Subprograme (Funcții)

Sunt situații când dorim să folosim aceeași bucată de cod în mai multe locuri într-un program. În general limbajele de programare oferă facilități pentru a permite scrierea o singură dată a codului de repetat și invocarea sa de câte ori se dorește, eventual parametrizat. În limbajul C/C++ acest lucru se realizează prin intermediul funcțiilor.

Odată cu dezvoltarea limbajului au apărut biblioteci cu mult cod scris deja și pe care programatorul îl poate folosi doar prin includerea în program a bibliotecii corespunzătoare (`#include` ...).

Ne mai putem gândi la funcții ca la o extensie a setului de operatori. Astfel, dacă limbajul oferă deja semne pentru operațiile uzuale (+, −, % ...) cum am putea realiza alte operații precum radical, ridicare la putere etc, știind că nu avem un semn dedicat pentru asta? Răspunsul vine de la funcții.

Funcțiile sunt așadar mici programe, cu datele lor, cu codul lor, care realizează o anume sarcină. În C/C++ funcțiile se pot clasifica în mai multe moduri, dar în primul rând după valoarea returnată și anume, dacă returnează (oferă în afară direct prin ele) sau nu vreo valoare. Avem astfel:

- Funcții **operand** – cele care returnează o valoare și astfel pot fi folosite (apelate) în cadrul unor expresii;
- Funcții **procedurale** – cele care nu returnează valori și se apelează ca instrucțiune separată;

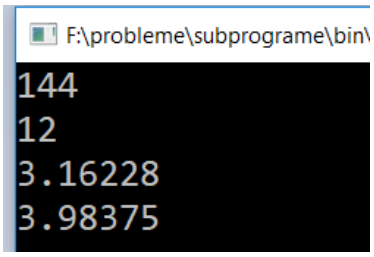
Funcții operand

Am spus că bibliotecile limbajului sunt din ce în ce mai ofertante cu cod scris deja și care apoi să poată fi folosit. Spunem așadar că dispunem în acest mod de funcții predefinite. În materialul de față nu ne concentrăm pe funcțiile predefinite ci pe modul în care scriem altele noi. Mereu va fi nevoie de asta, indiferent de cât de largă este oferta de funcții predefinite, apărând des situații particulare în orice program.

Vom porni totuși de la un exemplu de funcție predefinită.

```
#include <iostream>
#include <cmath>
using namespace std;
int n;
double aux;
int main () {
    cin>>n;
    cout<<sqrt(n)<<"\n";
    cout<<sqrt(10)<<"\n";
    aux = sqrt(12 + sqrt(14));
    cout<<2*sqrt(aux)<<"\n";
    return 0;
}
```

Dacă se citește de la tastatură `144`, în fereastra de consolă va apărea:



Observăm că sunt cinci apeluri ale codului care calculează radicalul, realizate acolo unde este permisă folosirea unei date reale (în cadrul unei expresii de atribuire, în cadrul unei instrcțiuni de afișare etc)

Funcția `sqrt` poate fi așadar privită ca un operator unar (deoarece se aplică unei singure date) și care oferă o valoare de tip real. Din exemplul de mai sus putem trage deja câteva concluzii despre modul de apel al unei funcții (chiar dacă este predefinită – principiul este același), anume: trebuie cunoscut numele funcției, tipul valorii returnate precum și la ce fel de date se poate aplica. Datele scrise între paranteze la apel sunt așadar date transmise către funcție – un fel de date de intrare pentru aceasta – ele se numesc parametri, iar valoarea returnată este data de ieșire.

Ce s-ar întâmpla dacă toate apelurile sqrt le-am înlocui cu apeluri `sumaCifre`. Adică, am scrie în cod `sumaCifre` peste tot pe unde apare `sqrt`:

```
#include <iostream>
#include <cmath>
using namespace std;
int n;
double aux;
int main () {
    cin>>n;
    cout<<sumaCifre(n)<<"\n";
    cout<<sumaCifre(10)<<"\n";
    aux = sumaCifre(12 + sumaCifre(14));
    cout<<2*sumaCifre(aux)<<"\n";
    return 0;
}
```

Dacă avem norocul să existe într-o bibliotecă dintre cele incluse vreo funcție predefinită cu acest nume, și care totodată să accepte un singur parametru și să returneze o valoare care poate fi folosită în expresiile de mai sus, atunci programul va rula. Dar cel mai probabil nu este așa deci vom primi eroare de compilare.

Ce ar fi de făcut dacă dorim să facem să funcționeze ce este mai sus ? Adică în loc de radicali să facem suma cifrelor? Completăm codul ca mai jos (am adăugat înainte de `main`):

```
#include <iostream>
using namespace std;
int n;
double aux;

int sumaCifre(int n) {
    int r;
    r = 0;
    while (n!=0) {
        r += n%10;
```

```
        n /= 10;
    }
    return r;
}

int main () {
    cin>>n;
    cout<<sumaCifre(n)<<"\n";
    cout<<sumaCifre(10)<<"\n";
    aux = sumaCifre(12 + sumaCifre(14));
    cout<<2*sumaCifre(aux)<<"\n";
    return 0;
}
```

În acest caz lucrurile funcționează, iar dacă voi introduce `144`, se vor afișa, câte una pe rând, valorile: `9`, `1`, `16`.

Observați că nu a mai fost necesară includerea bibliotecii cmath întrucât nu am mai folosit nimic de acolo. Totodată, dacă lăsam și acea linie de cod nu s-ar fi întâmplat nimic, doar că ar fi fost inutil și poate să crească și dimensiunea fișierului executabil.

Așadar am scris o singură dată codul de calcul pentru sumei cifrelor unui număr (am definit funcția) și am apelat-o de cinci ori.

Să ne concentrăm asupra modului de definire a funcției (am mai amintit și o să o mai facem, definierea se întâmplă o dată, este momentul construirii funcției, iar apelul, care se face de câte ori dorim, este momentul folosirii funcției).

```
int sumaCifre(int n) {
    int r;
    r = 0;
    while (n!=0) {
        r += n%10;
        n /= 10;
    }
    return r;
}
```

Acest lucru se face în afara funcției main (în general, în afara altei funcții) și respectă regula cerută de identificatori, de a fi definiți înaintea locului în care vor fi folosiți.

- Cuvântul `int` de la început indică tipul rezultatului funcției. Acest lucru este foarte important, apelul putându-se apoi face doar acolo unde este permisă folosirea unei valori de acest tip.
- Urmează `sumaCifre`, așadar un identificator care reprezintă numele funcției. Acesta va fi folosit la apel.
- Apoi sunt parantezele rotunde, semnul distinctiv principal al funcțiilor. Între ele se scriu parametrii, dar vom vedea că putem avea și funcții fără parametri însă cu toate acestea parantezele rotunde sunt obligatorii atât la definire cât și la apel.
- Între parantezele rotunde se scriu parametrii funcției. Aceștia se indică sub forma `tip valoare`, sepatați prin virgulă, dacă sunt mai mulți. Așa cum am mai amintit, prin intermediul lor se pot transmite date către funcții. Vom vedea într-un subcapitol următor că avem și varianta ca prin parametri să returnăm în afară și alte valori decât cea pe care o returnează funcția.

Cele descrise mai sus formează antetul funcției. Cunoașterea acestuia este necesară celor care vor dori folosirea mai departe a funcției.

Între acolade urmează codul efectiv al funcției, numit și corpul funcției, unde observăm:

- Putem declara date locale, așa cum este `r`. Aceste date sunt vizibile doar în interiorul funcției și le folosim în procesul de transformare a parametrilor în rezultatul dorit. Aceste date locale, imediat după declarare nu se inițializează implicit cu `0` (ca în cazul datelor globale – cele declarate în afara altor funcții). Am făcut separat inițializarea cu valoarea `0` pentru `r`.
- Avem apoi instrucțiunile prin care realizăm prelucrarea dorită. Acestea lucrează cu datele locale și cu parametrii, dar vom vedea că pot apărea și date globale (acest lucru se evită de regulă).
- Avem și o instrucțiune separată, `return`. În general aceasta se folosește sub forma: `return expresie;` Expresia trebuie să fie de tipul rezultatului funcției. Executarea lui `return` face ca să se încheie imediat codul funcției și să se revină în programul apelant cu valoarea expresiei ca fiind cea returnată de funcție.

În termeni mai generali, modul de definire pentru o funcție este:

```
tip_rezultat nume_functie (lista de parametri) {
    declarații locale și instrucțiuni
}
```

Iată un alt exemplu după care vom trage mai multe concluzii.

```
#include <iostream>
using namespace std;
int n, i;
int prim(int n) {
    int i;
    if (n < 2)
        return 0;
    for (i=2;i<=n/i;i++)
        if (n%i == 0)
            return 0;
    return 1;
}
int main () {
    for (i=1;i<=100;i++)
        if (prim(i) == 1)
            cout<<i<<" ";
    return 0;
}
```

Programul de mai sus afișează numerele naturale prime mai mici decât `100`. Analizând definirea funcției prim observăm:

- Nu a mai fost necesară folosirea unei variabile logice, `ok`, în care să notăm starea verificării: returnăm direct `0` dacă găsim vreun divizor care face ca numărul să nu fie prim. Putem scrie `return 1` la final, în afara vreunei condiții întrucât nu s-ar ajunge acolo dacă s-ar fi găsit divizori.
- Observăm că scăpăm de tratarea cazurilor particulare cu numerele mai mici decât `2` care nu sunt prime tot cu un `return`, la început. Așadar, putem evita să scriem restul de cod pe `else`, ceva similar cu folosirea de `break`/`continue` în structurile repetitive.

- Apelul funcției îl facem ca parte a condiției de la `if`, lucru care este în regulă pentru că se compară două numere, unul fiind valoarea de tip int returnată de funcție.
- Funcția are o variabilă locală numită `i`, iar apelul său se face într-un `for` care are un contor numit tot `i`. Ce se întâmplă de fapt ? Lucrurile sunt în regulă, variabila globală `i` nu mai este vizibilă în funcție atâta timp cât în intrior am declarat alta cu același nume. În funcție se va folosi variabila `i` locală. Dacă am fi omis linia `int i;` din interiorul funcției nu am fi primit eroare de complilare întrucât `i` folosit în funcție ar fi fost chiar variabila globală, lucrurile însă s-ar fi stricat, variabila globală `i` ajungând să fie contor în două foruri care prin apelul funcției ajung unul în altul.

Cu toare că nu am discutat în detaliu, am foloait deja ***apeluri ale funcției***. Acum este momentul să spunem mai clar cum se face asta:

- Apelul apare în orice loc unde poate apărea o dată de tipul rezultatului funcției.
- La apel se indică numele funcției iar între paranteze rotunde parametii de la apel. Acestea sunt expresii de tipul datei corespunătoare de la definirea funcției, valorile acestor expresii evaluându-se și fiind valorile inițiale ale parametrilor în corpul funcției.

În continuare vom analiza detaliat ce se întâmplă cu datele din memoria RAM atunci când rulează un program, în plus, când programul conține și apel de funcție.

Vom discuta pe următorul exemplu:

```
#include <iostream>
using namespace std;
int n, m;

int cmmdc(int a, int b) {
    int r;
    while (b != 0) {
        r = a%b;
        a = b;
        b = r;
    }
    return a;
}

int main () {
    cin>>n>>m; /// se introduc de la tastatura 26
20
    cout<<cmmdc(n, m)<<"\n";
    cout<<n<<" "<<m;
    cout<<cmmdc(12, 16)<<"\n";
    return 0;
}
```

Vom analiza pas cu pas efectul odată cu executarea fiecăreia dintre cele cinci linii de cod din `main`.

Analiza	***Memoria***	***Ecran***
Înainte de executarea primei linii de cod din main, programul a rezervat memorie RAM pentru datele sale globale, în acest caz `n` și `m`. Acest lucru se realizează în așa numita zonă de date. Vom vedea că ulterior, în	Zona de date `n` 0 `m` 0	

momentul apelului funcțiilor, acestea își vor rezerva și ele memorie, în altă zonă, numită stivă.

Zona de stivă

`cin>>n>>m;`
Se introduc datele de la tastatură și se memorează în variabilele asociate.

Zona de date

n	26
m	20

Zona de stivă

`cout<<cmmdc(n, m)<<"\n";`
Ceea ce se vede în dreapta completat în stivă este oglinda memoriei de imediat după începerea executării funcției, înainte să se execute codul funcției.
Observăm că se rezervă noi zone de memorie, pentru: parametrii, datele declarate separat local și chiar o zonă unde se va stoca valoarea returnată de funcție.
Diferența între parametri și celelalte variabile este că parametrii au și valori inițiale (ale valorilor expresiilor scrise în dreptul lor la apel). Astfel, nu mai este nicio legătură între `n` și `a`, de exemplu.
Pentru datele declarate local, dar și pentru valoarea returnată se alocă deci memorie, dar nu putem conta pe o anume valoare inițială. Valoarea inițială a lor este dată de rezultatul configurației de biți găsiți în zona rezervată. Acești biți nu sunt neapărat toți `0`, așa că și valorile inițiale ale acestor date sunt neprevăzute. Numerele tipărite în dreptul celor două zone neinițializate în exemplul nostru sunt fictive și sunt scrise pentru a sublinia că aceste valori nu sunt neapărat `0` cum se întâmplă cu datelor globale imediat după declarare.

Zona de date

n	26
m	20

Zona de stivă

a	26
b	20
r	12324
Valoarea de returnat	43567

```
while (b != 0) {
    r = a%b;
    a = b;
    b = r;
}
```
După executarea codului funcției, și imediat înainte de executarea lui `return a`, memoria arată ca în dreapta

Zona de date

n	26
m	20

Zona de stivă

a	2
b	0
r	0
Valoarea de returnat	43567

<table>
<tr>
<td>

`return a;`
Executarea lui return are ca efect plasarea în locul de returnat a valorii expresiei de după cuvântul `return`, în acest caz `2`. Această valoare va fi preluată de programul apelant și folosită mai departe după terminarea executării funcției.

</td>
<td>

Zona de date

<table>
<tr><td>n</td><td>26</td></tr>
<tr><td>m</td><td>20</td></tr>
</table>

Zona de stivă

<table>
<tr><td>a</td><td>2</td></tr>
<tr><td>b</td><td>0</td></tr>
<tr><td>r</td><td>0</td></tr>
<tr><td>Valoarea de returnat</td><td>2</td></tr>
</table>

</td>
<td></td>
</tr>
<tr>
<td>

Imediat după executarea codului funcției și după ce programul apelant și-a preluat valoarea returnată, memoria arată astfel:

Faptul că nu am desenat chenarele la tabel are semnificația că zonele respective nu mai sunt alocate, ele putând apoi fi date altor variabile.
Faptul că nu am mai scris numele datelor locale are semnificația că acestea nu mai sunt vizibile în programul principal după terminarea apelului funcției.
Valorile însă e posibil să rămână și tocmai din acest motiv este posibil ca variabile care se vor aloca în continuare și vor primi spațiu în aceste zone să preia valorile lăsate anterior.
Observăm că variabilele globale rămân nemodificate, ele fiind, așa cum am mai spus doar pe post de expresii de inițializare la datele locale în momentul apelului.
Întrucât apelul funcției a fost făcut ca parte a unei instrucțiuni de afișare, apare pe ecran valoarea `2`.

</td>
<td>

Zona de date

<table>
<tr><td>n</td><td>26</td></tr>
<tr><td>m</td><td>20</td></tr>
</table>

Zona de stivă

2
0
0
2

</td>
<td>2</td>
</tr>
<tr>
<td>

`cout<<n<<" "<<m;`
Pe ecran apar cele două valori citite de la tastatură. În zona de stivă nu am mai desenat nimic (nici valorile lăsate de apelul de funcție anterior) deoarece acea memorie nemaifiind alocată, nu trebuie să ne mai bazăm pe valorile din ea.

</td>
<td>

Zona de date

<table>
<tr><td>n</td><td>26</td></tr>
<tr><td>m</td><td>20</td></tr>
</table>

Zona de stivă

</td>
<td>2
26 20</td>
</tr>
<tr>
<td>

`cout<<cmmdc(12, 16)<<"\n";`
Se reia ce am descris la apelul anterior, adică se alocă iarăși memorie în stivă, se execută codul funcției, iar la final se eliberează memoria alocată de aceasta.

</td>
<td>

Zona de date

<table>
<tr><td>n</td><td>26</td></tr>
<tr><td>m</td><td>20</td></tr>
</table>

</td>
<td>2
26 20</td>
</tr>
</table>

<table>
<tr>
<td></td>
<td>Zona de stivă
<table><tr><td>a</td><td>12</td></tr><tr><td>b</td><td>16</td></tr><tr><td>r</td><td>43242</td></tr><tr><td>Valoarea de returnat</td><td>66464</td></tr></table></td>
<td></td>
</tr>
<tr>
<td>La terminare se afișează pe ecran rezultatul, 4.</td>
<td>Zona de date
<table><tr><td>n</td><td>26</td></tr><tr><td>m</td><td>20</td></tr></table>
Zona de stivă
4
0
0
4</td>
<td>2
26 20
4</td>
</tr>
<tr>
<td>După return 0; /// din main</td>
<td>Zona de date

Zona de stivă</td>
<td>2
26 20
4</td>
</tr>
</table>

O observație de final: Încă de la primul program C/C++ am vorbit despre funcția main. Dacă ne gândim mai bine observăm că ea este o funcție operand după toate regulile. Singurul lucru este că nu noi suntem cei care trebuie să o apelăm, ci sistemul de operare.

Exerciții și probleme rezolvate

1. Fie următoarea funcție:

```
int f(int n) {
    if (n%2 == 0)
        return 2*n;
}
```

a) Care este rezultatul apelului f(4) ?
b) Care este rezultatul apelului f(3) ?

Soluție

La subpunctul a) în corpul funcției se va executa return cu valoarea 8, deci acesta este răspunsul. În cazul subpunctului b) nu se execută return în corpul funcției, deci în locul unde programul apelant va căuta valoarea returnată nu se pune nimic și, cum am văzut mai sus (în tabelul cu explicarea modelului de memorie de la exemplul cu funcția cmmdc), acolo zona este neinițializată, așadar ne putem aștepta la orice rezultat. Trebuie avut deci grijă ca în cazul funcțiilor operand să se returneze mereu ceva.

2. Scrieți o funcție care primește ca parametru un număr natural n și care returnează cea mai mare valoare care se poate obține cu cifrele lui n așezate într-o ordine convenabilă.

Soluții:

```
int cmmnr(int n) {
    int v[12];
    int k = 0;
    while (n!=0) {
        k++;
        v[k] = n%10;
        n /= 10;
    }
    for (int i=1;i<k;i++)
        for (int j=i+1;j<=k;j++)
            if (v[i] > v[j]) {
                int aux = v[i];
                v[i] = v[j];
                v[j] = aux;
            }
    int r = 0;
    for (int i=k;i>=1;i--)
        r = r*10 + v[i];
    return r;
}
```

Strategia urmată mai sus este de a pune toate cifrele într-un vector, de a-l sorta descrescător și de a construi valoarea de returnat cu cifrele în această ordine.

Observați că putem declara un vector local unei funcții, acesta alocându-se în memorie în momentul apelului unei funcții ca și celelalte variabile locale. De asemenea, acesta eliberează memoria în momentul terminării executării funcției.

```
int cmmnr(int n) {
    int f[10];
    for (int i=0;i<=9;i++)
        f[i] = 0;
    while (n!=0) {
        f[n%10] ++;
        n /= 10;
    }
    int r = 0;
    for (int i=9;i>=0;i--)
        while (f[i] != 0) {
            r = r*10 + i;
            f[i]--;
        }
    return r;
}
```

Soluția de mai sus folosește un vector de frecvență pentru a sorta cifrele numărului de procesat.După cum vedem, putem declara un vector local unei funcții. Întrucât în componentele acestuia contorizăm numărul de apariții pentru fiecare cifră, este esențial să-i inițializăm valorile cu `0`.

3. Scrieți un program care calculează suma factorialelor cifrelor unui număr natural.

```
#include <iostream>
using namespace std;
int n;
int _fact(int n) {
    int p = 1;
    for (int i=1;i<=n;i++)
        p *= i;
    return p;
}
long long sumfactcif(int n) {
    if (n == 0)
        return 1;
    long long r = 0;
    while (n!=0) {
        r += _fact(n%10);
        n /= 10;
    }
    return r;
}
int main () {
    cin>>n;
    cout<<sumfactcif(n);
    return 0;
}
```

Am ales să folosim două funcții pentru a scoate în evidență posibilitatea ca în corpul unei funcții să apelez altă funcție. Funcția apelată trebuie să fie definită anterior (sau poate fi predefinită).

Funcții procedurale

Am clasificat la început funcțiile în operand și procedurale. Cele operand returnează o valoare și apelul lor se face în expresii. Cele procedurale nu returnează o valoare. Atunci la ce ne sunt utile ? Și cum le folosim?

Iată un exemplu:

```
#include <iostream>
using namespace std;
int n, m;
void afiseaza(int n, int m) {
    cout<<n<<"/"<<m<<"\n";
}
int main () {
    cin>>n>>m; /// se introduc de la tastatura 26 20
    afiseaza(n, m);
    afiseaza(12, 34);
```

```
    afiseaza(2*7, 4*9);
    return 0;
}
```

Se va afișa:

```
26/20
12/34
14/36
```

Ne putem imagina că dorim să afișăm într-un mod relevant o fracție. Funcția de mai sus este ceea ce ne dorim. Chiar dacă nu returnează o valoare, funcția procedurală are scopul ei, în acest caz fiind acela de a tipări ceva pe ecran, într-un anumit format.

Mai tot ce am discutat la funcțiile operand este valabil și aici. Iată însă și diferențele:

La definire, în loc de tipul rezultatului se scrie `void`. Este un cuvânt cheie care aici indică faptul că funcția este procedurală, nu returnează valoare.

- Apelul unei astfel de funcții se face ca instrucțiune separată, cum se poate vedea mai sus, și nu în cadrul unei expresii.
- Instrucțiunea return poate fi prezentă în funcțiile procedurale, dar fără a fi urmată de o expresie, ci doar de caracterul punct și virgulă. Dacă se ajunge la executarea ei funcția se încheie imediat.

O situație des întâlnită de utilizare a funcțiilor procedurale este de a organiza codul pe secțiuni. De exemplu, în concursuri, găsim de multe ori funcția main scrisă astfel:

```
int main() {
    citeste();
    rezolva();
    afiseaza();
}
```

Iată un program complet:

```
#include <iostream>
using namespace std;
int n, v[100];
void citeste() {
    cin>>n;
    for (int i=1;i<=n;i++)
        cin>>v[i];
}
void rezolva() {
    for (int i=1, j=n;i<j;i++,j--) {
        int aux = v[i];
        v[i] = v[j];
        v[j] = aux;
    }
}
void afiseaza() {
    for (int i=1;i<=n;i++)
        cout<<v[i]<<" ";
    cout<<"\n";
}
int main () {
```

```
    citeste();
    rezolva();
    afiseaza();
    return 0;
}
```

Programul de mai sus cere de la tastatură dimensiunea și elementee unui vector, îl oglindește, apoi îl afișează pe ecran în noua configurație. Funcțiile nu au parametri, ele lucrând cu vectorul `v` și cu numărul de elemente `n` ca variabile globale.

Funcții cu parametri transmiși prin referință

Să analizăm următorul program:

```
#include <iostream>
using namespace std;
int n, m;
void schimba(int a, int b) {
    int aux;
    aux = a;
    a = b;
    b = aux;
}
int main () {
    n = 2;
    m = 3;
    schimba(n, m);
    cout<<n<<" "<<m;
    return 0;
}
```

Cand sunt întrebați ce se va afișa, foarte mulți oameni se grăbesc și afirmă că `3` și apoi `2`. Nu este așa, se vor tipări `2` și `3`, în această ordine și separate prin spațiu întrucât interschimbarea se face între variabilele `a` și `b`, care, după cum am analizat mai sus se alocă în stivă și există doar pe durata executării funcției. Modificarea lor în funcție nu are niciun efect asupra lui `n` și `m`.

Mai exact, memoria evoluează astfel:

Înainte de apelul funcției:

Zona de date

n	2
m	3

Zona de stivă

Înainte ca funcția să își execute codul:

Zona de date

n	2
m	3

Zona de stivă

a	2
b	3
aux	12324

După executarea codului funcției, înainte ca aceasta să își elibereze memoria alocată:

Zona de date

n	2
m	3

Zona de stivă

a	3
b	2
aux	2

După terminarea apelului funcției

Zona de date

n	2
m	3

Zona de stivă

Așadar valorile lui `n` și `m` rămân nemodificate.

Modul de transfer al parametrilor către funcții folosit de la începutul capitolului până acum poatrtă numele de transmitere a parametrilor prin valoare.

Mai dispunem de un instrument pe care îl vom explica mai departe: transmiterea parametrilor prin referință. Aceștia sunt parametrii care sunt scriși precedați de caracterul `&` la definirea funcției. Vom modifica, pentru exemplul anterior ambii parametrii indicând că îi transmitem prin referință.

```
#include <iostream>
using namespace std;
int n, m;
void schimba(int &a, int &b) {
    int aux;
    aux = a;
    a = b;
```

```
    b = aux;
}
int main () {
    n = 2;
    m = 3;
    schimba(n, m);
    cout<<n<<" "<<m;
    return 0;
}
```

La apelul funcției, `a` și `b` nu vor fi variabile noi ci ele vor fi un fel de porecle (aliasuri) pentru parametii care sunt în dreptul lor la apel. Așadar, scrierea `a` în interiorul funcției face să se lucreze direct cu valoarea din zona de memorie asociată lui `n`, tot așa folosirea lui `b` în funcție face să lucrăm direct cu `m`.

Evident, atribuiri în funcție spre `a` și `b` fac să se modifice direct `n` și `m`. În acest caz subprogramul va interschimba deci valorile celor două variabile globale, ele devenind după apel: `n` cu valoarea `3` și `m` cu valoarea `2`.

Iată și o reprezentare grafică a transformărilor din memorie:

Înainte de apelul funcției:

Zona de date

n	2
m	3

Zona de stivă

Înainte ca funcția să își execute codul:

Zona de date

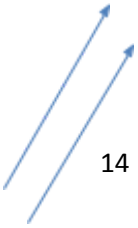
n	2
m	3

Zona de stivă

a	
b	
aux	12324

După executarea codului funcției, înainte ca aceasta să își elibereze memoria alocată:

Zona de date



n	3
m	2

Zona de stivă

a	
b	
aux	2

După terminarea apelului funcției

Zona de date

n	3
m	2

Zona de stivă

Așadar, în cazul parametrilor referință memoria rezervată în zona de stivă este mai mică.

Consecință a faptului că parametrilor referință trebuie să li se dea o zonă de memorie în care să își păstreze valoarea este că la apel, în dreptul unui parametru referință, trebuie scrisă obligatoriu o variabilă și nu alt fel de expresie (precum constante sau expresii ce conțin și operatori).

Un alt lucru important de care să ținem cont: parametrii transmiși prin referință reprezentând de fapt zonele de memorie ale unor variabile din afara funcției, ei au ca valoare inițială chiar valoarea acelei variabile în momentul apelului. Deci daca dorim să calculăm independent în ei o valoare în funcție, trebuie să îi inițializăm cu cât ne convine.

Acum putem spune că parametrii transmiși prin valoare reprezintă date de intrare pentru funcții iar cei transmiși prin referință sunt atât date de intrare cât și date de ieșire.

Exerciții și probleme rezolvate

1. Care este efectul următorului program?

```
#include <iostream>
using namespace std;
int n;
void schimba(int &a, int &b) {
    int aux;
    aux = a;
    a = b;
    b = aux;
}
int main () {
    n = 2;
    schimba(n, 3);
    cout<<n;
    return 0;
```

```
}
```

Răspunsul este: eroare de compilare. Acest lucru apare din cauza apelului cu constanta `3` în dreptul parametrului referință `a`.

2. Ce afișează programul următor?

```
#include <iostream>
using namespace std;
int x;
void f() {
    cout<<x<<" ";
}
int main () {
    x = 2;
    f();
    cout<<x;
    return 0;
}
```

Răspunsul este `2 2`. Variabila `x` este globală și este vizibilă și în funcție. Ea este singura variabilă a programului.

3. Ce afișează programul următor?

```
#include <iostream>
using namespace std;
int x;
void f() {
    int x;
    x = 3;
    cout<<x<<" ";
}
int main () {
    x = 2;
    f();
    cout<<x;
    return 0;
}
```

Răspunsul este `3 2`. Variabila `x` este redeclarată în funcție. Acest lucru este permis, ea este vizibilă în timpul executării codului funcției (timp în care la `x` global nu avem acces, cu toate că ea încă există). Așadar se afișează întâi `3` (`x` local) apoi, la terminarea executării funcției redevine vizibil `x` global care are tot valoarea `2`.

4. Care este efectul programului următor?

```
#include <iostream>
using namespace std;
int calcul(int a, int b) {
    int c;
    c = a+b;
    return a+b;
}
int main () {
    cout<<calcul(12, 30)<<" ";
    cout<<c;
```

```
    return 0;
}
```

Apare eroare de compilare. Motivul este că domeniul de vizibilitate al variabilei `c` este între acoladele funcției (acolo unde este declarată).

5. Care este efectul următorului program ?

```
#include <iostream>
using namespace std;
int n;
void calcul(int n, int &m) {
    m = n + m;
}
int main () {
    n = 2;
    calcul(n, n);
    cout<<n;
    return 0;
}
```

Funcția are dala locală (parametrul) `n` care ia valoarea expresiei din dreptul său de la apel (`2`). Celălalt parametru, fiind referință, reprezintă variabila din dreptul său de la apel, adică `n` global. În funcție atribuind lui `m` valoarea `n+m`, adică `4`, devine `4` variabila `n` globală. Așadar acesta este rezultatul afișat.

6. Următorul program arată cum folosim o funcție procedurală care să returneze rezultatul calculului printr-un parametru.

```
#include <iostream>
using namespace std;
void sumaCifre(int n, int &r) {
    r = 0;
    while (n!=0) {
        r += n%10;
        n /= 10;
    }
}
int main () {
    int x;
    sumaCifre(132, x);
    cout<<x;
    return 0;
}
```

Funcția primește prin parametrul `n`, transmis prin valoare, numărul pentru care se dorește calcularea sumei cifrelor și returnează prin parametrul referință `r` rezultatul. Ne amintim că la începutul materialului am calculat suma cifrelor unui număr și prin folosirea unei funcții operand. În acel mod, apelul este mult mai comod:

Cu funcție operand	Cu funcție procedurală

`cout<<sumaCifre(123);`	`int x;`
	`sumaCifre(123, x);`
	`cout<<x;`

Nu am prezentat comparația în scopul de a descuraja folosirea funcțiilor procedurale cu parametri referință ci pentru a arăta că de atunci când acestea returnează o singură valoare se recomandă folosirea celor operand. Situațiile când se dorește returnarea mai multor valori se pot trata doar prin folosirea cel puțin a unui parametru referință (sau variabile globale, dar despre oportunitatea folosirii sau nu a acestora am mai exemplificat și o vom mai face).

7. Scrieți o funcție care primește printr-un parametru un număr natural și prin alți doi patametri returnează cifra maximă și cifra minimă ale numărului primit.

```
void calcul(int n, int &maxi, int &mini) {
    if (n == 0) {
        maxi = mini = 0;
        return;
    }
    maxi = 0;
    mini = 9;
    while (n!=0) {
        if (n%10 > maxi)
            maxi = n%10;
        if (n%10 < mini)
            mini = n%10;
        n /= 10;
    }
}
```

Acesta este exemplul prin care observăm cum putem returna două valori calculate într-o funcție. Mai observăm și folosirea lui `return` (fără expresie în cazul funcțiilor void) pentru a trata mai compact cazul particular al numărului `0`.

8. Scrieți o funcție care primește un număr natural și care returnează trei valori, astfel: prin doi parametri referință cifra maximă resprectiv cifra minimă ale numărului și chiar prin funcție suma cifrelor numărului.

```
int calcul(int n, int &maxi, int &mini) {
    if (n == 0) {
        maxi = mini = 0;
        return 0;
    }
    maxi = 0;
    mini = 9;
    int suma = 0;
    while (n!=0) {
        if (n%10 > maxi)
            maxi = n%10;
        if (n%10 < mini)
            mini = n%10;
        suma += n%10;
        n /= 10;
    }
    return suma;
}
```

Acest exemplu ne arată că putem folosi parametri referință și la funcțiile operand. Evident, doar unul dintre rezultate îl putem returna prin fucnție, restul prin parametri referință.

Două exemple de apel pentru această funcție sunt:

```
int suma, maxim, minim;
suma = calcul(234, maxim, minim);
cout<<suma<<" "<<maxim<<" "<<minim;
```

```
int maxim, minim;
cout<<calcul(234, maxim, minim);
cout<<" "<<maxim<<" "<<minim;
```

Funcții cu parametri tablouri.

Pentru început analizăm cazul tablourilor unidimensionale. La definirea unei astfel de funții, patametrul de tip tablou poate fi anunțat în unul dintre modurile:

`tip nume[dimensiune]`	`tip nume[]`	`tip *nume;`

Cunoaștem că numele unui tablou este un pointer, iar al treilea mod de declarare specifică tabloul exact în acest mod. Al doilea mod este echivalent. Dacă anunțăm la definire parametrul ca în primul mod vom fi obligați să apelăm funcția cu un tablou cu exact aceeași dimensiune. Așadar se recomandă folosiriea unuia dintre al doilea și al treilea mod.

Un lucru deosebit de important de reținut este că la transferul către funcție a unui vector se transferă funcției doar pointerul către zona de date a tabloului și nu se face o copie a acestuia (ca în cazul variabilelor transmise prin valoare). Avantajul este că durează mai puțin începerea apelului (în cazul unui tablou mare s-ar consuma timp cu copierea). Trebuie avut în vedere că vrând-nevrând, modificarea componentelor tabloului în funcție are efect după terminarea funcției. Asta deci fără să fie nevoie să îl declarăm ca referință.

Nu trebuie să facem confuzie între parametrii tablouri și cazul când declarăm tabouri locale în funcție, când se alocă în stivă memorie pentru tabloul declarat local.

La apelul unei funcții cu parametru tablou, în dreptul acestor prametri se scrie doar numele tabloului, nu și alte semne.

Exerciții și probleme rezolvate

1. Care este efectul programului următor?

```
#include <iostream>
using namespace std;
int n, v[100];
void afisare(int v[], int n) {
    for (int i=1;i<=n;i++)
        cout<<v[i]<<" ";
}
int main () {
    afisare(v[], n);
    return 0;
}
```

Rezultatul este: Eroare la compilare. Motivul este că la apelul funcției cu parametru tablou, se scrie doar numele unei variabile de tip tablou, fără a fi însoțită de `[]`.

2. Scrieți o funcție cate primește ca parametri un tablou unidimensional cu valori de tip `int` și un număr `n`. În tablou se știe că valorile sunt memorate pe poziții între `1` și `n`. Funcția returnează valoarea maximă. Scrieți apoi un program care cere de la tastatură un număr natural `n` (care se dă par) și apoi `n`

numere naturale și care afișează mai întâi maximul din tot tabloul și apoi maximul din prima jumătate a tabloului. Realizați asta prin apeluri utile ale funcției scrise.

```
#include <iostream>
using namespace std;
int n, v[100];
int calcul(int v[], int n) {
    int maxim = v[1];
    for (int i=2;i<=n;i++)
        if (v[i] > maxim)
            maxim = v[i];
    return maxim;
}
int main () {
    cin>>n;
    for (int i=1;i<=n;i++)
        cin>>v[i];
    cout<<calcul(v, n)<<"\n";
    cout<<calcul(v, n/2)<<"\n";
    return 0;
}
```

Observăm că prin scrierea funcției am putut să invocăm codul de calcul al maximului din vector dintre poziția 1 și o alta dată, de două ori.

3. Considerăm următoarea secvență de program:

```
#include <iostream>
using namespace std;
int n, v[100], m, w[100];
.....................
int main () {
    citeste(v, n);
    citeste(w, m);
    afiseaza(v, n);
    afiseaza(w, m);
    cout<<maxim(v, 3)<<"\n";
    sorteaza(v, 1, n);
    sorteaza(w, 1, m/2);
    sorteaza(w, m/2+1, m);
    afiseaza(v, n);
    afiseaza(w, m);
    return 0;
}
```

Se dorește ca în locul punctelor de suspensie să scriem definițiile funcțiilor așa încât o executare să aibă următorul efect: se cer de la tastatură datele despre doi vectori (numărul de elemente și elementele, memorate începând cu poziția 1); se afișează cei doi vectori imediat după citire; se afișează maximul dintre primele trei elemente ale primului vector – presupunem că vom da dimensiunea sa cel puțin 3; se sortează crescător primul vector; se sortează crescător, independent, cele două jumătăți ale celui de-al doilea vector – presupunem că pentru acesta vom da o dimensiune pară; se afișează vectorii după sortare.

Rezolvare

Este necesar ca funcția de citire să aibă parametri întrucât prin intermediul ei preluăm datele despre doi vectori. Din modul de apel deducem că primul parametru va fi vectorul în care memorăm valorile citite iar al doilea va fi dimensiunea lui. Întrucât dimensiunea se preia în interiorul funcției și trebuie să fie vizibilă în afară (folosim apoi, după apel, `n` și `m`) trebuie ca acest parametru să fie referință. În cazul vectorului, chiar dacă îl modificăm în funcție – citindu-i elementele, nu trebuie scris vreun semn special la antet în momentul definirii întrucât, așa cum am descris la început, la funcție i se dă doar adresa vectorului – și prin intermediul ei se accesează originalul. Deci:

```
void citeste(int v[], int &n) {
    cin>>n;
    for (int i=1;i<=n;i++)
        cin>>v[i];
}
```

În cazul funcției de afișare avem de asemenea parametri vectorul și dimensiunea sa. Niciunul nu se modifică deci nu mai avem parametri referință. Cineva ar putea spune că am putea pune pe `n` și referință. Strict pe exemplul de mai sus ar merge, dar daca de pildă am dori să afișăm mai puține elemente și să invocăm la apel acest lucru printr-o constantă (spre exemplu `afiseaza(b,3);`) nu am mai trece de compilare.

```
void afiseaza(int v[], int n) {
    for (int i=1;i<=n;i++)
        cout<<v[i]<<" ";
    cout<<"\n";
}
```

Funcția de sortare este mai generală. Observăm că dorim să o folosim pentru ordonarea unei secvențe oarecare a vectorului, deci va avea trei parametri: vectorul și indicii între care se sortează.

```
void sorteaza(int v[], int st, int dr) {
    for (int i=st;i<dr;i++)
        for (int j=i+1;j<=dr;j++)
            if (v[i] > v[j]) {
                int aux = v[i];
                v[i] = v[j];
                v[j] = aux;
            }
}
```

Cele trei funcții de mai sus sunt procedurale. Din modul de apel am observat asta. Și nici nu s-ar justifica returnarea vreunei valori. În schimb, pentru funcția de calcul al maximului, care se apelează în cadrul instrucțiunii de afișate, nu avem decât alegerea: operand.

```
int maxim(int v[], int n) {
    int m = v[1];
    for (int i=2;i<=n;i++)
        if (v[i] > m)
            m = v[i];
    return m;
}
```

Funcții cu parametri matrice

Spre deosebire de vectori, în cazul matricelor suntem obligați să specificăm la definire, în antetul funcției, în mod exact a doua dimensiune.

Programul din următorul exemplu este corect și cere de la tastatură datele despre o matrice care poate avea oricâte linii dar maxim 10 coloane. Apoi este afișată această matrice pe ecran.

```
#include <iostream>
using namespace std;
int n, m, a[15][10];
void citire(int a[][10], int &n, int &m) {
    cin>>n>>m;
    for (int i=1;i<=n;i++)
        for (int j=1;j<=m;j++)
            cin>>a[i][j];
}
void afisare(int a[15][10], int &n, int &m) {
    for (int i=1;i<=n;i++) {
        for (int j=1;j<=m;j++)
            cout<<a[i][j]<<" ";
        cout<<"\n";
    }
}
int main () {
    citire(a, n, m);
    afisare(a, n, m);
    return 0;
}
```