

# POINTERI și REFERINȚE

## 1. Operatorul adresă (operatorul &)

Pentru a putea lucra cu un obiect (variabilă simplă, tablou, funcție, etc), programatorul trebuie să dispună de o modalitate de referire (de indicare) a respectivului obiect. Cea mai simplă modalitate este *referirea directă*, prin utilizarea numelui obiectului, adică a identificatorului asociat acestuia în momentul declarării lui. Dar nu orice obiect are un nume, de exemplu fiecare dintre cele 10 elemente ale tabloului **tab** declarat astfel

```
int tab[10];
```

este o variabilă anonimă de tip **int**. Pentru a accesa valorile unor astfel de variabile trebuie să folosim o modalitate de *referire indirectă*, cum ar fi, în acest caz, utilizarea operatorului de indexare [ ]:

```
tab[2]=13;
```

Modalitatea generală de referire indirectă a unui obiect constă în precizarea adresei sale de memorie prin intermediul unei variabile specializate, capabile să rețină o astfel de adresă. Orice obiect trebuie să ocupe, în momentul utilizării, o anumită zonă de memorie și, în consecință, orice obiect are o adresă de memorie. Zona de memorie ocupată de un obiect se numește *locația* obiectului și, prin definiție, adresa unui obiect este adresa primului octet al locației sale. Amintim că memoria este organizată la nivel de octet, octeții fiind numerotați începând de la numărul 1, adresele sunt memorate pe 32 biți (în cazul PC-urilor actuale) și, în consecință, *spațiul de adrese* este de la 1 la  $2^{32}-1$ .

Dacă dispunem deja de o cale de referire la un obiect (printr-o expresie l-valoare: nume, expresie cu indici, etc.), putem afla adresa acestuia folosind *operatorul adresă* (operatorul &), astfel:

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

int main() {
    int i;
    cout<<"adresa lui i    "<<&i<<endl;
    char c;
    cout<<"adresa lui c    "<<(&c)<<endl;

    struct Punct{ double x,y; } A;
    cout<<"\nOBS1 adresa unei structuri==adresa primului
membru:\n"<<endl;
    cout<<"adresa lui A    "<<&A<<endl;
    cout<<"adresa lui A.x "<<&A.x<<endl;
    cout<<"adresa lui A.y "<<&A.y<<endl;

    int tab[10];
    cout<<"\nOBS2 numele unui tablou==adresa tabloului:\n"<<endl;
    cout<<"adresa lui tab=  "<<&tab<<endl;
    cout<<"numele lui tab=   "<<tab<<endl;
    cout<<"adresa lui tab[0]="<<&tab[0]<<endl;
```

```

    cout<<"adresa lui tab[1]="<<&tab[1]<<endl;

    int suma(int, int);
    cout<<"\nOBS3 numele unei functii==adresa functiei:\n"<<endl;
    cout<<"adresa functiei suma="<<&suma<<endl;
    cout<<"numele functiei suma="<<suma<<endl;
    return 0;
}
int suma(int i, int j){
    return i+j;
}
/*
adresa lui i    0042FECC
adresa lui c    0042FEC3

OBS1 adresa unei structuri==adresa primului membru:

adresa lui A    0042FEA8
adresa lui A.x  0042FEA8
adresa lui A.y  0042FEB0

OBS2 numele unui tablou==adresa tabloului:

adresa lui tab=    0042FE78
numele lui tab=    0042FE78
adresa lui tab[0]=0042FE78
adresa lui tab[1]=0042FE7C

OBS3 numele unei functii==adresa functiei:

adresa functiei suma=00EC11F9
numele functiei suma=00EC11F9
Press any key to continue . . .
*/

```

Numele unui tablou este o constantă care are ca valoare adresa tabloului, care coincide cu adresa primului element al tabloului; analog, numele unei funcții este o constantă care are ca valoare adresa funcției, adică adresa primului octet al zonei de memorie ocupate de codul funcției. Adresa unei variabile simple sau a unei variabile de tip structură poate fi aflată numai cu ajutorul operatorului **&**.

Operatorul **&** aplicat unei l-valorii formează o expresie care, ca oricare altă expresie în C, are un anumit tip. Dacă variabila *x* are tipul **tip<sub>x</sub>**, atunci expresia **&x** are tipul "**pointer către tip<sub>x</sub>**". Mai precis, **&x** este o constantă de tip "**pointer către tip<sub>x</sub>**".

Un **pointer** este o variabilă care are ca valoare adresa unei locații de memorie. O variabilă pointer este o variabilă simplă, ea ocupă numai spațiul necesar pentru scrierea unei adrese, (patru octeți în cazul compilatorului MS Visual Studio actual). Deși toate variabilele pointer au valorile de același fel (adrese, adică numere în domeniul 1, 2, ...,  $2^{32}-1$ ), ele se deosebesc între ele prin tipul variabilelor la care fac referire (tipul țintei). Avem astfel "**pointer către int**", "**pointer către tablouri de 10 întregi**", etc.

De exemplu, dacă *x* este o variabilă de tip **int** și dorim să reținem undeva adresa sa, vom declara o variabilă *p* de tip "**pointer către int**" astfel

```
int *p;
```

și apoi îi atribuim valoarea pe care vrem să o reținem:

```
int main(void) {
    int x;
    int *p;
    p=&x;
    cout<< "x se afla la adresa " << p << endl;
    return 0;
}
/*
x se afla la adresa 0012FF60
Press any key to continue . . .*/
```

După cum se observă, o declarație de pointer declară de fapt tipul țintei pointerului, astfel încât compilatorul să poată organiza calculele cu data referită de acel pointer.

## 2. Operatorul țintă (operatorul \*)

O variabilă pointer, după cum am văzut, are ca valoare adresa unui obiect (în general anonim) de un tip bine precizat, numit *ținta* pointerului. Pentru a putea utiliza ținta unui pointer trebuie să facem o referire către ea. Calea către ținta pointerului *p* se obține prin aplicarea operatorului unar prefixat \*, expresia *\*p* desemnează obiectul a cărui adresă este conținută de pointerul *p*, atât ca r-valoare cât și ca l-valoare. Expresia *\*p* este de fapt numele țintei, poate să apară într-o expresie oriunde poate apare numele unui obiect de tipul țintei (atât în dreapta cât și în stânga unei atribuirii).

Exemplu:

```
int i;
int *p, *q;
p=&i;
i=13;
cout<<"i="<<i<<" *p="<<*p<<endl; //i=13 *p=13
*p=12;
cout<<"i="<<i<<" *p="<<*p<<endl; //i=12 *p=12
q=p;
*q=11;
cout<<"i="<<i<<" *p="<<*p<<endl; //i=11 *p=11
cout<<i+*p+*q<<endl; //33
```

În acest exemplu, pointerul *p* a fost inițializat cu adresa variabilei *i* și astfel ținta lui *p* este *i*, iar expresia *\*p* este sinonimă cu numele variabilei *i*.

Operatorul \*, "*ținta lui ...*", se aplică unui pointer și are ca rezultat o *referință* către ținta pointerului. Amintim că operatorul & se aplică unei referințe (unei l-valori) și are ca rezultat o constantă de tip "pointer către ..."

Operatorii adresă și țintă sunt unul inversul celuilalt (cu anumite precizări). Din acest motiv ei sunt numiți (oarecum impropriu) "operatorul de referențiere &" și "operatorul de dereferențiere \*".

Dacă  $x$  este o variabilă oarecare, atunci expresiile  $x$  și  $*\&x$  sunt sinonime (au aceeași r-valoare și aceeași l-valoare, ambele nu au efecte secundare).

```
int j, k;
k=12;
j=&k;
cout<<"j="<<j<<endl; //j=12
*&k=13;
cout<<"k="<<k<<endl; //k=13
```

De fapt, compilatorul “sterge” orice pereche  $*\&$  întâlnită:

```
004114CC rep stos     dword ptr es:[edi]
        int h;
        h=108;
004114CE mov         dword ptr [h],6Ch
        *&h=666;
004114D5 mov         dword ptr [h],29Ah
...
```

Reciproc, dacă  $p$  este un pointer, atunci  $p$  și  $\&*p$  au aceeași r-valoare. Totuși, deoarece rezultatul aplicării operatorului adresă nu are l-valoare, fiind o constantă de tip pointer, cele două expresii nu sunt complet echivalente:

```
int *p, *q;
int h=10;
p=&h;
q=&*p;
cout<<"p="<<p<<" q="<<q<<endl; // p=0012FF48 q=0012FF48
// *&p=q; // error C2106: '=' : left operand must be l-value
```

### 3. Chestiuni elementare despre pointeri

**a.) Declarații de pointeri.** În definirea unei variabile de tip pointer simbolul  $*$  poate fi interpretat ca fiind simbolul operatorului țintă. Mai precis, declarația

```
int *pi;
```

poate fi scrisă sub forma echivalentă

```
int* pi;
```

și citită “***pi*** este o variabilă de tip ***int\****”, adică un pointer către ***int***, sau poate fi scrisă, la fel de bine, sub forma

```
int *pi;
```

și citită așa: “ținta lui ***pi*** este o variabilă de tip ***int***”, adică ***pi*** este un pointer a cărei țintă este de tip ***int***. Când declarăm mai mulți pointeri de același tip se utilizează numai această ultimă interpretare: instrucțiunea

```
int *pi, *qi, *ti;
```

declară de fapt că țintele celor trei pointeri au tipul ***int***. Tipul abstract ***int\****, util în multe alte situații, aici poate provoca confuzii: instrucțiunea

```
int* p1, p2, p3;
```

nu declară trei variabile de tip ***int\****, așa cu ar fi de așteptat, ci numai una, ***p1***, celelalte două fiind de tip ***int***. Compilatorul o “înțelege” sub forma

```
int (*p1), p2, p3;
```

Declararea unui pointer către un tip compus (tablou, funcție) se face după aceeași regulă ca în cazul pointerilor către tipurile simple: declarăm de fapt tipul țintei, dar acum mai trebuie să ținem cont și de prioritatea operatorilor implicați - amintim că operatorul unar `*` leagă mai slab decât operatorii postfixați `()` și `[]`.

De exemplu, dacă dorim să definim un pointer *p* către un tablou de 10 caractere, regula practică este următoarea: definim o variabilă *x* de tipul țintei:

```
char x[10];
```

după care înlocuim numele *x* cu ținta lui *p*, adică *\*p*, scrisă între paranteze rotunde:

```
char (*p)[10];
```

Uneori parantezele rotunde nu sunt necesare, dar în cazul de mai sus ele nu pot fi omise, deoarece declarația

```
char *p[10];
```

nu declară un pointer ci un tablou de 10 elemente de tip *char\**, adică un tablou de 10 pointeri către *char*. Explicație: în declarația de mai sus asupra operandului *p* acționează doi operatori: `*` și `[]`, cum operatorul de indexare `[]` leagă mai tare decât operatorul țintă `*` compilatorul citește această ultimă declarație sub forma:

```
char *(p[10]);
```

de unde deducem că *p* este un tablou cu 10 elemente, fiecare element *e* al său fiind dat de declarația pe care o obținem înlocuind *p[10]* cu *e*

```
char *(e);
```

care este echivalentă cu

```
char *e;
```

și care declară pe *e* ca pointer către *char*.

Alt exemplu: cu instrucțiunea

```
double *f(int, int);
```

declarăm o funcție *f* cu două argumente *int* și care întoarce un rezultat de tip *double\**, adică un pointer către *double*, iar cu instrucțiunea

```
double (*pf)(int, int);
```

declarăm un pointer *pf* a cărui țintă este dată de declarația

```
double tinta(int, int);
```

adică este o funcție cu două argumente *int* și un rezultat de tip *double*.

În exemplele de mai sus observăm utilitatea *declaratorilor abstracti de tip* - secvențe de cod care descriu tipurile compuse, cum ar fi *int\** (pentru tipul “pointer către *int*”), *double\**, etc. Declaratorul abstract al unui anumit tip se obține ștergând pur și simplu numele variabilei din declarația unei variabile de acel tip. Exemple:

- din *int x* obținem *int* – tipul *int*;
- din *double mat[10]* obținem *double[10]* – tablou de 10 elemente de tip *double*;
- din *int \*p* obținem *int \**, adică “*pointer către int*”
- din *char (\*ptab)[5]* obținem *char (\*)[5]*, adică “*pointer către tablouri de tip char [5]*”
- din *void (\*p)(int, int)* rezultă *void (\*)(int, int)*, adică “*pointer către funcții de tip void (int, int)*”

Deoarece declarațiile de pointeri către tipuri compuse pot deveni foarte complicate, este recomandată utilizarea lui *typedef* pentru redenumirea tipului țintă. Comparați cele două modalități echivalente de declarare a pointerului *p* din programul următor:

```

#include<iostream>
using namespace std;
void unu () {
    char (*p) [10];
    char text[10]="mesaj";
    p=&text;
    cout<<*p<<endl;
}
void doi () {
    typedef char Text[10];
    Text *p;
    Text text="mesaj";
    p=&text;
    cout<<*p<<endl;
}
int main(void) {
    unu(); //mesaj
    doi(); //mesaj
    return 0;
}

```

Să definim și pointeri către funcții:

```

#include<iostream>
using namespace std;
typedef int Functie(int, int);

int suma(int i, int j){
    return i+j;
}

int main(void) {
    Functie *pf;
    pf=suma;
    cout<<(*pf) (2,3)<<endl;
    return 0;
}

```

Iată și o altă variantă a programului de mai sus:

```

#include<iostream>
using namespace std;
typedef int Functie(int, int);
typedef Functie *PFunctie;

int suma(int i, int j){
    return i+j;
}

int main(void) {
    PFuncție pf;
    pf=suma;
    cout<<(*pf) (2,3)<<endl;
    return 0;
}

```

**b.) Pointeri către structuri.** Orice tip de dată poate fi ținta unui pointer, în particular datele de tip structură:

```
#include<iostream>
using namespace std;
struct Punct { double x,y,z;};
int main(void) {
    Punct A={1,20,300};
    Punct *p;
    p=&A;
    //cout<<*p.x<<endl;
    //error C2228: left of '.x' must have class/struct/union
    //did you intend to use '->' instead?
    cout<<(*p).x<<endl;    //1
    cout<<p->x<<endl;    //1
    return 0;
}
```

Observăm că la accesarea indirectă (adică printr-un pointer) a membrilor unei structuri trebuie să ținem cont de prioritatea operatorilor: operatorul țintă leagă mai slab decât operatorul punct (operatorul de selecție membru). Expresia *\*p.x* este înțeleasă de compilator ca *\*(p.x)* și este eronată (*p* nu este o structură), referirea la membrul *x* al țintei lui *p* trebuie scrisă cu paranteze *(\*p).x*, după cum se vede în codul de mai sus.

Deoarece accesarea indirectă a structurilor (și a claselor) este foarte frecventă în programare, fiind singura lor modalitate de accesare în cazul alocării dinamice, limbajul a fost prevăzut cu *operatorul de accesare indirectă*, “operatorul săgeată”, definit astfel: dacă *p* este un pointer către o structură care are un membru *x*, atunci expresia *p->x* este sintactic echivalentă cu *(\*p).x* (adică la compilare prima este tradusă în a doua). Utilizarea operatorului săgeată simplifică foarte mult scrierea și mărește lizibilitatea codului.

**c.) Adresa unui pointer.** Orice variabilă pointer are o locație în memorie (de patru octeți în MS Visual C++ 9.0) și deci are o adresă. Adresa unui pointer nu trebuie confundată cu adresa conținută de acel pointer (care este valoarea pointerului, adică adresa țintei sale) :

```
#include<iostream>
using namespace std;
int main(void) {
    int k=12;
    int *pi;
    pi=&k;
    cout<<"adresa lui pi, &pi="<<&pi<<endl;
    cout<<"valoarea lui pi, pi="<<pi<<" (adresa tinte)"<<endl;
    cout<<"valoarea tinte, *pi="<<*pi<<endl;
    return 0;
}
/* REZULTAT
adresa lui pi, &pi=0012FF54
valoarea lui pi, pi=0012FF60 (adresa tinte)
valoarea tinte, *pi=12
Press any key to continue . . . */
```

În exemplul de mai sus, expresia *&pi* este o constantă de tip pointer către *pi*, deci de tip “pointer către tipul lui *pi*”, cum tipul lui *pi* este *int\** obținem că *&pi* este o constantă de

tip “*pointer către int\**”, adică de tip *int\*\** (pointer către pointer către *int*). O variabilă pointer *pp* căreia să i se poată atribui valoarea constantei *&pi* trebuie declarată astfel:

```
int **pp;
```

Această declarație poate fi scrisă și sub forma

```
int *(*pp);
```

adică “ținta țintei lui *pp* este un *int*”. Următoarea secvență de cod este echivalentă cu precedenta:

```
int k=12;
int *pi=&k;
int **pp;
pp=&pi;
cout<<"adresa lui pi, &pi="<<pp<<endl;
cout<<"valoarea lui pi, pi="<<*pp<<" (adresa țintei)"<<endl;
cout<<"valoarea țintei, *pi="<<*pp<<endl;
```

**d.) Adresa adresei.** Operatorul *&* nu poate fi aplicat în mod succesiv: *&x* este o constantă (entitate fără locație de memorie) iar *&&x*, adresa unei constante, nu are sens. În schimb, operatorul de dereferențiere *\** poate fi aplicat succesiv, atât timp cât rezultatul fiecărei dereferențieri este tot de tip pointer (vezi exemplul de mai sus).

**e.) Conversii între pointeri.** Tipul pointerului trebuie respectat cu strictețe. Deși toți pointerii au valorile de același fel (adrese), nu sunt permise atribuirii între pointeri de tip diferit. Exemplu:

```
int k;
double *g;
g=&k; // error C2440: cannot convert from 'int *' to 'double *'
```

Programatorul poate forța, pe propria răspundere, astfel de atribuirii prin conversii explicite utilizând operatorul *cast*:

```
#include<iostream>
using namespace std;
#include<iostream>
using namespace std;
int main() {
    int k=12;
    double *pp;
    pp = (double *)&k;
    cout << "*pp=" << *pp << endl;
    cout << "(int)*pp=" << (int)*pp << endl;
    cout << "**(int*)pp=" << *(int*)pp << endl;
    return 0;
}
/* REZULTAT
*pp = -9.25596e+061
(int)*pp = -2147483648
*(int*)pp = 12
Press any key to continue . . .*/
```

Observăm că pointerul *pp*, de tip *double\**, a fost încărcat (printr-o expresie *cast*) cu adresa unui întreg. Pentru a regăsi în mod corect valoarea țintei, trebuie să folosim iarăși o conversie explicită: expresia *(int\*)pp* are înțelesul: deși *pp* este de tip *double\**, acum el este utilizat ca și cum ar fi de tip *int\**. Aceste conversii sunt necesare pentru că tipul țintei



precizează formatul intern al datei țintite, iar formatul intern al unei date de tip *int* este complet diferit de al uneia de tip *double*.

Un caz special îl constituie pointerii de tip *void\**. Un pointer *pv* de tip *void\** poate conține, prin definiție, adrese către date de orice tip, în consecință, la o atribuire către *pv* nu mai sunt necesare conversii explicite, în schimb ținta sa, *\*pv*, are tipul nedeterminat și trebuie precizat prin expresii *cast*:

```
void *pv;
int n = 108;
pv = &n; //ok
cout << *(int*)pv << endl; //108
```

**f.) Conversii între pointeri și întregi.** Adresele (valorile pointerilor), deși sunt exprimate prin numere întregi (obținute în urma numerotării octeților de memorie), nu sunt date de tip întreg. În consecință, nu sunt permise atribuiri între pointeri și tipuri aritmetice. Sunt permise (dar ne-uzuale) doar conversii explicite între pointeri și tipul *int*.

```
int tab[2] = { 100, 200 };
int a, *p;
p = &tab[0];
a = (int)p;
p = (int*)(a + 4);
cout << *p << endl; // 200
```

După cum vom vedea, limbajul a fost prevăzut cu operații speciale asupra pointerilor, așa numita aritmetică a pointerilor, care fac inutile conversiile exemplificate mai sus. Este permisă, fără conversie explicită, numai atribuirea

```
double *p;
p=0;
```

deoarece zero are o semnificație specială în acest context (vezi mai jos).

**g.) Inițializarea pointerilor.** Ca orice altă variabilă, un pointer trebuie definit și inițializat înainte de a fi folosit. Utilizarea unui pointer definit dar neinițializat conduce la erori grave de execuție:

```
#include<iostream>
using namespace std;
int main(void){
    int *pn;
    cout<<"adresa lui pn:"<<&pn<<endl;
    //adresa lui pn:0012FF60
    cout<<"valoarea lui pn:"<<pn<<endl;
    //valoarea lui pn:CCCCCCCC
    cout<<"valoarea tinteii lui pn:"<<*pn<<endl;
    //exemplu.exe has stopped working!
    return 0;
}
```

În exemplul de mai sus încercăm să citim un *int* aflat la adresa CCCCCCCCCh, care se dovedește a fi într-o zonă protejată memorie, fapt sancționat de sistemul de operare prin întreruperea executării programului.

Nu trebuie confundată inițializarea unui pointer cu atribuirea unei valori țintei sale; următorul program trece de compilare dar la rulare este și el scos din execuție:

```
#include<iostream>
using namespace std;
int main(void) {
    int a=5;
    int *p;
    *p=a; // exemplu.exe has stopped working
    cout<<*p<<endl;
    return 0;
}
```

Varianța corectă este următoarea:

```
#include<iostream>
using namespace std;
int main(void) {
    int a=5;
    int *p;
    p=&a;
    cout<<*p<<endl; //5
    return 0;
}
```

Unui pointer *i* se poate atribui în mod corect o valoare inițială numai în următoarele trei moduri:

1. *prin atribuirea adresei unui obiect deja definit*, așa cum am văzut mai sus:

```
int a=5;
int *p;
p=&a;
```

2. *prin atribuirea valorii unui pointer deja inițializat*:

```
int a=5;
int *p, *q;
p=&a;
q=p;
```

Un caz particular îl constituie inițializarea cu pointerul nul:

```
double *p;
p=NULL;
```

Pointerul **NULL** este de fapt o constantă simbolică definită în `<stdio.h>` dată, în funcție de compilator, prin

```
#define NULL ((void *)0)
```

sau prin

```
#define NULL 0
```

astfel încât  $p=NULL$  atribuie pointerului *p* valoarea zero, fapt care semnalează că pointerul *p* conține o adresă invalidă (standardul limbajului C/C++ garantează că zero nu este adresa nici unei locații de memorie). Biblioteca `<stdio.h>` este inclusă în `<iostream>` și nu trebuie cerută în mod explicit printr-o directivă `include`. În C++ putem folosi (și chiar este recomandat) direct atribuirea  $p = 0$ . Atenție, nu avem

voie să citim date de la adresa zero, pentru că ea nu există; prin urmare și acest program este scos din execuție:

```
#include<iostream>
using namespace std;
int main(void) {
    int *p;
    p=0;
    cout<<*p<<endl; // exemplu.exe has stopped working
    return 0;
}
```

Pointerul nul este util în multe situații, de exemplu o funcție care returnează un pointer poate semnaliza că rezultatul este invalid returnând pointerul nul. Este clar că într-o astfel de abordare funcția care primește pointerul trebuie să testeze mai întâi validitatea acestuia. Coșmarul programatorilor are un nume: *null pointer exception*.

3. *prin atribuirea adresei unei locații de memorie din zona alocată programului*, adresă obținută prin utilizarea operatorului **new** (specific C++) sau a funcțiilor de alocare de memorie **malloc**, **calloc**, etc (specific C). Asupra acestei modalități vom reveni pe larg mai târziu.

O modalitate legală, dar cu efecte imprevizibile, de inițializare a unui pointer constă în atribuirea unei valori întregi printr-o expresie *cast*:

```
char *pc;
pc=(char *)0x0012ff60;
```

O astfel de inițializare este complet hazardată și fără șanse de reușită, programatorul nu poate ști a priori care va fi zona de memorie alocată programului la rulare.

## 4. Aritmetica pointerilor

Operațiile aritmetice cu pointeri au fost concepute în principal pentru parcurgerea tablourilor, mai precis, pentru accesarea datelor stocate în zone de memorie organizate ca locații succesive de aceeași mărime. Să reamintim că mărimea locației de memorie a unei variabile sau a unui tip poate fi aflată cu operatorul **sizeof**. Unitatea de măsură este mărimea tipului **char**, adică un octet în cazul nostru. Avem următoarele reguli de calcul:

- mărimea unui tip simplu = **sizeof**(tip);
- mărimea unui tablou = nr.elemente\***sizeof**(tip\_element);
- mărimea unei funcții = nu se definește (nu sunt permise tablouri de funcții, chiar dacă două funcții au același tip, codurile lor pot avea lungimi diferite);
- mărimea unui pointer către o funcție = mărimea oricărui pointer = 4 octeți.

Verificare:

```
#include<iostream>
using namespace std;
void f(int i){
    cout<<"in f avem i="<<i<<endl;
    return;
}
int main(void){
    double a=12;
    double* p=&a;
```

```

cout<<sizeof(p)<<endl;          //4
cout<<sizeof(*p)<<endl;         //8

cout<<sizeof(double[10])<<endl; //80
cout<<sizeof(double *[10])<<endl; //40
cout<<sizeof(double (*)[10])<<endl; //4

void (*pf)(int);                //pointer către functii de tip void(int)
pf=&f;                          //    <=>   pf=f;
(*pf)(13);                      //    "in f avem i=13 "
cout<<sizeof(pf)<<endl; //      4
//cout<<sizeof(*pf)<<endl; // error : illegal sizeof operand
//cout<<sizeof(f)<<endl; //error: illegal sizeof operand
//pf++; //error : illegal on operands of type 'void (*) (int)'
//(pf[0])(13); //error: subscript requires array or pointer type
//! nu exista tablouri de functii !
return 0;
}

```

Operațiile aritmetice sunt permise numai pentru pointeri care au mărimea țintei bine definită (și deci ținta poate fi element al unui tablou). Sunt definite numai următoarele operații:

**a) Incrementare/decrementare.** Dacă  $p$  este o *variabilă pointer* pentru care este definită mărimea țintei  $*p$ , atunci sunt permise operațiile  $p++$ ,  $p--$ ,  $++p$  și  $--p$ , care au aceeași interpretare ca în cazul variabilelor aritmetice, cu singura deosebire că pasul incrementării este egal cu mărimea țintei:

```

int a=12;
int* p=&a;
cout<<"p="<<p<<endl; //p=0012FF60
p++;
cout<<"p="<<p<<endl; //p=0012FF64
double *pp=NULL;
cout<<"pp="<<pp<<endl; //pp=00000000
pp++;
cout<<"pp="<<pp<<endl; //pp=00000008

```

Observăm că incrementarea se face cu pasul  $sizeof(*p)$ .

**b) Suma și diferența dintre un pointer și un întreg.** Dacă  $p$  este un pointer iar  $i$  este un întreg, expresiile  $p+i$  și  $i+p$  au ca rezultat valoarea lui  $p$  mărită cu  $i*sizeof(*p)$ , iar diferența  $p-i$  are ca rezultat valoarea lui  $p$  micșorată cu  $i*sizeof(*p)$ .

```

int i=2;
int *pp=&i;
cout<<"pp="<<pp<<endl; //pp =0033FDEC
cout<<"pp+i="<<pp+i<<endl; //pp+i=0033FDF4
cout<<"pp-1="<<pp-1<<endl; //pp-1=0033FDE8
//cout<<"i-pp="<<i-pp<<endl;
//error: pointer can only be subtracted from another pointer
pp+=5;
cout<<"pp="<<pp<<endl; //pp =0033FE00

```

**c) Diferența a doi pointeri.** Este permisă scăderea a doi pointeri de același tip, rezultatul este de tip **int**, iar pasul operației este egal cu mărimea tipului țintă.

```
int a,b, *p=&a,*q=&b;
cout<<"p="<<p<<endl;           //p=0012FF60
cout<<"q="<<q<<endl;           //q=0012FF54
cout<<"p-q="<<p-q<<endl;       //p-q=3
```

**d) Comparația a doi pointeri.** Doi pointeri de același tip pot fi comparați cu operatorii <, <=, ==, >= și >. Un caz particular îl reprezintă comparația cu zero, care este permisă, zero fiind asimilat cu pointerul nul.

```
int a,b, *p=&a,*q=&b;
double bb, *pp=&bb;
if (p<=q) cout<<"p<=q"<<endl;    //p>q
else cout<<"p>q"<<endl;
p=NULL;
if (p==0) cout<<"p==NULL"<<endl; //p==NULL
//if (p<pp) cout<<"???"<<endl; //error: no conversion
//from 'double *' to 'int *'
```

## 5. Tablouri și pointeri. Operatorul de indexare [ ]

După cum am mai spus, aritmetica pointerilor a fost special concepută pentru lucrul cu tablouri. Să urmărim exemplul de mai jos în care inițializăm un pointer cu adresa unui element dintr-un tablou:

```
int tab[5]={0,10,20,30,40};
int *p;
p=&tab[2];
cout<<*p<<endl;           //20
p++;
cout<<*p<<endl;           //30
```

Observăm că incrementarea pointerului mută ținta acestuia la următorul element al tabloului. Analog, în cazul adunării unui întreg la un pointer:

```
int tab[5]={0,10,20,30,40};
int *p;
p=&tab[0];
cout<<*p<<endl;           //0
cout<<*(p+1)<<endl;        //10
cout<<*(p+2)<<endl;        //20
```

Din modul de definire a operațiilor aritmetice cu pointeri deducem imediat că din atribuirea `p=&tab[0]` rezultă egalitatea `*(p+i)==tab[i]`. După cum știm deja, numele unui tablou este o constantă de tip pointer care țintește către primul element al tabloului, adică `tab <=> &tab[0]`. Deci, dacă `p==tab` atunci `*(p+i)==tab[i]`.

Această egalitate este mult mai profundă, ea servește de fapt la definirea *operatorului de indexare* [ ]: dacă  $p$  este un pointer iar  $i$  este un întreg, atunci expresia  $p[i]$  înseamnă, prin definiție,  $*(p+i)$ , altfel spus, la compilare expresia  $p[i]$  este înlocuită cu expresia  $*(p+i)$ . În același context, expresia  $i[p]$  este înlocuită cu  $*(i+p)$ , și prin urmare  $i[p]$  și  $p[i]$  sunt expresii echivalente.

Subliniem că și în cazul tablourilor se aplică definiția de mai sus, dacă  $tab$  este un tablou iar  $i$  un întreg, expresia  $tab[i]$  este înțeleasă de compilator ca  $*(tab+i)$ , identificatorul  $tab$  desemnând aici o constantă de tip pointer. Verificare:

```
int main() {
    char text[]="Clar!";
    cout<<text[4]<<endl;    ///
    cout<<*(text+4)<<endl;  ///
    cout<<*(4+text)<<endl;  ///
    cout<<4[text]<<endl;    ///
    cout<<(int*)"Clar!"<<endl;    //011D7800
    cout<<"Clar!"[4]<<endl;    ///
    cout<<4["Clar!"]<<endl;    ///
    return 0;
}
```

Ultimele trei rezultate au următoarea explicație: orice constantă de tip string este chiar identificatorul tabloului de caractere care conține stringul respectiv, tablou alocat la compilare într-o zonă specială de memorie; prin urmare în expresii o constantă de tip string desemnează adresa “fizică” a primului său caracter.

### Exemplul 1. Parcurgeri de tablouri . Cazul unu-dimensional:

```
#include<iostream>
using namespace std;
int main() {
    const int dim=10;
    int tab[dim];
    for(int i=0; i<dim; i++) tab[i]=i*i;
    for(int i=0; i<dim; i++) cout<<*(tab+i)<<" ";
    cout<<endl;    //0 1 4 9 16 25 36 49 64 81

    int* const p=tab; //<=> p=&tab[0]    // => *p=tab[0];
    for(int i=0;i<dim;i++) cout<<*(p+i)<<" ";
    cout<<endl;    //0 1 4 9 16 25 36 49 64 81

    for(int i=0;i<dim;i++) cout<<p[i]<<" ";
    cout<<endl;    //0 1 4 9 16 25 36 49 64 81

    // tab++; //      error: '++' needs l-value
    // (tab este o constanta de tip "pointer catre int")

    // p++;// error C3892: 'p' : you cannot assign to a variable
    // that is const (p a fost declarat pointer constant)

    int* pp=tab;
    for(int i=0;i<dim;i++) cout<<*pp++<<" ";
    cout<<endl;    //0 1 4 9 16 25 36 49 64 81

    for(int *q=tab,*qfin=tab+dim; q<qfin; q++) cout<<*q<<" ";
    cout<<endl;    //0 1 4 9 16 25 36 49 64 81
}
```

```

    return 0;
}

```

Dintre toate variantele de parcurgere a unui tablou prezentate mai sus, ultima este cea mai rapidă.

Revenim asupra dublei semnificații a identificatorilor de tablouri. În declarația  

```
int tab[5]={0, 1, 4, 9, 25};
```

identificatorul **tab** desemnează numele unei date de tip **int[5]**, la fel ca în expresia

```
cout<<&tab<<endl;
```

Pe de altă parte, în expresia

```
cout<<tab<<endl;
```

care produce exact același rezultat pe monitor, identificatorul **tab** este o constantă de tip pointer către **int**, deci de tip **int\***, și care are ca valoare adresa primului element al tabloului. Adresa unui tablou (obținută cu operatorul **&**) este și ea o constantă de tip pointer, dar având tipul pointer către tablouri și nu tipul pointer către elementele tabloului. În cazul nostru, **tab** desemnează adresa unui **int**, deci este de tip **int\***, iar **&tab** este adresa unui tablou de tip **int[5]**, deci **&tab** este de tip **int(\*)[5]**:

```

#include<iostream>
using namespace std;
int main(void){
    int tab[5]={0, 11, 22, 33, 44};
    int *p;
    p=tab;
    cout<<p<<endl;           //0043FB78
    cout<<p[2]<<endl;         //22
    cout<<*(p+2)<<endl;       //22
    //p=&tab; error: cannot convert from 'int (*)[5]' to 'int *'
    int (*ptab)[5];
    ptab=&tab;                //deci tab==*ptab==ptab[0]
    cout<<ptab<<endl;         //0043FB78
    cout<<(*ptab)[2]<<endl;    //22
    cout<<*(ptab+2)<<endl;    //22
    cout<<ptab[0][2]<<endl;    //22
    return 0;
}

```

Reținem: numele unui tablou este o constantă de tip pointer către primul element al tabloului, nu către tablou, în exemplul de mai sus expresia **tab** este echivalentă cu **&tab[0]** și nu cu **&tab**, chiar dacă cele două adrese au aceeași valoare numerică, una este adresa unui **int** iar cealaltă este adresa unui tablou de tip **int[5]**.

Amintim că o matrice bidimensională este un tablou de tablouri. Mai precis, o declarație de forma

```
int mat[3][5];
```

declară **mat** ca fiind un tablou cu 3 elemente, **mat[0]**, **mat[1]** și **mat[2]**, fiecare element fiind un tablou de tip **int[5]**.

Această interpretare o obținem din declarația de mai sus plecând de la identificatorul **mat** și observând că asupra lui acționează un singur operator, și anume operatorul **[3]**:

```
int (mat[3])[5];
```

Rezultă că *mat* este un tablou cu 3 elemente, pentru a afla tipul acestora substituim *mat[3]* cu *Element* și obținem:

```
int Element[5]
```

de unde rezultă că *Element* este un tablou cu 5 elemente, pentru a afla tipul lor substituim mai departe *Element[5]* cu *Elementel* și obținem în final

```
int Elementel;
```

adică *Elementel* este un întreg de tip *int*. Deducem că *mat* este un tablou de 3 tablouri de câte 5 *int*.

Să analizăm acum legătura dintre matricea *mat* și pointeri. Deoarece fiecare *mat[i]* este un tablou de tip *int[5]* rezultă identitatea *mat[i]==&mat[i][0]* iar valoarea lui *mat[i]* poate fi atribuită unui pointer de tip *int\**:

```
int *p;
p=mat[i];
```

cu ajutorul căruia putem parcurge apoi toată linia *i* a matricei.

Pe de altă parte, *mat* este un tablou cu elemente de tip *int[5]*, deci *mat==&mat[0]* de unde urmează că valoarea lui *mat* poate fi atribuită unui pointer către *int[5]*

```
int (*pLin)[5];
pLin=mat;
```

pointer care prin incrementare sare de la o linie la alta.

## Exemplul 2. Parcurgeri de tablouri. Cazul bi-dimensional:

```
#include<iostream>
using namespace std;
const int nrMaxLinii=10, dimMaxLinie=10;
typedef int Linie[dimMaxLinie];
int main() {
    int nrLinii=3, nrColoane=5;
    int mat[nrMaxLinii][dimMaxLinie] = {{-1, -2, -3, -4, -5},
                                         {11, 22, 33, 44, 55},
                                         {10, 20, 30, 40, 50}};

    cout<<"Parcurgere cu doi indici \n"<<endl;
    for(int i=0; i<nrLinii; i++) {
        for(int j=0; j<nrColoane; j++) {
            cout<<mat[i][j]<<' ';          // <=>
            //cout<<*(mat[i]+j)<<' ';      // <=>
            //cout<<*(mat+i)[j]<<' ';      // <=>
            //cout<<*(mat+i+j)<<' ';
        }
        cout<<endl; }
    cout<<endl;
    cout<<"Parcurgerea liniilor cu un pointer\n"<<endl;
    for(int i=0; i<nrLinii; i++) {
        int *p=mat[i]; // => *p==mat[i][0]==inceputul liniei;
        for(int* pFinal=p+nrColoane; p<pFinal; p++)
            cout<<*p<<' ';
        cout<<endl;
    }
    cout<<endl;

    cout<<"Selectarea liniilor cu un pointer catre linii\n"<<endl;
    Linie *pLin;      // <=> int (*pLin)[dimMaxLinie];
    pLin=mat;          // => *pLin==mat[0] == prima linie;
```



```

for(Linie* pFinal=mat+nrLinii; pLin<pFinal; pLin++){
    for(int j=0;j<nrColoane;j++) cout<<(*pLin)[j]<<' ';
    //for(int* p=*pLin;p<*pLin+nrColoane;p++) cout<<*p<<' ';
    cout<<endl;
}
cout<<endl;

cout<<"Parcurgere dublu indirectata\n"<<endl;
int* colP[nrMaxLinii];
//coloana pointerilor cap de linie
for(int i=0;i<nrLinii;i++)
    colP[i]=mat[i]; // => colP[i][j]==mat[i][j];
int **m;
m=colP; // => m[i]==colP[i]
for(int i=0;i<nrLinii;i++){
    for(int j=0;j<nrColoane;j++)
        cout<<*(m+i+j)<<' '; // <=> cout<<m[i][j]<<' ';
    cout<<endl;
}
cout<<endl;
return 0;
}

```

## 6. Tablouri ca parametri formali

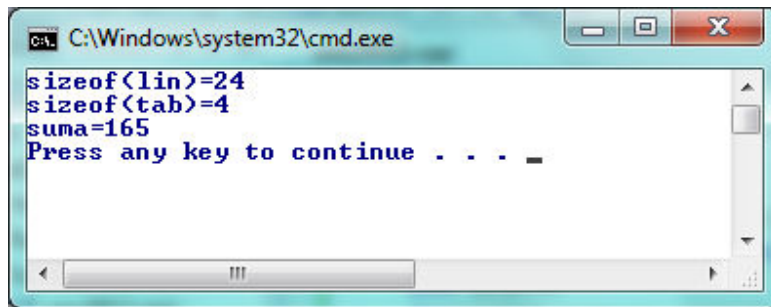
Să examinăm cu atenție următorul program în care calculăm cu ajutorul unei funcții suma primelor  $n$  elemente ale unui vector:

```

#include<iostream>
using namespace std;
int suma(int tab[6], int n){
    cout<<"sizeof(tab)="<<sizeof(tab)<<endl;
    int i,s=0;
    for(i=0;i<n;i++) s+=tab[i];
    return s;
}
int main(void){
    int lin[6]={11,22,33,44,55,66};
    cout<<"sizeof(lin)="<<sizeof(lin)<<endl;
    cout<<"suma="<<suma(lin,5)<<endl;
    //<--> cout<<"suma="<<suma(&lin[0],5)<<endl;
    return 0;
}

```

Rezultatul este următorul:



Observăm că în funcția *suma* parametrul formal *tab* a fost declarat de tip *int*[6], dar *sizeof(tab)* are valoarea 4 și nu 24 cum ar fi de așteptat. Explicația este următoarea: compilatorul înlocuiește orice declarație de tablou întâlnită în lisa parametrilor formali ai unei funcții cu declarația unui pointer către tipul elementelor tabloului. Prin urmare, identificatorul *tab* din exemplul de mai sus este o variabilă de tip pointer către *int* alocată pe stivă în momentul apelului (deci o variabilă locală funcției *suma*), iar apelul *suma(lin, 5)* atribuie lui *tab* valoarea constantei *lin*, adică adresa lui *lin*[0]. Cum orice pointer este memorat pe 4 octeți, avem *sizeof(tab)*==4. Declarațiile următoare sunt tratate de compilator în același fel:

```
int suma(int tab[6], int n){ ... }
int suma(int tab[], int n){ ... }
int suma(int *tab, int n){ ... }
```

iar dintre ele ultima este de preferat.

În concluzie, tablourile se transmit în funcții prin pointeri, funcția care face apelul are responsabilitatea alocării tabloului prelucrat, pentru prelucrare ea îi trimite funcției apelate adresa primului element și numărul de elemente care vor fi prelucrate, iar funcția apelată primește adresa primului element într-o variabilă locală de tip pointer.

**Exemplul 3.** Cazul unidimensional: un vector *tab* cu *dim* elemente de tip *double* este un tablou cu *dim* elemente de tip *double*, în consecință, în funcția apelată vom trimite adresa primului element (ca valoare a unui pointer de tip *double\**).

```
#include<iostream>
using namespace std;
double suma(double *tab, int dim){
//<=>double suma(double tab[ ], int dim){
//<=>double suma(double tab[5], int dim){
    double suma=0;
    for(double* tabFinal=tab+dim; tab<tabFinal; tab++) suma+=*tab;
    return suma;
}
int main(void){
    double tab[5]={10000,2000,300,40,5};
    cout<<suma(tab,5)<<endl;        //12345
    cout<<suma(tab,3)<<endl;        //12300
    cout<<suma(tab+2,3)<<endl;      //345
    //tab++; // error: '++' needs l-value
    return 0;
}
```

**Exemplul 4.** Cazul bidimensional: o matrice *mat* cu *m* linii și *n* coloane este un tablou cu *m* elemente de tip tablouri de dimensiune *n*, prin urmare adresa primului element al lui *mat* este o constantă de tip “pointer către tablouri de dimensiune *n*” și deci de acest tip trebuie declarat parametrul formal capabil să primească matricea *mat* la apelare:

```
#include<iostream>
using namespace std;
const int nrMaxLinii=30, dimMaxLinie=6;
typedef int Linie[dimMaxLinie];

int suma(Linie *tab, int nrLin, int nrCol){
//<=>int suma(int tab[ ][dimMaxLinie], int nrLin, int nrCol){
//<=>int suma(int tab[nrMaxLinii][dimMaxLinie], int nrLin, int nrCol){
    int s=0;
    for(Linie* ftab=tab+nrLin; ftab<tab+nrLin; ftab++)/*ftab==linia curenta
        for(int j=0; j<nrCol; j++)
            s+=(*ftab)[j];
    return s;
}
int main(void){
    int mat[nrMaxLinii][dimMaxLinie]={{11,22,33,44},
                                        {1,2,3,4},
                                        {10,20,30,40}};
    cout<<"suma="<<suma(mat,2,4)<<endl;
    return 0;
}
```

În *suma* declarăm că *tab* este un pointer către linii de 6 întregi, în *main* alocăm static tabloul *mat* de 30 astfel de linii iar la apelare îi furnizăm funcției *suma* adresa primei linii și dimensiunile zonei din colțul de dreapta-sus a matricei *mat* care va fi prelucrată.

Evident că zona sumată poate fi parcursă și cu doi indecși:

```
int suma(Linie *tab, int nrLin, int nrCol){
    int s=0;
    for(int i=0; i<nrLin; i++)
        for(int j=0; j<nrCol; j++)
            s+=tab[i][j];
    return s;
}
```

În final prezentăm un exemplu de prelucrare de stringuri. Se cere să se implementeze funcția cu antetul

```
void elimina(char *s, char ch)
```

care elimină din stringul *s* orice apariție a caracterului *ch*. Deoarece nu se precizează o dimensiune maximă a stringurilor prelucrate nu putem alocă static în funcția *elimina()* un tablou de sprijin (un buffer) în care, eventual, să mutăm provizoriu numai caracterele valide, altfel spus trebuie să “lucram pe loc” în tabloul primit.

O posibilă rezolvare este următoarea:

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

```

void elimina(char *s, char ch){
    char cc,*p=s;
    do
        if(ch!=(cc=*s++)) *p++=cc;
    while(cc!='\0');
}

int main(void){
    char text[100]="bbAMb EbLbIbMIbNAbbT bBb bMbICb";
    elimina(text, 'b');
    cout<<'>'<<text<<'<'<<endl;
    return 0;
}
/*REZULTAT:
>AM ELIMINAT B MIC<
Press any key to continue . . .*/

```

În subrutina *elimina()* stringul primit la apel este parcurs cu doi pointeri: pointerul *s* care țintește caracterul curent *cc* și pointerul *p* care țintește către locul unde va fi mutat *cc* dacă acesta este diferit de *ch*. Pointerul *s* este incrementat totdeauna, iar *p* numai dacă *cc* a fost mutat. Astfel *p* rămâne în urma lui *s*, mutările au loc numai în față și se poate lucra pe loc în tabloul care conține stringul prelucrat.

În final câteva precizări privind sintaxa și utilizarea modifierului *const* (specific C++). Dacă dorim să înștiințăm compilatorul că valoarea unei variabile nu poate fi modificată în program, atunci la declararea sa scriem cuvântul cheie *const* imediat în *dreapta* tipului inițial al variabilei. În cazul tipurilor aritmetice *const* poate fi pus și pe primul loc. Inițializarea unei variabile nemodificabile poate fi făcută doar odată cu declararea ei:

```

int const a=5;
const int b=50;
double const d=5.55;
//d=5.55; // you cannot assign to a variable that is const

```

Am utilizat deja astfel de variabile pentru desemnarea dimensiunilor tablourilor alocate static.

În cazul pointerilor, trebuie să avem grijă să nu confundăm declarația unui pointer către un tip constant (la care valoarea țintei este nemodificabilă) cu cea a unui pointer constant (care nu poate să-și schimbe ținta, dar valoarea țintei este modificabilă) sau cu declarația unui pointer constant către un tip constant:

```

#include<iostream>
using namespace std;
int main(){
    //p - pointer variabil catre tip constant:
    int const a=5, b=6;
    int const * p=&a; //<=> const int *p=&a;
    cout<<*p<<endl;//5
    //*p=555; //you cannot assign to a variable that is const
    p=&b; //p isi poate schimba tinta
    cout<<*p<<endl;//6

    //pc - pointer constant catre tip variabil

```

```

int alfa=1,beta=2;
int * const pc=&alfa;
cout<<*pc<<endl; //1
//pc=&beta; // you cannot assign to a variable that is const
*pc=111; //tinta lui pc isi poate schimba valoarea
cout<<*pc<<endl; //111

//pointer constant catre tip constant
int const c=10,d=11;
int const * const pp=&c;
cout<<*pp<<endl; //10
//pp=&d; // you cannot assign to a variable that is const
//*pp=100; // you cannot assign to a variable that is const
return 0;
}

```

## 7. Referințe (specific C++)

Pointerii sunt foarte utili, permit programatorului să utilizeze eficient memoria alocată programului, dar utilizarea lor cere o atenție sporită deoarece erori minore pot avea consecințe grave la rulare, după cum am văzut deja. Din acest motiv unele limbaje de programare provenite din C, au restricționat drastic utilizarea lor, de exemplu C#, iar altele au renunțat complet la pointeri, de exemplu Java.

Pentru a veni în sprijinul programatorilor, limbajul C++ a introdus unele *facilități* care să evite utilizarea intensivă a pointerilor, prin dotarea programatorului cu modalități mai simple de realizare a unor sarcini specifice, sarcini care pot fi rezolvate și cu pointeri, aceștia având rolul de unelte universale. Una dintre aceste facilități constă în introducerea *tipului referință*. O dată de tip referință este o dată capabilă să rețină, în esență, *numele* unui obiect și nu adresa sa, ca în cazul pointerilor. De exemplu, în următoarele linii de cod scrise în C

```

int alfa = 100;
int * const palfa = &alfa;
*palfa = 12;
cout << alfa << endl; //12

```

definim pointerul constant *palfa* care reține adresa variabilei *alfa* și, în acest caz, expresiile *\*palfa* și *alfa* au aceeași r-valoare și aceeași l-valoare, sunt sinonime, oriunde apare *alfa* poate fi scris *\*palfa* și efectul e același, pentru că ele se referă la același obiect.

În C++ putem obține același efect definind o referință:

```

int alfa = 100;
int& beta = alfa;
beta = 12;
cout << alfa << endl; //12

```

Aici am declarat identificatorul *beta* ca fiind o dată de tip “referință către un *int*” și am inițializat-o cu numele unei variabile de tip *int*. În urma acestei declarații, identificatorul *beta* este sinonim cu *alfa*, se spune că *beta* este un “alias” al lui *alfa*, un alt nume. Se mai poate spune și așa: *beta* este referința iar *alfa* este “referitul”, cel la care se referă *beta*. Alt exemplu:

```

typedef int Vector[3];
Vector v = { 1, 2, 3 };
Vector& w = v;
w[0] = 12;

```

```
cout << v[0] << endl; //12
```

Nu este obligatoriu ca “referitul” să fie desemnat printr-un identificator:

```
double tab[3] = { 0.1, 0.2, 0.3 };
double& primul = tab[0];
primul = 1.2;
cout << tab[0] << endl; //1.2
```

Sintaxa declarației este foarte simplă, numele unui tip *t* urmat de & desemnează tipul “referință către *t*”, o dată de acest tip este prin definiție o constantă, ea trebuie inițializată obligatoriu la declarare, orice atribuire ulterioară schimbă numai valoarea variabilei referite, nu numele referitului:

```
int alfa = 100;
//int& beta; //error C2530 : 'beta' : references must be initialized
int& beta = alfa;
int n = 0;
beta = n;
beta = 12;
cout << n << endl; //0
cout << alfa << endl; //12
```

Subliniem că referințele simplifică rezolvarea unor probleme care se pot rezolva cu pointeri, dar ele nu sunt pointeri, de exemplu nu există un calcul aritmetic cu referințe, ele nu au rezervată o locație de memorie, operatorul adresă aplicat unei referințe întoarce adresa referitului.

Utilitatea tipului referință este destul de limitată în interiorul funcțiilor, ea este deplină în cazul apelurilor de funcții. De exemplu, ne propunem să implementăm funcția *schimba*, care să schimbe între ele valorile a două variabile, *a* și *b*, definite la nivelul funcției apelante.

Încercarea următoare este greșită

```
void schimba(int va, int vb){
    int aux = va;
    va = vb;
    vb = aux;
}
int main(){
    int a = 1, b = 10;
    schimba(a, b);
    cout << "a=" << a << " b=" << b << endl;
    //a=1 b=10
    return 0;
}
```

deoarece în momentul apelului funcției *schimba* în variabilele *va* și *vb* sunt încărcate valorile variabilelor *a* și *b*, funcția prelucrând așadar copiile lor.

Rezolvarea corectă, în stil C, este cu pointeri:

```
void schimba(int* pa, int* pb){
    int aux = *pa;
    *pa = *pb;
    *pb = aux;
}
int main(){
    int a = 1, b = 10;
    schimba(&a, &b);
}
```

```

        cout << "a=" << a << " b=" << b << endl;
        //a=10 b=1
        return 0;
    }

```

Acum modificările produse în funcția apelată sunt persistente deoarece aceasta primește chiar adresele variabilelor *a* și *b* și, prin urmare, poate să le schimbe valorile între ele.

O rezolvare mult mai elegantă este, în C++, prin *transmiterea parametrilor prin referință* :

```

void schimba(int& ra, int& rb){
    int aux = ra;
    ra = rb;
    rb = aux;
    return;
}

int main(){
    int a = 1, b = 10;
    schimba(a, b);
    cout << "a=" << a << " b=" << b << endl;
    //a=10 b=1
    return 0;
}

```

Acum funcția *schimba* primește în momentul apelurilor numele variabilelor referite de către referințele *ra* și *rb*, în consecință modificările survenite în funcție sunt iarăși persistente.

Rezultatul unei funcții poate fi de tip referință, caz în care rezultatul apelului are atât *valoare numerică* (valoarea variabilei referite) cât și *valoare de locație* (adică poate sta în partea stângă a unei atribuirii). Exemplu:

```

int & maxim(int &a, int &b){
    if (a<b) return b;
    else return a;
}

int main(void){
    int x = 7, y = 13;
    maxim(x, y) = 10;
    cout << "x=" << x << " y=" << y << endl;
    return 0;
}

/* rezultat:
x=7 y=10
Press any key to continue . . .*/

```

În atribuirea `maxim(x,y)=10` folosim valoarea de locație a rezultatului returnat de funcția *maxim*. La fel de bine poate fi folosită și valoarea numerică a rezultatului, printr-o expresie de forma `z=10+maxim(x,y)`, de exemplu.

Greșeala tipică în utilizarea acestei modalități de returnare este întoarcerea unei referințe la o variabilă locală funcției apelate - care după cum știm este dealocată de pe stivă la terminarea apelului. În exemplul următor

```

int & maxim(int& a, int& b){
    int max = a;
    if (a < b) max = b;
    return max;
}
int main(void){
    int x = 7, y = 13;
    maxim(x, y) = 10;
    cout << "x=" << x << " y=" << y << endl;
    return 0;
}
/* rezultat:
x=7 y=13
Press any key to continue . . .*/

```

observăm că apelul `maxim(x,y)=10` nu-l mai modifică pe `y` în 10, deoarece *maxim* întoarce acum o referință “în aer”, către locul unde fusese alocată variabila *max* în timpul apelului funcției *maxim*. De altfel, compilatorul ne avertizează: warning C4172: returning address of local variable or temporary.

Întrebare: de ce nici următoarea variantă a funcției *maxim* nu este corectă ?

```

int& maxim(int a, int b){
    int max = a;
    if (a < b) max = b;
    return max;
}

```

În final, deoarece utilizarea returnării prin referință este pândită de numeroase primejdii, nu recomandăm începătorilor să utilizeze această posibilitate.