Programarea calculatoarelor
Limbajul C

CURS 12

Funcții recursive

Ce este recursivitatea ?

- Un obiect sau un fenomen se defineşte în mod **recursiv** dacă în definiția sa există o **referire la el însuşi**.
- Recursivitatea este folosită cu multă eficiență în matematică. Exemple de definiții matematice recursive:

1. definiția numerelor naturale:
   0 € N
   dacă i € N, atunci succesorul lui i € N
2. definiția funcției factorial
   fact : N -> N
   fact(n) = 1, dacă n=0
   = n * fact (n-1), dacă n>0
3. definiția funcției Fibonacci
   fib : N -> N
   fib(n) = 1, dacă n=0 sau n=1
   = fib(n-2) + fib(n-1), dacă n>1

Recursivitate

- Utilitatea practică a recursivității: posibilitatea de a defini un **set infinit de obiecte printr-o singură relație sau printr-un set finit de relații**.

- Recursivitatea s-a impus în programare odată cu apariția unor limbaje de nivel înalt, ce permit scrierea de module ce se autoapelează
  - PASCAL, LISP, ADA, ALGOL, C - limbaje recursive
  - FORTRAN,BASIC,COBOL - limbaje nerecursive

- Un program recursiv poate fi exprimat: P=M(Si,P) , unde M este mulțimea ce conține instrucțiunile Si şi pe P însuşi.

Recursivitate versus iterativitate

- **Iterația**
  - execuția repetată a unei porțiuni de program, până la îndeplinirea unei condiții (while, do-while , for din C).
- **Recursivitatea**
  - execuția repetată a unui modul
  - în cursul execuției lui se verifică o condiție
  - nesatisfacerea condiției implică reluarea execuției modulului de la început, fără ca execuția curentă să se fi terminat
  - în momentul satisfacerii condiției se revine în ordine inversă în lanțul de apeluri, reluându-se şi încheindu-se apelurile suspendate.

Funcții recursive în C

- Exprimarea recursivității în C : O funcție recursivă este o funcție care se apelează pe ea însăşi.
- Recursivitatea poate fi:
  - directă - o funcție P conține o referință la ea însăşi
  - indirectă - o funcție P conține o referință la o funcție Q ce include o referință la P.
- Se pot deosebi două feluri de funcții recursive directe:
  - Funcții cu un singur apel recursiv, ca ultimă instrucțiune
    - se pot rescrie uşor sub formă nerecursivă (iterativă).
  - Funcții cu unul sau mai multe apeluri recursive
    - forma lor iterativă trebuie să folosească o stivă pentru memorarea unor rezultate intermediare.

Exemplu

```
#include<stdio.h>

//Funcția recursivă de calcul a factorialului:
int fact (int n) {
    if (n==1 || n==0 ) return 1;
    return n*fact(n-1); //apel recursiv
}

//Varianta nerecursivă, iterativă
int fact_it (int n) {
    int i, f;
    for( i=f=1; i<=n; i++ )
        f *= i;
    return f;
}
```

Exemplu - continuare

```
int main(){
    int n;
    scanf("%d",&n);
    for( int i=1; i<=n; i++ )
    printf("%d %d %d\n",i, fact(i),fact_it(i));
    fflush(stdin);
    getchar();
    return 0;
}
```

- Observație: tipul funcției factorial trebuie să devina long pentru n>=16!

Exemplu de funcție recursivă de tip *void*

```
#include<stdio.h>
void binar (int n) { // afiseaza n în baza 2
    if (n>0) {
       binar (n/2);     // scrie echiv. binar al lui n/2
       printf ("%d",n%2);        // şi restul impartirii lui n la 2
    }
}
int main(){
    int n;
    scanf ("%d",&n);
    printf ("%d = ", n);
    if (n) binar(n);
    else printf("0");
    fflush (stdin);
    getchar ();
    return 0;
}
```

Verificarea şi simularea programelor recursive

- Printr-o demonstrație formală sau testând toate cazurile posibile.
- Se verifică întâi dacă toate cazurile particulare (ce se execută când se îndeplineşte condiția de terminare a apelului recursiv) funcționează corect.
- Se face apoi o verificare formală a funcției recursive, pentru restul cazurilor, presupunând că toate componentele din codul funcției funcționează corect.
- Verificare inductivă
  - avantaj al programelor recursive, ce permite demonstrarea corectitudinii lor simplu şi clar.
- Exemplu: factorial
  - Demonstrarea corectitudinii cuprinde doi paşi:
    - pentru n=1 valoarea 1 ce se atribuie factorialului este corectă
    - pentru n>1, presupunând corectă valoarea calculată pentru predecesorul lui n de către fact(n-1), prin înmulțirea acesteia cu n se obține valoarea corectă a factorialului lui n.

Observații

- Orice funcție recursivă trebuie să conțină (cel puțin) o instrucțiune *if* (de obicei chiar la început), prin care se verifică dacă (mai) este necesar un apel recursiv sau se iese din funcție!
- Absența instrucțiunii *if* conduce la o recursivitate infinită (la un ciclu fără condiție de terminare)!

- Pentru funcțiile de tip diferit de *void* apelul recursiv se face printr-o instrucțiune *return*, prin care fiecare apel preia rezultatul apelului anterior!

- Funcțiile recursive nu conțin în general cicluri explicite (cu unele excepții), iar repetarea operatiilor este obținută prin apelul recursiv!

Observație

- Apelul recursiv al unei funcții trebuie să fie condiționat de o decizie care să împiedice apelul în cascadă (la infinit); aceasta ar duce la o eroare de program - depăşirea stivei.

- funcție recursiva:

```
void p( ){
        p();  //apel infinit
}
```

- apelul este de obicei condiționat astfel:

```
  void p( ){
        if (condiție) p();  // apel finit condiționat
}
```

Realizarea recursivității în C

- La fiecare apel al funcției sunt puse într-o stivă (gestionată de compilator)
  - adresa de revenire
  - variabilele locale
  - argumentele formale
  - acestea fiind referite şi asupra lor făcându-se modificările în timpul execuției curente a funcției
- La ieşirea din funcție (prin *return*) se scot din stivă toate datele puse la intrarea în funcție (se "descarcă" stiva)
  - modificările operate asupra parametrilor-valoare nu afectează parametrii efectivi de apel corespunzători

Observații

- Pe parcursul unui apel, sunt accesibile doar variabilele locale şi parametrii pentru apelul respectiv, nu şi cele pentru apelurile anterioare, chiar dacă acestea poartă acelaşi nume.

- Atenție! Pentru fiecare apel recursiv al unei funcții se crează copii locale ale parametrilor valoare şi ale variabilelor locale, ceea ce poate duce la risipă de memorie!

Ce este mai indicat de utilizat: recursivitatea sau iterația?

- Algoritmii recursivi sunt potriviți pentru
  - probleme care utilizează formule recursive
  - pentru prelucrarea structurilor de date definite recursiv (liste, arbori)
  - mai eleganți şi mai simplu de înțeles şi verificat
  - uneori mai greu de dedus relația de recurență
- Iterația este preferată (uneori) din cauza
  - vitezei mai mari de execuție
  - memoriei mai reduse
- Exemplu
  - varianta recursivă a funcției Fibonacci duce la 15 apeluri pentru n=5
  - varianta iterativă este mult mai performantă

Exemplu: Fibonacci

```
// fib(n) = 1, dacă n=0 sau n=1
// fib(n) = fib(n-2) + fib(n-1), dacă n>1

#include<stdio.h>

int fibo_it (int n){
    int f1=1,f2=1,fn=1, i;
    if (n==0 || n==1) return 1;
    for (i=2; i<=n; i++) {
            fn=f1+f2;
            f2=f1;
            f1=fn;
    }
    return fn;
}
```

Exemplu: Fibonacci

```
int fibo (int n) {
    if ( n==0 || n==1 ) return 1;
    return fibo (n-2) + fibo (n-1);
    }

int main (){
    int n;
    scanf ("%d", &n);
    printf ("%d %d", fibo(n), fibo_it(n));
    fflush (stdin);
    getchar ();
    return 0;
}
```

Tehnica eliminarii recursivității

- Orice program recursiv poate fi transformat în unul iterativ, dar algoritmul său poate deveni mai complicat şi mai greu de înțeles.
- De multe ori, soluția unei probleme poate fi elaborată mult mai uşor, mai clar şi mai simplu de verificat, printr-un algoritm recursiv.
- pentru implementare, poate fi necesară transformarea algoritmului recursiv în unul nerecursiv, în situațiile:
  - soluția problemei trebuie scrisă într-un limbaj nerecursiv; un caz particular sunt compilatoarele ce "traduc" un program recursiv dintr-un limbaj de nivel înalt în cod maşină (nerecursiv)
  - varianta recursivă ar duce la o viteză de execuție şi spatiu de memorie prea mari, transformarea în una nerecursivă eliminând dezavantajele.
- În scrierea unei varianta nerecursive, trebuie parcurşi toti paşii implicați în varianta recursivă, prin tehnici nerecursive.

Eliminare recursivitate pentru funcții cu un singur apel recursiv, ca ultimă instrucțiune

- Înlocuind instrucțiunea *if* cu o instrucțiune repetitivă (*while, for, do)*
- Recursiv:

```
double putere(double x, int n) {
    if (n==0) return 1;
    return n*putere(x, n-1);            // x^n=x*x^(n-1)
    }
```

- Iterativ:

```
double putere(double x, int n) {
    double p=1;
    if (n==0) return 1;
    while (n) {
            p=p*x;
            n--;
    }
    return p;
}
```

Eliminare recursivitate, caz general

- Recursivitatea implică folosirea a cel puțin unei stive. La fiecare apel recursiv sunt depuse în stivă date care sunt extrase la revenirea din acel apel.
- Funcțiile recursive cu mai multe apeluri sau cu un apel care nu este ultima instrucțiune pot fi rescrise iterativ numai prin folosirea unei stive.
  - Datele pentru un apel se organizează într-o structură
  - Apel: introducerea în stivă a unei structuri
  - Revenirea din funcție: extragerea structurii din stivă
- Stiva: Last în First Out
  - Implementare utilizând un vector
  - Adăugare şi extragere de la sfârşit vector
- Această stivă poate fi un simplu vector local funcției

Exemplu: afişare în binar

```
void binar ( int n) {
    int c[32], i;            // c este stiva de cifre
   // pune resturi în stiva c
   i=0;
   while ( n>0) {
       c[i++]=n%2;
       n=n/2;
   }
    // descarca stiva: scrie vector în ordine inversa
   while (i>0)
        printf ("%d",c[--i]);
}
```

Exemplu

- Program care citeşte caracterele tastate până la un blanc, tipărindu-le apoi în ordine inversă.
- Varianta recursiva:

```
void invers_car(void){
    char car;
    if ( (car=getche()) != ' ‘ )  invers_car();
    putchar(car);
}
```

- Varianta nerecursiva

```
void invers_car(void){
    char car;
   //initializeaza stiva
    while( (car=getche()) != '  ')     push(car);
    while (!stiva_goala() ) {
        pop (car);
        putchar (car);
    }
}
```

Algoritmi de traversare şi inversare a unei structuri

- Traversarea / inversarea unei structuri înseamnă efectuarea unor operații oarecare asupra tuturor elementelor unei structuri în ordine directă / inversă
- Mai uzuale sunt variantele iterative, caz în care inversarea echivalează cu două traversări directe (o salvare în stivă urmată de parcurgerea stivei)
- Variantele recursive sunt mai elegante şi mai concise.
- Se pot aplica structurilor de tip tablou, lista, fişier şi pot fi o soluție pentru diverse probleme (transformarea unui întreg dintr-o baza în alta, etc).

Formă generală

- Parcurgere directă:
  - apelul inițial al funcției se face cu primul element al structurii

```
void traversare ( tip_element element ) {
        prelucrare ( element );
        if ( element != ultimul_din_structura )
                traversare ( element_urmator ) ;
   }
```

- Parcurgere inversă:
  - apelul inițial al funcției se face cu primul element al structurii

```
void inversare ( tip_element element ) {
      if ( element != ultimul_din_structura )
                traversare ( element_urmator );
        prelucrare (element);
   }
```

Exemplu

- Funcție recursivă ce determină elementul maxim dintr-un vector

```
// maxim dintre doua valori
double max2 (double a, double b) {
    return a > b ? a:b;
}

// maxim dintr-un vector
double maxim (double a[ ], int n) {
     if (n==1)
          return a[0];
      else
          return max2 (maxim (a,n-1), a[n-1]);
}
```

Algoritmi care implementeaza definiții recursive

- O definiție recursivă e cea în care un obiect se defineşte prin el însuşi.
- Definiția conține:
  - o condiție de terminare, indicând modul de părăsire a definiției
  - o parte ce precizează definirea recursivă propriu-zisă
- Exemple:
  - algoritmul lui Euclid de aflare a c.m.m.d.c.,
  - calcul combinări
  - ridicarea la o putere întreagă prin înmulțiri repetate
  - definirea recursivă a unei expresii aritmetice
  - calculul valorii unui polinom

Exemple

- Algoritmul lui Euclid:
  - condiția de terminare: a%b == 0
  - relația de recurență:

    cmmdc(a,b) = cmmdc(b,a%b)

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

Algoritmi de divizare - "divide and conquer"

- Constă în:
  - descompunerea unei probleme complexe în mai multe subprobleme
  - subprobleme a căror rezolvare e mai simplă
  - din soluțiile subproblemelor se poate determina soluția problemei inițiale

- Exemple:
  - determinarea minimului şi maximului valorilor elementelor unui tablou
  - căutarea binară
  - sortare Quicksort
  - turnurile din Hanoi

Algoritm de divizare general

```
void rezolva (problema x) {
        if (x e divizibil în subprobleme) {
            //divide pe x în parti x1,...,xk
            rezolva(x1);
            //...
            rezolva(xk);
            //combina solutiile partiale intr-o solutie
        }
        else
            //rezolva pe x direct
}
```

Funcție recursivă de căutare binară într-un vector ordonat

- reduce succesiv porțiunea din vector în care se caută, porțiune definită prin doi indici întregi

```
//caută b între a[i] şi a[j]
 int cautb (int b, int a[], int i, int j) {
  int m;                                  // indice median intre i şi j
  if ( i > j)           // dacă indice inferior mai mare ca indice superior
    return -1;                            // atunci b negasit în a
  m=(i+j)/2;                              // m = indice intre i şi j (la mijloc)
  if (a[m]==b)      return m;
  if (b < a[m])                          // dacă b în prima jumatate
    return cautb (b,a,i,m-1);           // atunci se cauta intre a[i] şi a[m-1]
  else                                   // dacă b în a doua jumatate
     return cautb (b,a,m+1,j);          // atunci se cauta intre a[m+1] şi a[j]
 }
```

Observație

- Varianta recursivă a unor funcții poate necesita un parametru în plus față de varianta nerecursivă pentru aceeaşi funcție, dar diferența se poate elimina printr-o altă funcție:

```
 // funcție 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);
}
```

Recursivitate mutuală sau indirectă

- Între două sau mai multe funcții f1 şi f2, de forma f1→ f2 → f1→ f2 → …
- Trebuie declarate funcțiile apelate, deoarece nu se pot evita declarațiile prin ordinea în care sunt definite funcțiile.
- Exemplu:

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

Recursivitate mutuală sau indirectă

- Altă variantă de scriere:

```
// declaratii funcții
    void f1 ( );
    void f2 ( );

// definiții funcții
    void f1 ( ) {
        …
        f2( );        // apel f2
    }
    void f2 ( ) {
        …
        f1( );        // apel f1
    }
```

Exerciții

- Să se scrie câte o funcție recursivă pentru:
  a) transformarea unui întreg din baza 10 în altă bază b dată

  b) tipărirea elementelor unui tablou în ordine inversă
  c) copierea în ordine inversă a liniilor dintr-un fişier text în altul

  d) calculul valorii unui polinom cu coeficienți întregi pentru o valoare x.