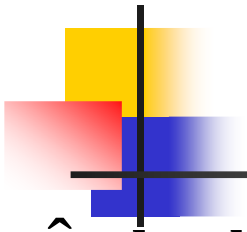
$$fib(n) = \begin{cases} 1 & \text{dacă } n = 0 \text{ sau } n = 1 \\ fib(n-2)+fib(n-1) & \text{dacă } n > 1 \end{cases}$$

ac: $N \times N \rightarrow N$ (Ackermann)

$$ac(m, n) = \begin{cases} n+1 & \text{dacă } m = 0, \\ ac(m-1, 1) & \text{dacă } n = 0, \\ ac(m-1, ac(m, n-1)) & \text{dacă } m \neq 0 \text{ şi } n \neq 0. \end{cases}$$

Recursivitatea directă

Principii



- În limbajul C funcțiile se pot apela pe ele însele, adică sunt direct recursive. Pentru o funcționare corectă (din punct de vedere logic), apelul recursiv trebuie să fie condiționat de o decizie care, la un moment dat în cursul execuției, să împiedice continuarea apelurilor recursive şi să permită astfel revenirea din şirul de apeluri.

- Lipsa acestei condiții sau programarea ei greşită va conduce la executarea unui şir de apeluri a cărui terminare nu mai este controlată prin program şi care, la epuizarea resurselor sistemului, va provoca o eroare de execuție : *Depăşirea stivei de date*.

Recursivitatea directă

Exemplu schematic

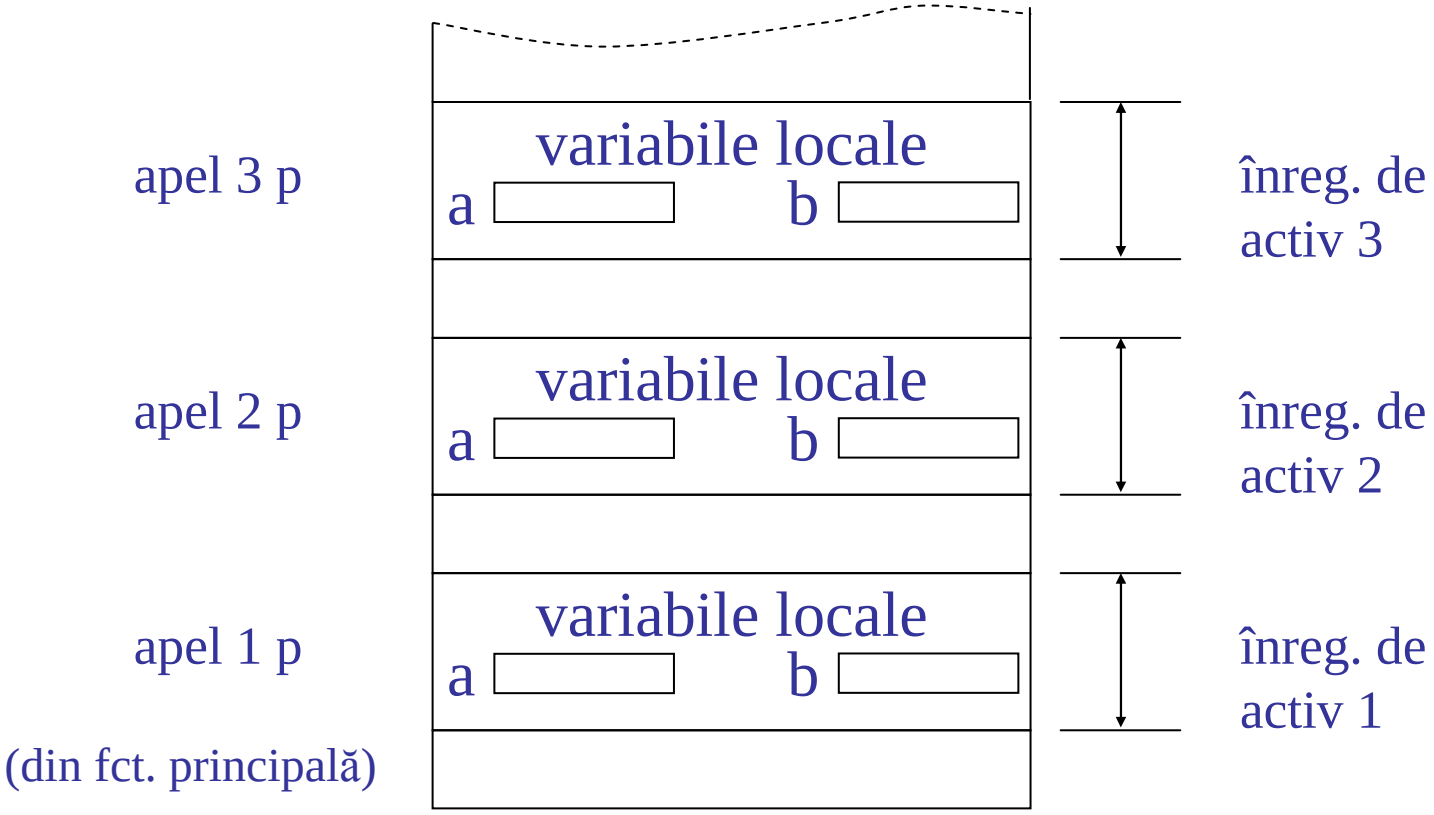
```
void p (listă de parametri){
        ... a, b;
        ...
                                  if (cond)  p(...) ...    sau:
        p(...);        →{  while (cond) { ...p(...) } sau:
                                  do ... p(...)... while (cond)
}
```

- Condiția care trebuie testată este specifică problemei de rezolvat. Programatorul trebuie să o identifice în fiecare situație concretă şi, pe baza ei, să redacteze corect apelul recursiv.
- Revenirea din apeluri se face în ordine inversă.

Înregistrarea de activare

- Unei funcții i se "acordă", la apel, o zonă de memorie numită **înregistrare de activare**. Aceasta conține, printre altele, locațiile rezervate pentru valorile *parametrilor* (argumentelor), pentru *variabilele automatice* şi pentru *rezultatul returnat* de funcție.
- La fiecare apel se crează o nouă înregistrare de activare, incluzând un nou set de parametri şi de variabile automatice. Zona de memorie în care se face această rezervare se numeşte **stiva de date**.
- În cazul apelurilor recursive vor exista în memorie, simultan, mai multe înregistrări de activare pentru aceeaşi funcție recursivă, fiecare conținând câte un set de parametri şi de variabile automatice (locale). Deşi locațiile poartă acelaşi nume în fiecare set, ele reprezintă entități diferite şi au valori distincte.
- Pe parcursul unui apel nu sunt accesibile decât locațiile din înregistrarea de activare creată la apelul curent.

Stiva de date a programului pe parcursul unui apel recursiv



Exemplu de program recursiv

Enunţul problemei

- Se cere să se scrie un program care să citească un cuvânt de pe mediul de intrare şi să-l afişeze atât normal cât şi în ordinea inversă a literelor. Cuvântul va fi urmat de spațiu. Citirea se va efectua caracter cu caracter, într-o singură variabilă de tip *char*.

Model de execuție :

Linia de intrare : ABC ↵

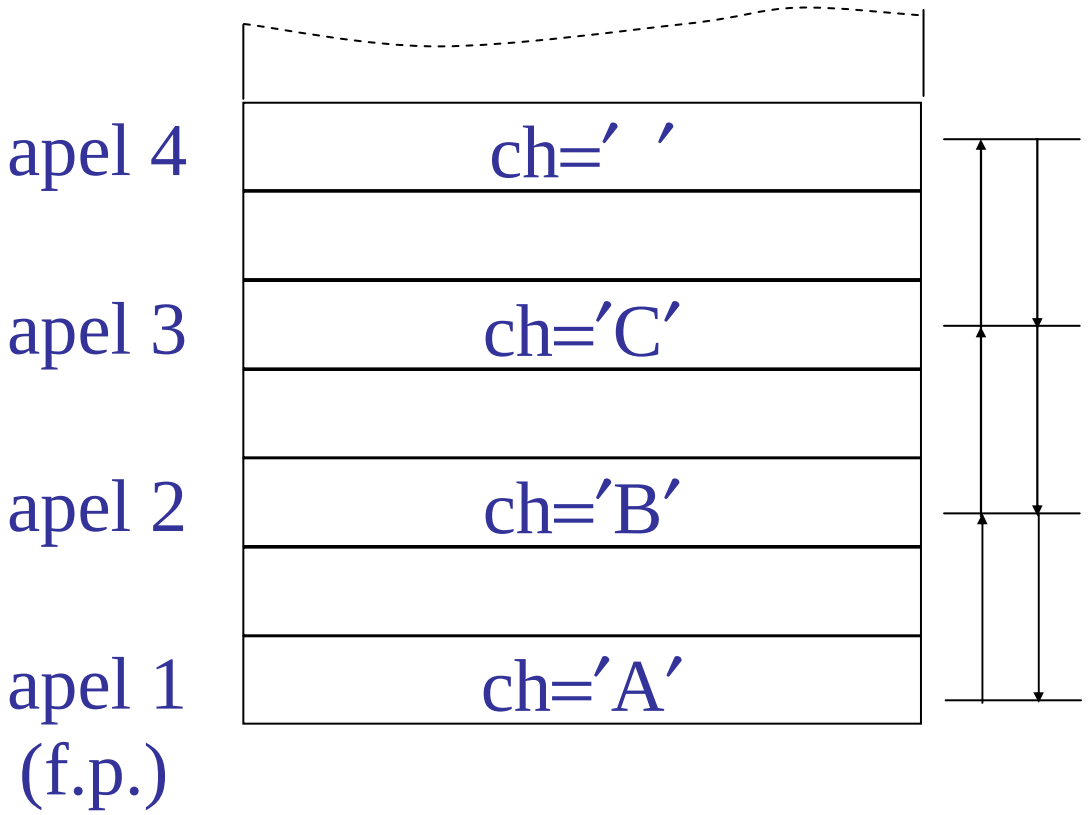
Linia de ieşire : ABC CBA

Exemplu de program recursiv

Soluţia problemei

```
#include <stdio.h>
void Invers( void ){
   char ch;
   ch = getchar( );  printf("%c",ch);
   if (ch != ′ ′ )
       Invers();
   printf("%c",ch);  }
void main(void ){
   Invers( );
}
```

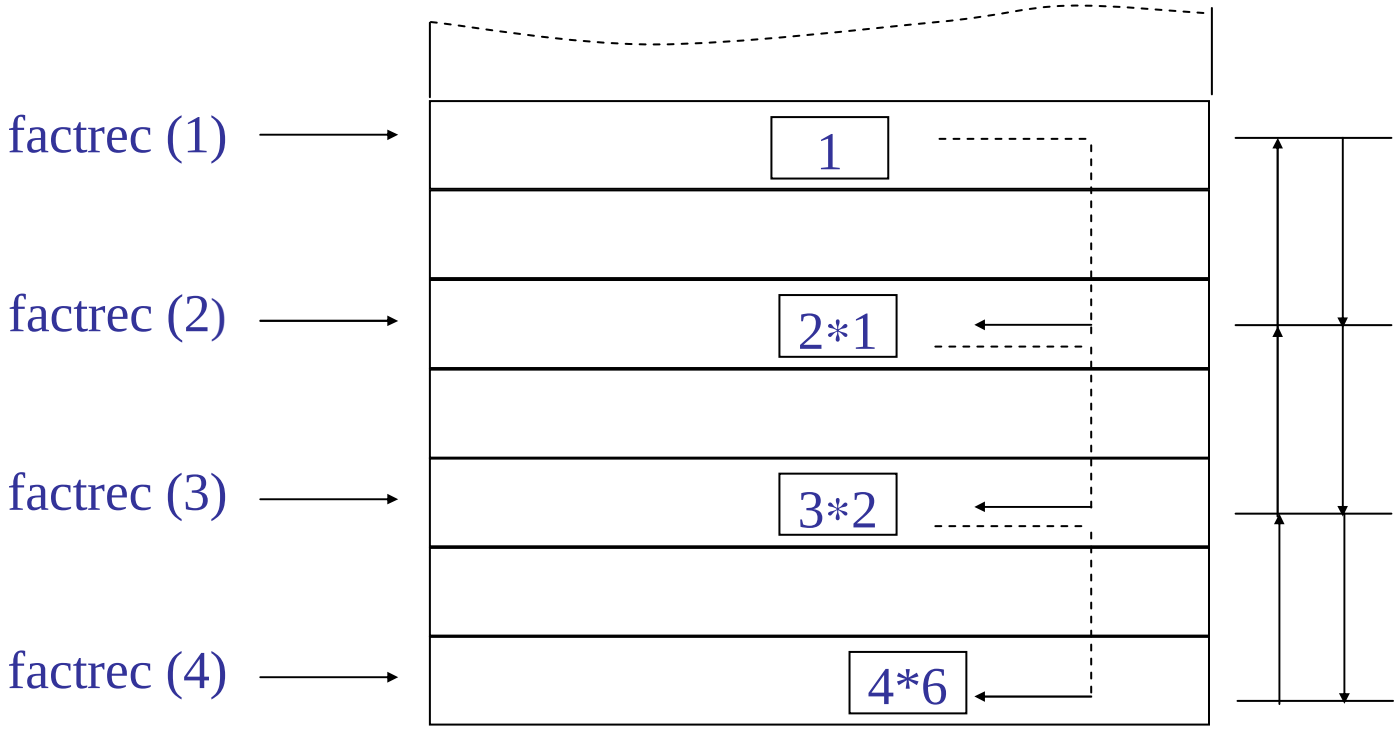
Dinamica stivei la execuţie, pentru linia de intrare dată ca exemplu



Calculul recursiv al factorialului

Dinamica stivei în cazul apelului *factrec(4)*

```
double factrec (unsigned n){
    if (n <= 1) return 1;
    else return n*factrec(n-1);
}
```

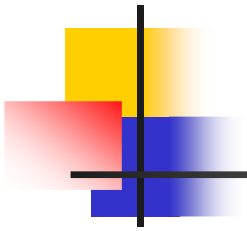


Calculul recursiv al funcţiei putere

$$a^n = \begin{cases} 1 & n=0 \\ a \times a^{n-1} & n>0 \end{cases}$$

```
#include <stdio.h>
double putere(double a, unsigned n) {
    return n = = 0 ? 1 : a*putere(a, n-1);
}
int main(void) {
   printf("1.5 la puterea 5 = %f \n", putere(1.5, 5));
   return 0;
}
```

Relaţia dintre recursivitate şi iteraţie - Comparaţie



Iterația

- execuția repetată a unei secvențe de instrucțiuni
- o nouă iterație se execută doar în urma evaluării unei condiții (**la început sau sfârşit**)
- fiecare iterație se execută până la capăt şi apoi se trece, eventual, la o nouă iterație
- se recomandă atunci când algoritmul de calcul este exprimat printr-o formulă iterativă

Recursivitatea

- execuția repetată a unei funcții
- un nou apel recursiv se execută tot în urma evaluării unei condiții (**pe parcurs**)
- funcția recursivă se apelează din nou, înainte de terminarea apelului precedent
- se recomandă doar atunci când problema este prin definiție recursivă (recursivitatea consumă resurse în exces)

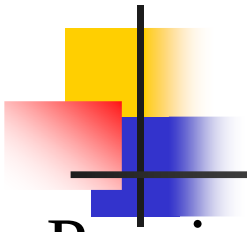
Relaţia dintre recursivitate şi iteraţie - Comparaţie

- Ce este mai indicat de utilizat, apelul recursiv sau calculul iterativ? Deşi, aparent, cele două variante sunt echivalente, se impune totuşi următoarea remarcă generală: dacă algoritmului de calcul îi corespunde o formulă iterativă este de preferat să se folosească aceasta în locul apelului recursiv, întrucât viteza de prelucrarea este mai mare şi necesarul de memorie mai mic. De exemplu, calculul recursiv al numerelor lui Fibonacci este complet ineficient întrucât numărul de apeluri creşte exponenţial odată cu n (pentru $n = 5$ sunt necesare 15 apeluri).
- În principiu, orice algoritm recursiv poate fi transformat într-unul iterativ. Această transformare se poate însă realiza mai uşor sau mai greu. Sunt situaţii în care, în varianta iterativă, programatorul trebuie să gestioneze, în mod explicit, stiva apelurilor, salvându-şi singur valorile variabilelor de la un ciclu la altul, ceea ce este dificil şi îngreunează mult înţelegerea algoritmului (ex. funcţia *Invers*).

Relaţia dintre recursivitate şi iteraţie - Comparaţie

```
double factrec (unsigned n){
   if (n <= 1)
       return 1;
   else
       return n*factrec(n-1);
}
double factiter(unsigned n) {
   double f=1 ; int i ;
   for (i=2; i<=n; i++)  f*=i;
   return f;  }
```

Relaţia dintre recursivitate şi iteraţie - Comparaţie

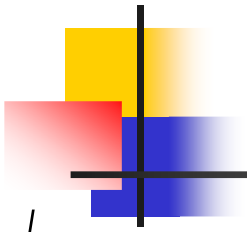


- Pornind de la performanțele calculatoarelor actuale şi de la faptul că principala resursă a devenit timpul programatorului, recursivitatea are astăzi domeniile ei bine definite, în care se aplică cu succes. În general se apreciază că algoritmii a căror natură este recursivă trebuie formulați ca funcții recursive: prelucrarea structurilor de date definite recursiv (liste, arbori), descrierea proceselor şi fenomenelor în mod intrinsec recursive.

- S-au elaborat de asemenea metode recursive cu mare grad de generalitate în rezolvarea problemelor de programare (exemplu: *backtracking)* pe care programatorii experimentați le aplică cu multă uşurință, ca nişte scheme, aproape în mod automat.

Exemplu

Rădăcina patrată



$\sqrt{x}$: $a_0=1$, $a_{n+1} = (a_n+x / a_n) / 2$

Când s-a atins precizia dorită ($|a_{n+1}-a_n|<\varepsilon$), $\sqrt{x}$ este a_n.

```
#include <stdio.h>
#include <math.h>
double rad(double x, double an) {
   return x < 0 ? 0 :
       fabs((x/ an-an)/2)<0.0001 ? an : rad(x, (an+x/ an)/2);
 }
int main(void) {
   printf("radical din 7 este %f \n", rad(7.0, 1.0));
   return 0;
}
```

Algoritmul lui Euclid de aflare a celui mai mare divizor comun a două nr. întregi

- Formularea recursivă, în cuvinte, a algoritmului:

Dacă unul dintre numere este zero, c.m.m.d.c. al lor este celălalt număr.

Dacă nici unul dintre numere nu este zero, atunci c.m.m.d.c. nu se modifică dacă se înlocuieşte unul dintre numere cu restul împărțirii sale cu celălalt.

- Algoritmul poate fi implementat sub forma următoarei funcții recursive:

```
unsigned cmmdc (unsigned m, unsigned n){
    if (n==0) return m;
    else return cmmdc(n, m % n);
}
```

Algoritmul lui Euclid

Exemplu numeric: *cmmdc(18, 27)*

m	***n***	***cmmdc***
18	27	*cmmdc (27, 18 % 27)*
27	18	*cmmdc (18, 27 % 18)*
18	9	*cmmdc (9, 18 % 9)*
9	0	*cmmdc=9*

Algoritmul lui Euclid

Varianta 2

$$cmmdc(m, n)=\begin{cases} m & m=n \\ cmmdc(m-n, n) & m>n \\ cmmdc(m, n-m) & m<n \end{cases}$$

```
#include <stdio.h>
unsigned cmmdc(unsigned m, unsigned n) {
  return m==n ? m:m>n ? cmmdc(m-n,n) : cmmdc(m, n-m);
}

int main(void) {
   printf("cmmdc(18, 27) este %u\n", cmmdc(18, 27));
   return 0;
}
```

Algoritmul lui Euclid

Varianta 3

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
unsigned cmmdc1(unsigned m, unsigned n){
        if (n==0)
            return m;
        else
            return cmmdc1(n, m%n);  }
unsigned cmmdc2(unsigned m, unsigned n){
        return m == n? m : m>n? cmmdc2(m-n,n) :
                cmmdc2(m, n-m);
}
```

Algoritmul lui Euclid

Varianta 3

```
int main(void){
        char *varianta=(char*)malloc(100);
        unsigned (*p)(unsigned,unsigned);
        scanf("%s",varianta);
        if(!strcmp(varianta,"cmmdc1"))
                p=&cmmdc1;
        else
                p=&cmmdc2;
        printf("%s(18, 27) = %u\n",varianta, (*p)(18, 27));
        free(varianta);
        return 0;
}
```

Generarea permutărilor

Enunţul problemei

- Se consideră o secvență de *n* numere naturale distincte. Se cere să se genereze toate secvențele reprezentând permutările celor *n* numere.

- Se porneşte de la primele *n* numere naturale ordonate crescător într-un tablou a_i, i=1,n. Prin apelul recursiv al funcției *Permută* se reduc în cascadă *permutări (k)* la *permutări (k-1)*, efectuându-se operațiile descrise în algoritmul următor (în pseudocod):

Generarea permutărilor

Algoritmul de rezolvare al problemei

```
** procedura Permută (K ; ) este
   intreg K, I
   dacă K = 1 atunci * Tipăreşte soluția
   altfel
      Permută(K-1);
      pentru I := 1 la  K-1  execută
         * schimbă între ele A[I] cu A[K]
         * Permută (K-1);
         * schimbă între ele A[I] cu A[K]
      □
   □
sfârşit
```

Generarea permutărilor

Programul

```
#include <stdio.h>
#define N 4
int a[N+1];
void Tipareste(){
  int i;
  for (i=1; i<=N; i++)
    printf("%d ", a[i]);
  printf("\n");
}
```

Generarea permutărilor

Programul

```
void Permutare(int k) {
  int i,x;
  if (k == 1)
        Tipareste();
  else {
    Permutare(k-1);
    for (i=1; i<=k-1; i++) {
       x = a[i]; a[i] = a[k]; a[k] = x;
       Permutare(k-1);
       x = a[i]; a[i] = a[k]; a[k] = x;
    }
  } }
```

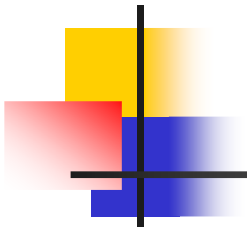
Generarea permutărilor

Programul

```
int main(int argc, char *argv[ ]) {
  int i;

  for (i=1; i<=N; i++)
        a[i] = i;
  Permutare(N);
  return 0;
}
```

Generarea permutărilor
Concluzii



- Pentru a parcurge în mod corect şi complet toate permutările prin înlocuirea fiecărui element a_k cu toate elementele a_i , la revenirea din cel de-al doilea apel recursiv trebuie refăcută starea inițială a tabloului *a*. Această situație sau situații similare apar frecvent în programele recursive, atunci când procedura recursivă efectuează modificări asupra unor variabile globale (externe funcțiilor).
- Se observă că programul conține chiar operațiile descrise anterior în pseudocod, la care se adaugă condiția de oprire a apelării recursive ($k = 1$, caz în care s-a obținut o secvență finală şi se tipăreşte). În concluzie, descrierea trecerii de la un pas la pasul următor, calitativ identic, împreună cu condiția de oprire, sunt suficiente pentru implementarea în program a unui algoritm recursiv(similitudine cu metoda inducției matematice).