Recursivitate

Funcții recursive procedurale

Să analizăm următorul program:

```
#include <iostream>
using namespace std;

void afisare(int n) {
    if (n!=0) {
        cout<<n;
        afisare(n-1);
    }
}

int main () {
    afisare(3);
    return 0;
}
```

Observăm că la definirea funcției, în corpul său, ea se autoapelează. Acesta este principalul mod în care se realizează recusivitatea. Vom vedea ulterior și alt mod, acela când o funcție ajunge să se autoapeleze nu direct ci prin intermediul altei funcții (recursivitate indirectă).

Să analizăm efectul acestui program. Majoritatea oamenilor dau răspunsul corect: se afișează `321`.

Analizăm și exemplul următor:

```
void afisare(int n) {
    if (n!=0) {
        cout<<n;
        afisare(n-1);
        cout<<n;
    }
}
```

Rămâne contextul de mai sus, am modificat doar funcția, adăugând o instrucțiune de afișare după autoapel.

În acest caz, majoritatea oamenilor nu dau răspunsul corect :D . Iată în continuare o analiză detaliată, pas cu pas, a ceea ce se întâmplă:

Ce se întîmplă	***Memoria***	***Ecranul***
Se apelează funcția cu parametrul `3`. Ca la orice începere de apel de funcție, se alocă mai întâi memorie pentru datele locale, adică o variabilă numită `n`, cu valoarea `3`, apoi se începe executarea instrucțiunilor. În acest caz condiția de la `if` este adevărată așa că se intră pe prima ramură, se	<table><tr><td>Apel `afisare(3)`</td><td>n</td><td>3</td></tr></table>	3

afișează `3`, apoi urmează instrucțiunea de apel de funcție (autoapel).		
Acum se face iarăși ceea ce necesită un apel de funcție: se alocă memorie pentru datele locale, apoi se începe executarea codului. Observați din desenul alăturat că memoria alocată pentru datele locale ale apelului anterior nu s-a eliberat deoarece apelul `afisare(3)` nu s-a încheiat încă. Și în acest caz este adevărată expresia de la `if` așa că se intră pe prima ramură, se afișează `2` și urmează autoapelul `afisare(1)`.	Apel `afisare(2)` n 2 Apel `afisare(3)` n 3	`32`
Ținând cont de cele explicate mai sus, memoria arată ca în tabelul alăturat, se afișează `1`, apoi se face iarăși autoapel.	Apel `afisare(1)` n 1 Apel `afisare(2)` n 2 Apel `afisare(3)` n 3	`321`
Apelul `afisare(0)`, după ce își alocă memorie pentru datele locale (un `n` cu valoarea `0`), începe să își execute și el codul. Acum condiția de la `if` este falsă și s-ar merge pe `else`. Dacă ar fi fost ceva de executat acolo s-ar face acum, însă în exemplul nostru nu avem `else`, așa că se ajunge la acolada de final pentru corpul funcției și aceasta se termină. Știm că atunci când o funcție se termină ea își eliberează memoria alocată la începutul apelului.	Apel `afisare(0)` n 0 Apel `afisare(1)` n 1 Apel `afisare(2)` n 2 Apel `afisare(3)` n 3	`321`
Dar se termină doar apelul `afisare(0)`, nu toate celelalte, deci memoria va arăta ca alăturat. Apelul `afisare(0)`, care tocmai s-a terminat, a apărut în cadrul apelului `afisare(1)` care încă nu s-a încheiat și care își va continua acum executarea cu instrucțiunea de după apelul tocmai terminat `afisare(0)`. Aceasta este a doua instrucțiune `cout<<n`. Dar cât este acum `n` ? Răspuns:`1` deoarece suntem înapoi în apelul `afisare(1)`. De altfel, vedem asta și în tabelul alăturat uitându-ne în vârful stivei. Așadar se afișează pe ecran `1`.	Apel `afisare(1)` n 1 Apel `afisare(2)` n 2 Apel `afisare(3)` n 3	`3211`
După această instrucțiune de tipărire se ajunge la acolada de final a instrucțiunii `if` din prima ramură a apelului `afisare(1)`. Astfel se termină `if` și cum după el nu mai este altceva între acoladele funcției, se termină și acest apel al funcției. Deci, se eliberează iar memoria alocată la acest apel, adică din vârful stivei, aceasta ajungând să arate ca în tabelul alătural. Am revenit deci după locul în care s-a făcut apelul tocmai încheiat `afisare(1)` . Adică, la instrucțiunea a doua `cout<<n` din apelul `afisare(1)`. Se tipărește așadar pe ecran `2`, apoi, ca și anterior, apelul `afisare(2)` se termină.	Apel `afisare(2)` n 2 Apel `afisare(3)` n 3	`32112`
Ținând cont de explicațiile de mai sus, acum se ajunge să se afișeze `3`, din apelul `afisare(3)`.	Apel `afisare(3)` n 3	`321123`

Se încheie și apelul `afisare(3)`, odată cu el și executarea funcției recursive, iar stiva ajunge goală, la fel ca la început.		

Se observă așadar o simetrie, adică întâi se afișează `321`, apoi `123`, acestea fiind valorile parametrilor care ajung în stivă, prima dată în ordinea de parcurgere, apoi în ordine inversă.
Vom adăuga funcției și ramura `else`, unde tipărim pe `n`. Adică:

```
void afisare(int n) {
    if (n!=0) {
        cout<<n;
        afisare(n-1);
        cout<<n;
    } else
        cout<<n;
}
```

Ținând cont de analiza detaliată de mai sus, se ajunge să se tipărească și pe ramura `else` ceva, adică `0`. Acest lucru se realizează după ce s-a terminat urcarea în stivă și înainte să se înceapă coborârea, adică după ce s-au parcurs parametrii într-o ordine și înainte să se înceapă parcurgerea lor invers. Astfel avem: `3210123`.

Putem trage de pe acum concluzia: prin recursivitate avem posibilitatea parcurgerii unui set de date de două ori: o dată prin instrucțiunile de pe ramura cu autoapelul, aflate înainte de autoapel, apoi prin cele de pe ramura cu autoapelul aflate după autoapel, în ordine inversă a valorilor parametrilor față de prima parcurgere. Între cele două parcurgeri se execută o singură dată ce se află pe ramura else.

Înainte de a trage și alte concluzii, să analizăm ce se întâmplă și în cazul când funcția arată ca mai jos:

```
void afisare(int n) {
    cout<<n;
    afisare(n-1);
}
```

Se tipărește: `3210-1-2-3-4-5` ...

Majoritatea oamenilor spun că se ciclează infinit și într-un fel este adevărat. La o analiză mai atentă, ținând cont de faptul că la fiecate autoapel se alocă memorie nouă în stivă, și observând că niciodată vreun autoapel nu se termină, tragem căncluzia că niciodată nu se eliberează memorie și se tot alocă. Atunci când spațiul rezervat zonei de stivă se epuizează programul se termină cu eroare la executare (Stack overflow error).

Trebuie deci avut grijă să condiționăm autoapelul și să ne asigurăm că ajungem cândva încât parametrii fac să nu se mai execute autoapel, moment în care începe întoarcerea, coborârera în stivă.

Există chiar o teoremă care afirmă că oricărei scrieri recursive îi corespunde una repetitivă și reciproc.

Astfel, putem spune că, la modul general, o funcție recursivă este de forma:

```
Funcție (parametri) {
    if (conditie) {
        Instrucțiuni1
        Funcție(valori urmatoare ale parametrilor)
        Instructiuni2
    } else
        Instructiuni3
}
```

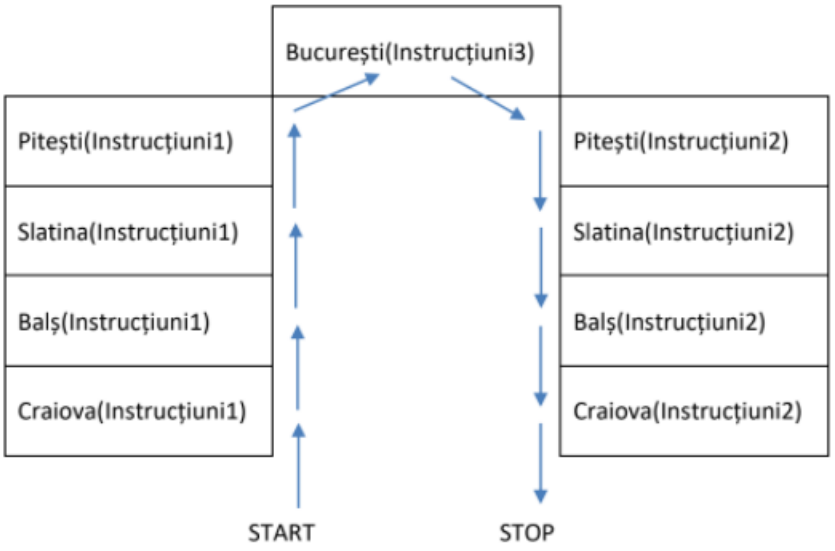
- Se execută așadar de mai multe ori numai `instrucțiuni1`, cât timp valorile parametrilor fac condiția adevărată.
- Se execută apoi, o singură dată `instrucțiuni3`, în momentul în care valorile parametrilor fac condiția falsă.
- Se execută apoi `instrucțiuni2`, de același număr de ori ca și `instrucțiuni1`, dar în ordine inversă a valorilor parametrilor ca în cazul executărilor lui `instrucțiuni1`.

Altfel spus, liniarizând, avem:

```
Instrucțiuni1
Instrucțiuni1
...
Instrucțiuni1
Instrucțiuni3
Instrucțiuni2
Instrucțiuni2
...
Instrucțiuni2
```

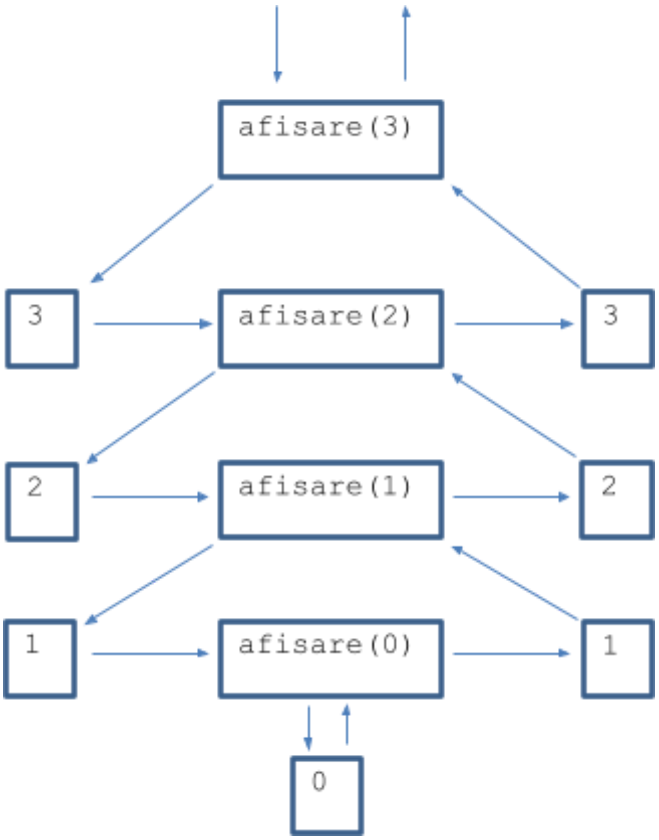
Eu când predau elevilor recursivitatea, pentru a sublinia traversarea de două ori, în ambele sensuri, a setului de date format de parametri, folosesc și următorul exemplu.

Dacă facem într-o zi, dus-întors, drumul Craiova-București, orașele pe care le traversăm, în ordine, sunt:



Iată și o altă modalitate de a reprezenta ce se întâmplă la un apel `afisare(3)` al următoarei funcții:

```
void afisare(int n) {
    if (n!=0) {
        cout<<n;
        afisare(n-1);
        cout<<n;
    } else
        cout<<n;
}
```



Obținem valorile afișate urmând săgețile.

Un alt procedeu pe care oamenii îl folosesc cu succes în evaluarea unui subprogram recursiv este cel al substituției. Astfel, pentru funcția scrisă mai sus avem:

- `afisare(3)` înseamnă: `3, afisare(2), 3`

Deci trebuie să vedem mai întâi ce efect are afișare(2), pentru a substitui:

- `afisare(2)` înseamnă: `2, afisare(1), 2`
- `afisare(1)` înseamnă: `1, afisare(0), 1`
- `afisare(0)` înseamnă: `0`.
- Acum ne întoarcem și substituim:
- `afisare(1): 1, afisare(0), 1  101`
- `afisare(2): 2, afisare(1), 2  21012`
- `afisare(3): 3, afisare(3), 3  3210123`

Am prezentat elementele cheie de la o funcție recursivă: faptul că autoapelul trebuie condiționat, că putem pune instrțiuni pe ramura cu autoapelul (înainte și după acesta) sau pe ramura cealaltă. Putem însă să adăugăm instrucțiuni chiar înaintea deciziei sau după ea. Un bun exercițiu în înțelegerea recursivității este să vă gândiți acum ce efect are un apel `afisare(3)` pentru funcția de mai jos:

```
void afisare(int n) {
    cout<<n; //1
    if (n!=0) {
        cout<<n;//2
        afisare(n-1);
```

```
        cout<<n;//3
    } else
        cout<<n;//4
    cout<<n;//5
}
```

Efectul este afișarea următorului șir de valori: `332211000112233`.

Așadar, instrucțiunea `1` fiind înainte de `if` ea se va executa la fiecare intrare în funcție deci și în cazul în care se continuă pe ramura cu autoapelul dar și în cazul când se merge pe cealaltă ramură. Așadar, dacă cea de dinainte de autoapel de pe ramura cu autoapelul se execută de trei ori, cea din fața lui `if` se execută odată în plus. Valorile îngroșate sunt cele afișate de ea: **3**3**2**2**1**1**0**00112233.

Simetric, instrucțiunea `5` se execută la fiecare ieșire din funcție, indiferent de ramura pe care se executase cod, deci se execută tot de `4` ori. Valorile îngroșate sunt cele afișate de ea: 33221100**0**1**1**2**2**3**3**.

Exerciții și probleme rezolvate

1. Scrieți o funcție recursivă care afișează cifrele unui număr în ordinea în care apar, separate prin câte un spațiu. De exemplu, un apel `f(352)` va avea ca efect afișarea: `3 5 2`.

```
void f(int n) {
    if (n!=0) {
        f(n/10);
        cout<<n%10<<" ";
    }
}
```

Traversarea cifră cu cifră se face prin autoapel cu valoarea obținută eliminând ultima cifră. Întrucât algoritmul standard de parcurgere a cifrelor unui număr le obține de la dreapta, noi decidem să afișăm după autoapel cifra curentă (ultima), tipărindu-le astfel în ordinea de la stânga la dreapta.

2. Scrieți o funcție recursivă cu parametru un număr `n` care afișează mai întâi descrescător valorile pare mai mici sau egale cu `n`, pe același rând, separate prin câte un spațiu, apoi afișează crescător, pe rândul următor valorile impare mai mici sau egale decât `n`. Se tipăresc doar valorile nautale nenule. De exemplu, un apel `f(8)` ar trebui să aibă efectul:

```
8 6 4 2
1 3 5 7
```

```
void f(int n) {
    if (n!=0) {
        if (n%2 == 0)
            cout<<n<<" ";
        f(n-1);
        if (n%2 != 0)
            cout<<n<<" ";
    } else
        cout<<"\n";
}
```

Noi traversăm mai întâi descrescător numerele de la `n` la `1` dar decidem să le afișăm doar pe cele pare, cu spații între ele. La revenire le afișam pe cele impare, tot cu spațiu între ele, deci le obținem în ordine crescătoare. Pe ramura `else`, al cărei cod se execută o singură dată, la mijloc, vom tipări un enter.

3. Considerăm următoarea secvență de cod:

```
cin>>n;
i=1;
while (i<=n) {
    cout<<i;
    i++;
}
```

Scrieți un program care rezlizează același lucru printr-o funcție recursivă.

```
#include <iostream>
using namespace std;
int n;
void f(int i) {
    if (i<=n){
        cout<<i;
        f(i+1);
    }
}
int main () {
    cin>>n;
    f(1);
    return 0;
}
```

Scopul acestui exercițiu este acela de a arăta un mod de "traducere" dintr-un cod repetitiv, mai ușor de scris, în unul recursiv.

Observăm:

- Contorul repetiției - `i` - este parametru la recursivitate (variabila a cărei valori valori se modifică de la un autoapel la altul);
- Modul de actualizare pentru contor este în legătură cu modul de autoapel: `i++` `f(i+1);`
- Condiția de la repetiție este aceeași cu cea de la condiționarea autoapelului;
- Inițializarea contorului (`i = 1`) are drept corespondent modul de apel inițial (`f(1)`).
- Instrțiunea subordonată repetiției este cea care se dorește a se executa la fiecare autoapel (`cout<<i`).

În momentul predării recursivității oamenii își pun întrebarea: de ce trebuie să mai învăț cum fac recursiv dacă am varianta repetitivă echivalentă? În plus, se observă ca la recursivitate se consumă și oarecare timp cu alocarea/eliberarea de memorie în stivă.

Motivul principal (și decisiv) pentru care este extrem de importantă recursivitatea este dat de ușurința cu care putem trata problemele ce necesită structuri neliniare precum arbori, grafuri. Codul de a vizita repetitiv astfel de structuri este infinit mai dificil de scris decât prin recursivitate.

4. Care este efectul următorului program?

```
#include <iostream>
using namespace std;
int maxim;
void f() {
    int x, s, y;
```

```
    cin>>x;
    if (x != 0) {
        s = 0;
        y = x;
        while (y) {
            s += y%10;
            y /= 10;
        }
        if (s > maxim)
            maxim = s;
        f();
        if (s == maxim)
            cout<<x<<" ";
    } else
        cout<<maxim<<"\n";
}
int main () {
    maxim = 0;
    f();
    return 0;
}
```

Acest program cere să introducem numere până se va da de la tastatură valoarea `0`. Apoi se afișează valoarea maximă pentru suma cifrelor vreunui număr introdus precum și numerele cu suma cifrelor maximă, pe rândul următor, separate prin spațiu și în ordine inversă a introducerii.

Cum de se obține asta? Funcția nu are parametri dar are variabile locale în care păstrează pentru revenire valori setate înainte de autoapel. Cele mai importante dintre acestea sunt `x` și `s`. În `x` se citesc toate numerele și se lasă în stivă. La calculul sumei cifrelor se folosește câte o copie a numărului citit, `y`. Valorile `s` calculate înainte de autoapel rămânând și ele în stivă (`s` este variabilă locală, deci fiecare autoapel cu `s` – ul lui), pot fi comparate direct cu `maxim` după autoapel, fără a mai fi necesară recalcularea sumei cifrelor.

Variabila `maxim` este globală, toate sumele obținute comparându-se cu valoarea din aceasta. Afișarea pe `else` a lui `maxim`, iar a celorlalte valori după autoapel, face să obținem pe ecran mai întâi maximul și apoi valorile care îl dau, în ordine inversă.

Din punct de vedere al memoriei ocupate, ținând cont că până se introduce `0` se tot alocă straturi în stivă, este ca și cum am fi alocat trei vectori, unul cu valori `s`, unul cu valori `x` și altul cu valori `y`.

Funcții recursive operand

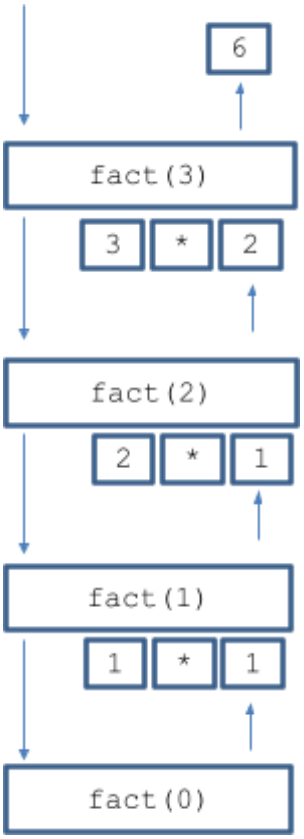
Pornim de la următorul exemplu, în care se calculează factorialul unui număr natural:

```
#include <iostream>
using namespace std;
int maxim;
int fact(int n) {
    if (n!=0)
        return n * fact(n-1);
    else
```

```
        return 1;
}
int main () {
    cout<<fact(3);
    return 0;
}
```

Să analizăm în detaliu ce se întâmplă de fapt.

Apelul `fact(3)` face să se intre pe prima ramură a lui `if`, așadar se va returna `n*fact(2)`. Pentru evaluarea acestei expresii este necasar să se calculeze `fact(2)`. Avem așadar întâi autoapel, apoi se va returna rezultatul. Similar, pentru calculul `fact(2)` se intră tot pe prima ramură a lui `if` și mai întâi se calculează `fact(1)` și apoi acest rezultat se înmulțește cu `2` etc. Iată, reprezentat cumva grafic acest mecanism:



Ordinea în care se fac înmulțirile efectiv este de la `1` spre `3`.

Care ar fi efectul funcției dacă nu am pune ramura `else`?

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

În acest caz, la executarea `fact(0)` nu s-ar executa `return`, adică suntem în cazul unei funcții operand din care se iese fără a se executa `return`. Cunoaștem că valoarea este una neprevăzută, ca în cazul unei date locale neinițializate de noi. Această valoare se înmulțește în continuare cu `1`, `2`, `3` iar în final se obține ceva de genul:

“valoare neprevăzută” `* 1 * 2 * 3`, așadar în final tot neprevăzut este rezultatul.

Concluzia este că în cazul funcțiilor recursive operand trebuie să avem prevăzut executarea de `return` indiferent de ramura pe care se intră.

Cu toate că nu are importanță practică ordinea în care tratăm cele două ramuri la recursivitate, programatorii tratează de cele mai multe ori mai întâi ramurile directe în cazul în care acestea există. Cum în cazul funcțiilor operand acestea sunt obligatorii, de regulă scrierea funcției de calcul al factorialului o întâlnim mai des astfel:

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

Spuneam în alt capitol că putem găsi o echivalență între trei lucruri:

- Funcțiile recursive
- Inducția matematică
- Șirurile recurente

Iată o analiză în sprijinul acestei afirmații.

Fie șirul factorialelor `1, 1, 2, 3, 6, 24, 120, …`

Acesta poate fi scris recurent:

$$f_n = \begin{array}{l} 1,\ dacă\ n = 0 \\ n * f_{n-1}\ dacă\ n > 0 \end{array}$$

Dacă ne gândim la modul în care citim relația de recurență de mai sus ne dăm seama că este cam totuna cu ordinea în care facem lucrurile la scrierea unei funcții recursive operand.

De multe ori, pentru scrierea unei funcții recursive operand este poate mai simplu să dăm semnificație numelui funcției ca și cum ar fi termenul general al unui șir recurent, apoi să scriem relația de recurență pentru acel șir și ulterior să o “traducem în limbaj”.

Iată un alt exemplu: Dorim să scriem o funcție recursivă care să caculeze suma cifrelor unui număr.

Mi întâi stabilim: $suma_n$ = suma cifrelor numărului `n`.

- Scriem apoi următarea relație de recurență:

$$suma_n = \begin{cases} n, \text{ dacă } n \text{ are o cifră} \\ suma_{n \text{ fără ultima cifră}} + \text{ultima cifră a lui } n, \text{ dacă } n \text{ are mai mut de o cifră} \end{cases}$$

- Traducem în C/C++ scriind o funcție pe baza relației de recurență:

```
int suma(int n) {
    if (n < 10)
        return n;
    else
        return suma(n/10) + n%10;
}
```

Și la funcțiile recursive, dar și la relațiile de recurență, pe lângă ramura generală avem și ramura directă, de pornire. Așa se întâmplă și la inducție: Rezolvăm cazul `n` folosindu-ne de cazul `n-1`, dar trebuie să avem și un caz rezolvabil direct.

Exerciții și probleme rezolvate

1. Scrieți o funcție recursivă care calculează numărul de cifre impare ale unui număr natural transmis ca parametru.

```
int cimp(int n) {
    if (n == 0)
        return 0;
    else
        if (n%2 == 1)
            return 1 + cimp(n/10);
        else
            return cimp(n/10);
}
```

Mai sus este rezolvarea corectă, dar merită discutată și următoarea situație: la o astfel de funcție unii elevi uită să trateze cazul care mai sus apare al treilea. Care aste efectul ? Cât timp sunt cifre impare la final, se face autoapel (notându-se ca la întoarcere să se adune `1`). La întâlnirea primei cifre pare nu se mai face autoapel dar nu se mai face nici `return`. Așa că se întoarce o valoare neprevăzută la care se adună atâția de `1` câte cifre impare sunt la finalul numărului. Deci rezultatul final este neprevăzut.

2. Scrieți o funcție recursivă operand care care verifică dacă un număr are toate cifrele impare. În caz afirmativ funcția trebuie să returneze `1`, în celălalt caz rezultatul său să fie `0`.

```
int verif(int n) {
    if (n < 10)
        if (n%2 == 0)
            return 0;
        else
            return 1;
    else
        if (n%2 == 1)
            return verif(n/10);
        else
            return 0;
```

```
}
```

Unii oameni scriu lucrurile ca mai jos:

```
int verif(int n) {
    if (n == 0)
        return 1;
    else
        if (n%2 == 1)
            return verif(n/10);
        else
            return 0;
}
```

Semnificația condiției `n==0` la algoritmii ce parcurg cifrele unui număr este că s-au terminat cifrele. La problema noastră de verificare, tot făcând autoapel când cifra este impară și întorcându-ne cu `0` când găsim una pară, înseamnă că dacă se termină cifrele trebuie returnat `1`. Însă în cazul particular că numărul inițial de testat este chiar `0`, varianta a doua ar returna `1` și nu ar fi corect. În rest, cele două sunt echivlente.

La ambele strategia folosită a fost că, dacă la pasul curent cifra este impară să lăsăm rezultatul în sarcina a ceea ce oferă autoapelul. Faptul că returnăm `0`, fără autoapel în cazul găsirii unei cifre face ca funcțiile să simuleze varianta repetitivă în care facem `break` la găsirea unui element ce contrazice presupunerea inițială.

Funcții recursive care întorc valori prin parametri

Să facem discuția în jurul următorului exemplu: Scrieți o funcție procedurală care are doi parametri, prin primul primește un număr iar prin al doilea oferă programului apelant factorialul numărului primit prin primul parametru. Am văzut mai sus cum se scrie o funcție operand care să calculeze factorialul:

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

Dacă prezentăm lucrurile mai didactic, observăm că acestea stau ca mai jos:

```
int fact(int n) {
    if (n==0)
        return 1;
    else {
        int rezultatAnterior = fact(n-1);
        return n * rezultatAnterior;
    }
}
```

Adică, dacă se poate face direct calculul, returnăm imediat, altfel, stocăm rezultatul autoapelului într-o variabilă, apoi compunem rezultatul apelului curent pe baza rezultatului autoapelului și pe baza contribuției la rezultat a apelului curent.

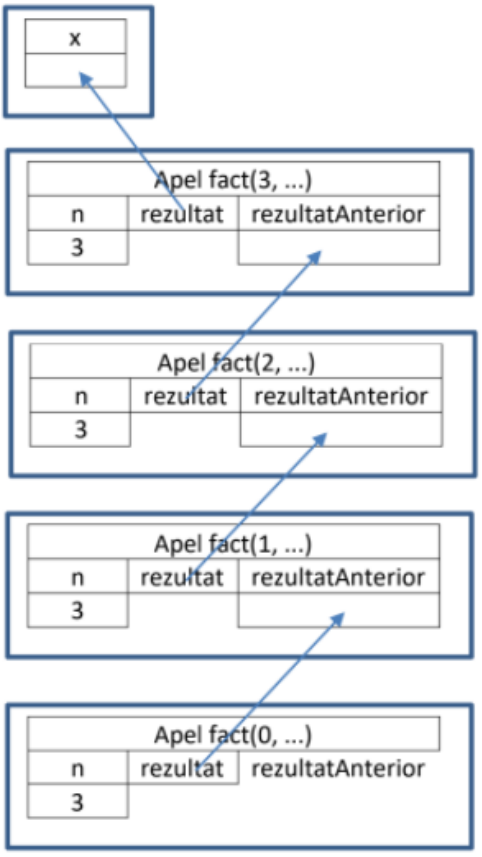
Dacă am face calculul cu funcție procedurală, în loc să returnăm, rezultatul îl lăsăm în parametrul corespunzător în loc să îl returnăm prin funcție:

```
void fact(int n, int &rezultat) {
    if (n == 0)
        rezultat = 1;
    else {
        int rezultatAnterior;
        fact(n-1, rezultatAnterior);
        rezultat = n * rezultatAnterior;
    }
}
```

În acest caz suntem obligați, ca lucrurile să fie bine la final, ca pe ambele ramuri să dăm o valoare corectă parametrului *rezultat*. Dacă în exemplul de mai sus nu am face asta pe prima ramură, ar fi greșită pornirea și s-ar produce exact „valoare neprevăzută” `* 1 * 2 * 3` ...

“Valoare neprevăzută” se datorează faptului că parametrul *rezultat* pe care noi nu îl setăm este chiar variabila *rezultatAnterior* de la penultimul autoapel care este declarată local și neinițializată.

Mai detaliat, lucrurile stau ca în desenul de mai jos:



Exerciții și probleme rezolvate

1. Scrieți o funcție recursivă care primește printr-un parametru un număr natural și care returnează prin alți doi parametri cifra maximă și pe cea minimă ale numărului primit. Scrieți și un program în care folosiți funcția.

```
#include <iostream>
using namespace std;
void calcul(int n, int &cmax, int &cmin) {
    if (n < 10) {
        cmax = n;
        cmin = n;
    } else {
        int cmaxAnt, cminAnt;
        calcul(n/10, cmaxAnt, cminAnt);
        if (n%10 > cmaxAnt)
            cmax = n%10;
        else
            cmax = cmaxAnt;
        if (n%10 < cminAnt)
            cmin = n%10;
        else
            cmin = cminAnt;
    }
}
int main () {
    int maxim, minim;
    calcul(132, maxim, minim);
    cout<<maxim<<" "<<minim;
    return 0;
}
```

Soluția implementează modul de lucru descris la prezentarea anterioară. Observăm că avem două rezultate oferite de funcție.

Recursivitate cu două sau mai multe autoapeluri

Conținutul de până acum al materialului cuprinde doar funcții recursive în care instrucțiunea de autoapel apare o singură dată. Este permisă apariția acestuia și de mai multe ori. Să analizăm următoarea funcție în contextul unui apel al său: `f(3);`

```
void f(int n) {
    if (n!=0) {
        cout<<n;
        f(n-1);
        cout<<n;
        f(n-1);
        cout<<n;
    } else
        cout<<n;
}
```

Practic, la forma generală cu un autoapel, dezvoltată în detaliu în prima parte a materialului, am adăugat pe ramura cu autoapelul încă un autoapel urmat de afișarea parametrului.

Majoritatea oamenilor se încurcă rău dacă încearcă să deducă ce se afișează urmărind pas cu pas sau folosind metoda cu simularea a ceea ce se întâmplă în stivă.

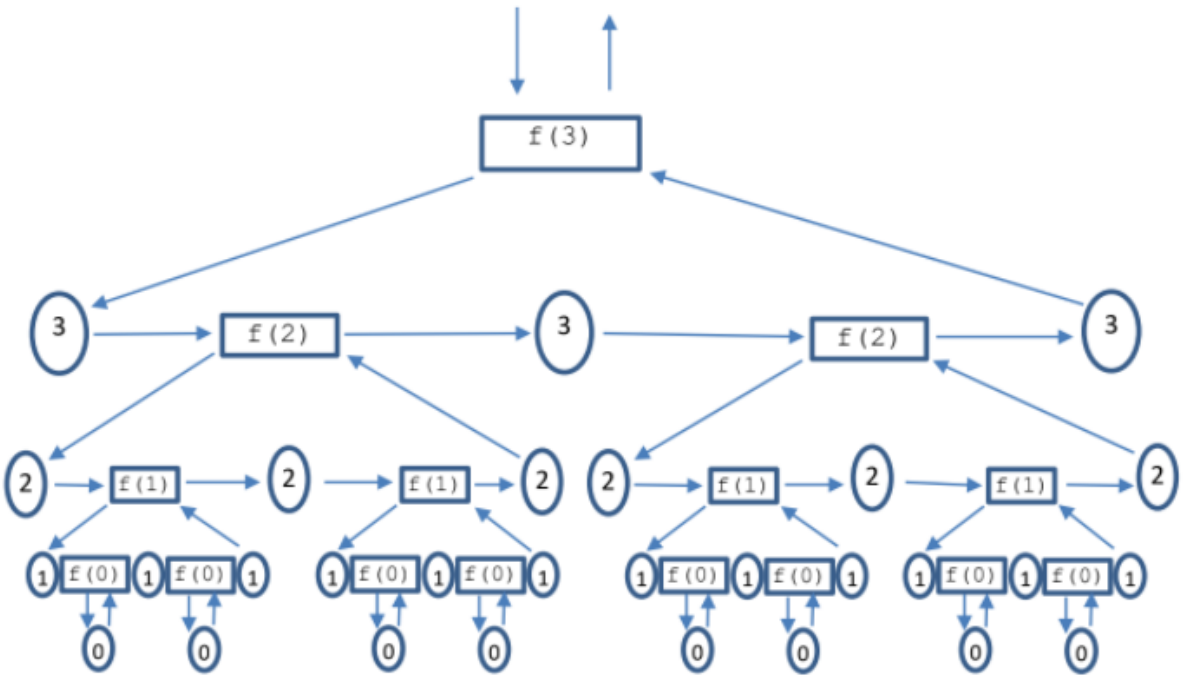
Pentru aceste situații se recomandă folosirea metodei substituției, sau chiar metoda desenării. Vom studia mai departe funcția cu fiecare dintre ele.

- `f(3): 3 f(2) 3 f(2) 3`
- `f(2): 2 f(1) 2 f(1) 2`
- `f(1): 1 f(0) 1 f(0) 1`
- `f(0): 0`
- Substituind invers obținem:
- `f(1): 1 f(0) 1 f(0) 1` `10101`
- `f(2): 2 f(1) 2 f(1) 2` `2101012101012`
- `f(3): 3 f(2) 3 f(2) 3` `32101012101012321010121010123`

Am putea continua, la `f(4)` obținem:
432101012101012321010121010123**4**32101012101012321010121010123**4**

Trebuie observat că de la o valoare la alta a parametrului, numărul de valori care se afișează se dublează. Acest lucru ne arată că numărul de operații în cazul recursivității cu mai multe autoapeluri crește exponențial. În acest caz numărul de operații este chiar de ordinul 2^n.

Iată și o analiză folosind metoda desenării:



Valorile de afișat se obțin urmând săgețile. Putem justifica natura exponențială a algoritmului prin faptul că dacă urcăm cu `1` valoarea parametrului trebuie practic să dublăm figura curentă.

Dar motivul principal pentru care am ținut să prezentăm și acest mod grafic este pentru a sublinia structura arborescentă care se formează prin această funcție recursivă cu două autoapeluri (aici ese vorba despre un arbore binar, adică fiecare nod are doi fii). Pe de altă parte, codul care produce acest arbore este foarte ușor de scris. Spuneam mai sus că unul dintre avantajele recursivității este ușurința cu care parcurgem structuri neliniare și sprijinim afirmația cu acest exemplu.

Exerciții și probleme rezolvate

1. Cunoaștem că șirul lui Fibonacci este definit recurent astfel:

$$f_n = \begin{cases} 1, \ dacă \ n \ este \ 1 \ sau \ 2 \\ f_{n-1} + f_{n-2}, \ dacă \ n > 2 \end{cases}$$

Scrieți o funcție recursivă care calculează termenul `n` al acestui șir.

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

Soluția de mai sus reprezintă "traducerea" relației de recurență în recursivitate așa cum am arătat într-un paragraf anterior că putem face. Deși este poate cel mai comod mod în care putem calcula un termen din șirul lui Fibonacci, este și unul destul de neperformant. Fiind vorba de recursivitate cu două autoapeluri timpul de calcul devine exponențial. Ne putem imagina că se formează un arbore de autoapeluri ca acela desenat mai sus și același termen cu corespondent mai jos în arbore ajunge să se calculeze, inutil, de foarte multe ori (mereu cu același rezultat). Pentru această problemă nu se recomandă soluția de mai sus ci cunoscuta abordare repetitivă în care se păstrează mereu ultimii doi termeni calculați.