

Recursivitate

1. Mecanism general

Recursivitatea este o metodă de elaborare a algoritmilor. Noțiunea se regăsește și în matematică, unde este o modalitate de definiție a conceptelor. Numită și definiția prin inducție, este principiul care corespunde demonstrației prin inducție matematică. Scrierea recursivă a programelor este un caz particular de dezvoltare descendentă, și anume problema de rezolvat este redusă la probleme din ce în ce mai simple, însă identice ca structură cu problema inițială.

Un obiect (algorithm, concept etc.) este **recursiv** dacă el este descris prin intermediul unor obiecte de același tip cu obiectul însuși. Paradoxal și în aparență, un obiect recursiv conține, în definiția sa, obiectul - dar nu identic, ci unul asemănător - care tocmai se definește. O definiție recursivă implică, deci, referiri la însuși obiectul definit.

În programare, recursivitatea este implementată cu ajutorul subprogramelor. Un subprogram este recursiv dacă el se autoapelează (când subprogramul este lansat în execuție, se fac apeluri la el însuși). Dacă autoapelul apare în instrucțiunea compusă a subprogramului (notat **P**), recursivitatea se numește **directă**.

```
tip P(...) { o funcție}
{.....
.....
P(...); {apel P}
.....
}
```

Dacă un subprogram nu se autoapelează, spunem că el este **iterativ**.

Exemplul1: Factorialul unui număr.

Fie **n** număr natural. Prin definiție: $n! = 1 * 2 * 3 * \dots * n$, pentru $n > 0$ și $0! = 1$.

Definim recursiv **n!** astfel :

$$n! = \begin{cases} 1 & , n=0 \\ n*(n-1)! & , n \geq 1 \end{cases}$$

Factorialul lui **n** este definit tot prin intermediul unui factorial, și anume al factorialului lui **n-1**. Deci factorialul care se definește este redus la factorialul unui număr mai mic cu o unitate. Evident, după un număr de pași, vom ajunge la calculul lui **0!**, care este definit direct ca egal cu 1.

$$\text{Deci } 3! = 3 * 2! = 3 * (2 * 1!) = 3 * (2 * (1 * 0!)) = 3 * (2 * (1 * 1)) = 6$$

Funcția C transcrie fidel definiția matematică și parcurge, în execuție, etapele calculului de mai sus.

```
#include<iostream>

int n;
int fn;
int fact(int k)
{
    if (k==1)
        return 1;
    else
        return fact(k-1)*k;
}

int main()
{
    cout<<"n=";<<n;
    fn=fact(n);
    cout<<"factorial="<<fn;
}
```

2. Conceperea algoritmilor recursivi

Definițiile recursive date în exemplele anterioare au toate aceeași structură, în sensul că fiecare din ele conține două părți. Prima ramură a definiției recursive cuprinde una sau mai multe definiții directe. Altfel spus, pentru anumite date de intrare, valorile funcției sunt date direct și nu prin intermediul unor alte valori ale acesteia. Această parte din definițiile recursive este esențială și face ca procesul de calcul să se oprească după un număr finit de pași. Ea se numește **condiție de oprire**. În implementarea C a algoritmilor, această parte de definiție corespunde cel puțin unei execuții fără autoapel.

A doua ramură a definiției recursive cuprinde descrierea operațiilor care se fac la un pas oarecare al algoritmului. La funcțiile matematice scrise anterior, constă într-un proces inductiv de calcul al unor noi valori ale funcției pe baza unor valori deja cunoscute ale ei.

Reguli pentru scrierea corectă a subprogramelor recursive:

- a) Orice subprogram recursiv trebuie să se execute, cel puțin o dată, fără a se autoapela. Condiția de oprire se va scrie pentru valori extreme ale mulțimii valorilor de testat.

- b) Autoapelurile trebuie să conducă spre situația (situațiile) în care subprogramul se execută direct (fără autoapel).

Exemplu: Suma cifrelor unui număr natural. Fie n natural și $S(n)$ suma cifrelor numărului n . Definim $S(n)$ recursiv astfel:

$$S(n) = \begin{cases} 0 & , n=0 \\ n \% 10 + S(n/10) & , n \geq 1 \end{cases}$$

Se observă că $S(n)$ este definită tot prin intermediul sumei cifrelor unui număr, și anume $n/10$, număr obținut din n prin eliminarea ultimei sale cifre, $n \% 10$. Deci, calculul sumei cifrelor unui număr este redus la calculul sumei cifrelor unui număr care are cu o cifră mai puțin. După un număr de pași, evident se va ajunge la calculul sumei cifrelor unui număr cu 0 cifre, adică numărul 0. $S(0)=0$, valoare definită direct, și calculul se oprește aici.

Astfel, pentru $n=473$, conform definiției avem :

$$S(473)=3+S(47)=3+(7+S(4))=3+(7+(4+S(0)))=3+(7+(4+0))=14$$

```
#include<iostream>
longint n;
int s(longint n)
{
    if(!n)
        return 0;
    else
        return ( n%10+s(n/10) );
}
int main()
{
    cout<<"n=";<<n;
    cout<<"suma cifrelor="<<s(n);
}
```

3. Avantajele și dezavantajele recursivității

S-a demonstrat că pentru orice algoritm recursiv există unul iterativ echivalent și, reciproc, pentru unul iterativ există unul recursiv echivalent. Pentru care optăm ?

Algoritmii recursivi, deoarece realizează la fiecare autoapel salvări pe stivă necesită mai mult spațiu de memorie și, implicit, timp de execuție mai îndelungat. Dacă numărul de autoapeluri (adâncimea stivei sau a recursivității) este mare, spațiul de memorie alocat stivei poate fi insuficient; compilatorul transmite, în aceste situații mesajul “**stack overflow**” (depășire stivă) și programul nu poate fi executat.