Fundamentele Programarii

Pointeri si referinte

Pointeri si referinte

- Ce este un pointer?
- Cum se declara un pointer?
- Pastrarea adreselor pointerilor
- Alocarea dinamica a memoriei
- Aritmetica pointerilor
- Pointeri si tablouri
- Apelul prin referinta cu pointeri
- Ce este o referinta?

Ce este un pointer?

- Un *pointer* este o variabila a carui valoare este o adresa a unui obiect de un tip bine precizat, numit *ţinta* pointerului
- Ce poate fi la adresa respectiva?
    - O alta variabila
    - O functie
- Daca x este o variabila intreaga a carui valoare este 10, atunci putem declara un pointer care pointeaza la ea; iar pointerul respectiv este stocat la o anumita adresa, are tipul int, iar valoarea sa este data de adresa lui x
- Trebuie sa stim intotdeauna tipul catre care pointeaza pointerul

De ce sa folosim pointerii?

- Nu putem folosi doar variabilele sau functiile insisi?
- DA, dar nu mereu

- In cadrul functiilor, pointerii pot fi folositi pentru a accesa date care sunt definite in afara functiei. Acele variabile pot sa nu fie accesibile prin numele lor

- Pointerii pot fi folositi pentru a lucra cu tablouri mai eficient
- Se pot aloca dinamic in memorie In zona HEAP: aceasta memorie nu contine nume de variabile, iar singurul mod de a ne referi la ea este prin pointeri
- Putem accesa adrese specifice din memorie!

Declararea pointerilor

- Sintaxa:

tipul_variabilei *nume_pointer;

- Exemple:

```
int *int_pointer;
double* double_pointer;
char *char_pointer;
string *string_pointer;
```

- Daca nu ii initializam, ei vor contine “garbage data” si ei “pot pointa oriunde”

Declararea pointerilor

- Initializarea pointerilor “care nu pointeaza nicaieri”:

tipul_variabilei *nume_pointer {nullptr};

- Exemple:

```
int *int_pointer { };
double* double_pointer {nullptr};
char *char_pointer {nullptr};
string *string_pointer {nullptr};
```

- **nullptr** a fost introdus in C++11 si contine adresa zero

- Initializati intotdeauna pointerii!

Accesarea adreselor pointerilor

- Operatorul adresa **&**

- Fiecare variabila este memorata la o adresa unica
- Operator unar
- Rezultatul folosirii lui este adresa operandului sau (deci operandul nu poate fi o constanta)

```
int num{ 10 };

cout << "Valoarea lui num este: " << num << endl;  //10
cout << "Variabila num ocupa " << sizeof(num) << " bytes" << endl;  //4
cout << "Adresa lui num este " << &num << endl; //00AFFB68
```

Accesarea adreselor pointerilor

- Exemplu: (se va genera o eroare deoarece p nu este initializat)

```
int *p;

cout << "Valoarea lui p este: " << p << endl;  //garbage CCCCCCCC
cout << “Pointerul p ocupa " << sizeof(p) << " bytes" << endl; //4
cout << "Adresa lui p este " << &p << endl; //0073F744
p = nullptr; //p nu pointeaza nicaieri
cout << "Valoarea lui p este: " << p << endl; //00000000
```

Accesarea adreselor pointerilor

- **Operatorul sizeof aplicat unui pointer**

- Sa nu confundam marimea pointerului cu marimea obiectului catre care pointeaza pointerul
- Toti pointerii din program au aceeasi marime
- Ei pot pointa cate tipuri foarte mari sau foarte mici

```
#include<iostream>
#include<string>
#include<vector>
using namespace std;
int main() {
    int *p1{nullptr};
    double *p2{ nullptr };
    unsigned long long *p3{ nullptr };
    vector <string> *p4{ nullptr };
    string *p5{ nullptr };
    cout << "Pointerul p1 ocupa " << sizeof(p1) << " bytes" << endl;
    cout << "Pointerul p2 ocupa " << sizeof(p2) << " bytes" << endl;
    cout << "Pointerul p3 ocupa " << sizeof(p3) << " bytes" << endl;
    cout << "Pointerul p4 ocupa " << sizeof(p4) << " bytes" << endl;
    cout << "Pointerul p5 ocupa " << sizeof(p5) << " bytes" << endl;
    return 0;
}
```

```
4
4
4
4
4
```

Accesarea adreselor pointerilor

- **Tipul catre care pointeaza un pointer**
- Compilatorul se asigura daca adresa memorata intr-un pointer are tipul corect

```
int scor{ 10 };
double temperatura{ 13.7 };

int *p{ nullptr };
p = &scor;

p = &temperatura;
```

p=0019FDDC

Eroare a compilatorului!!!

Pastrarea unei adrese intr-o variabila

Operatorul adresa &

- Pointerii sunt variabile deci isi pot schimba valoarea pe parcursul programului
- Pointerii pot fi nuli
- Pointerii pot sa nu fie initializati (nu este recomandat)

```
double greutate { 10.4 };
double temperatura{ 13.7 };

double *p;  //pointeaza oriunde

p = &greutate;
cout << p <<endl;

p = &temperatura;
cout << p << endl;

p = nullptr;  //nu pointeaza nicaieri
cout << p << endl;
```

00DBF7B0

00DBF7A0

00000000

Accesarea datelor la care pointeaza pointerii

- Dereferentierea pointerilor-

- Daca p este un pointer si are ca valoare o adresa valida, atunci putem accesa entitatea care se afla la adresa respectiva cu ajutorul operatorului de dereferentiere *

Exemplu:

```
int scor{ 10 };
int *p2{ &scor };

cout << *p2 << endl; //10
*p2 = 9;
cout << *p2 << endl; //9
cout << scor << endl; //9
```

Accesarea datelor la care pointeaza pointerii

- Dereferentierea pointerilor-

Exemplu:

```
double greutate { 10.4 };
double temperatura{ 13.7 };

double *p{ &greutate };

cout << *p << endl;     // 10.4
cout << &greutate << endl;  // 00AFFC70
cout << p << endl;          // 00AFFC70


p = &temperatura;
cout << *p << endl;      // 13.7
cout << &temperatura << endl;  // 00AFFC60
cout << p << endl;             // 00AFFC60
```

Accesarea datelor la care pointeaza pointerii

- Dereferentierea pointerilor-

Exemplu:

```
string nume{ "Simona" };

string *s{ &nume };
cout << *s << endl; //Simona
nume = "Elena";
cout << *s << endl; //Elena
```

Accesarea datelor la care pointeaza pointerii

- Dereferentierea pointerilor-

Exemplu:

```
int scor{ 100 };
int *p{ &scor };
cout << *p << endl;
*p = 200;
cout << scor << endl;
cout << *p << endl;
```

100

200

200

Accesarea datelor la care pointeaza pointerii

- Dereferentierea pointerilor-

Exemplu:

```
double temp_mare{ 28.4 };
double temp_mica{ 3.2 };
double *p{ &temp_mare };

cout << *p << endl;
p = &temp_mica;
cout << *p << endl;
*p = 14.3;
cout << temp_mare << endl;
cout << temp_mica << endl;
```

28.4

3.2

28.4

14.3

Alocarea dinamica a memoriei

- De multe ori nu stim de cata memorie avem nevoie de la inceput
- Putem aloca memorie pentru o variabila in timpul executiei

- Reamintim ca la tablouri dimensiunea maxima trebuie precizata si fixata de la inceput, in timp ce in cazul vectorilor putem creste in mod dinamic dimensiunea

- Putem folosi pointerii pentru a aloca un spatiu nou in memoria heap

Alocarea dinamica a memoriei

- Folosirea cuvantului cheie **new** pentru alocarea memoriei

```
int *pointer_catre_int{ nullptr };

pointer_catre_int = new int; //se aloca spatiu in memoria heap pentru un intreg

cout << pointer_catre_int << endl; //007A0578

cout << *pointer_catre_int << endl; //-842150451  --gunoi

*pointer_catre_int = 100;

cout << *pointer_catre_int << endl; //100
```

Alocarea dinamica a memoriei

- Folosirea cuvantului cheie **delete** pentru stergerea din memorie

```
int *pointer_catre_int{ nullptr };
pointer_catre_int = new int; //se aloca spatiu in memoria heap pentru un intreg
cout << pointer_catre_int << endl; //007A0578
cout << *pointer_catre_int << endl; //-842150451  --gunoi
*pointer_catre_int = 100;
cout << *pointer_catre_int << endl; //100
delete pointer_catre_int;
```

Alocarea dinamica a memoriei

- Folosirea cuvantului cheie **new** pentru alocarea memoriei heap necesara unui tablou si a cuvantului cheie **delete** pentru stergerea din memeoria heap a tabloului

```
int *pointer_catre_tablou{ nullptr };
int dim;

cout << "Care doriti sa fie lungimea tabloului? ";
cin >> dim;

pointer_catre_tablou = new int[dim];

cout << pointer_catre_tablou << endl;  //adresa primului element din tablou

delete[] pointer_catre_tablou;
```

Relatia dintre tablouri si pointeri

- Valoarea numelui unui tablou este adresa primului element din tablou;
- Valoarea unui pointer este o adresa
- Daca pointerul pointeaza la acelasi tip de date ca un element din tablou atunci pointerul si numele tabloului se pot interschimba (singura diferenta este ca numele tabloului nu este o variabila, deci nu se poate schimba)

Relatia dintre tablouri si pointeri

- Exemplu:

```
int note[]{ 10,9,8 };
cout << note << endl; //00EFFB90  adresa primului element din tablou
cout << *note << endl; //10

int *pointer_catre_note{ note };

cout << pointer_catre_note << endl; ///00EFFB90
cout << *pointer_catre_note << endl; //10
```

Relatia dintre tablouri si pointeri

- Exemplu:

```
int note[]{ 10,9,8 };

int *pointer_catre_note{ note };

cout << pointer_catre_note[0] << endl;  //10
cout << pointer_catre_note[1] << endl;  //9
cout << pointer_catre_note[2] << endl;  //8
```

Relatia dintre tablouri si pointeri

- Exemplu:

```
int note[]{ 10,9,8 };

int *pointer_catre_note{ note };

cout << pointer_catre_note << endl;  //00BEF988
cout << (pointer_catre_note+1) << endl;  //00BEF98C  = 00BEF988 + 4
cout << (pointer_catre_note+2) << endl; //00BEF990   = 00BEF98C + 4
```

Relatia dintre tablouri si pointeri

- Exemplu:

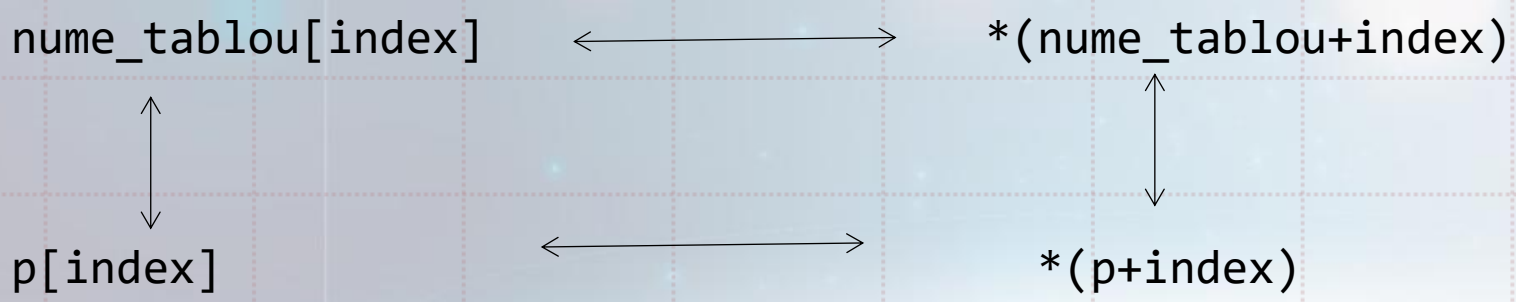
```
int note[]{ 10,9,8 };

int *pointer_catre_note{ note };

cout << *pointer_catre_note << endl;  //10
cout << *(pointer_catre_note+1) << endl;  //9
cout << *(pointer_catre_note+2) << endl; //8
```

Relatia dintre tablouri si pointeri

```
int nume_tablou [] {1,2,3,4,5};
int *p {nume_tablou};
```



Aritmetica pointerilor

Operaţiile aritmetice cu pointeri au fost concepute în principal pentru parcurgerea tablourilor, mai precis, pentru accesarea datelor stocate în zone de memorie organizate ca locaţii succesive de aceeaşi mărime. Să reamintim că mărimea locaţiei de memorie a unei variabile sau a unui tip poate fi aflată cu operatorul ***sizeof***.

Avem următoarele reguli de calcul:

- mărimea unui tip simplu = ***sizeof***(tip);
- mărimea unui tablou = nr.elemente**sizeof**(tip_element);

- Operaţiile aritmetice sunt permise numai pentru pointeri care au mărimea ţintei bine definită (şi deci ţinta poate fi element al unui tablou). Sunt definite numai următoarele operaţii:

Aritmetica pointerilor

I) ***Incrementare/decrementare***.

Dacă ***p*** este o *variabilă pointer* pentru care este definită mărimea ţintei ****p,*** atunci sunt permise operaţiile ***p++***, ***p--***, ***++p*** şi ***--p***, care au aceeaşi interpretare ca în cazul variabilelor aritmetice, cu singura deosebire că pasul incrementării este egal cu mărimea ţinei:

```
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***).

Aritmetica pointerilor

II) ***Suma şi diferenţa dintre un pointer şi un întreg.***

Dacă ***p*** este un pointer iar ***i*** este un întreg, expresiile ***p+i*** şi ***i+p*** au ca rezultat valoarea lui ***p*** mărită cu ***i**sizeof**(****p***), iar diferenţa ***p-i*** are ca rezultat valoarea lui ***p*** micşorată 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=0033FDEC+2*4
cout<<"pp-1="<<pp-1<<endl; //pp-1=0033FDE8=0033FDEC-1*4
//cout<<"i-pp="<<i-pp<<endl;
//error: pointer can only be subtracted from another pointer
pp+=5;
cout<<"pp  ="<<pp  <<endl; //pp  =0033FE00=0033FDEC+5*4
```

Aritmetica pointerilor

III) ***Diferenţa a doi pointeri.***

Este permisă scăderea a doi pointeri de acelaşi tip, rezultatul este de tip **int**, iar pasul operaţiei este egal cu mărimea tipului ţintă.

```
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=12/4
                                (0012FF60-0012FF54=C=12(10))
```

Aritmetica pointerilor

III) ***Comparaţia a doi pointeri.***

Doi pointeri de acelaşi tip pot fi comparaţi cu operatorii <, <=, ==, >= şi >.
Un caz particular îl reprezintă comparaţia cu zero, care este permisă, zero fiind asimilat cu pointerul nul.

```
int a, b, *p = &a, *q = &b;
double bb, *pp = &bb;
cout << p << endl; //010FFCA4 = 17824932 (in baza 10)
cout << q << endl; //010FFC98 = 17824920 (in baza 10)
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 *'
```

Pointeri si constante

- Pointeri care tintesc catre constante
- Pointeri constanti
- Pointeri constanti care tintesc catre constante

Pointeri a caror tinte sunt constante

- Valoarea obiectului catre care pointeaza pointerul este constanta si deci nu poate fi schimbata
- Pointerul poate sa se schimbe si sa pointeze la alt obiect

```
int notaMare{ 10 };
int notaMica{ 4 };
const int *p{ &notaMare };

*p = 9; //EROARE
p = &notaMica;
```

Pointeri constanti

- Valoarea obiectului catre care pointeaza pointerul poate fi schimbata
- Pointerul nu poate sa pointeze la alt obiect

```
int notaMare{ 10 };
int notaMica{ 4 };
int *const p{ &notaMare };

*p = 9; //OK
p = &notaMica; //EROARE
```

Pointeri constanti care tintesc catre constante

- Valoarea obiectului catre care pointeaza pointerul nu poate fi schimbata
- Pointerul nu poate sa pointeze la alt obiect

```
int notaMare{ 10 };
int notaMica{ 4 };
const int *const p{ &notaMare };

*p = 9; //EROARE
p = &notaMica; //EROARE
```

Apelarea functiilor prin referinta cu ajutorul pointerilor

Exemplu:

```
void  dubleaza(int *p) {
        *p *= 2;
}

int main(){
    int a{ 10 };
    int *p{nullptr};
    cout << "a = " << a << endl;
    dubleaza(&a);
    cout<< "a = " << a << endl;


    p = &a;
    dubleaza(p);
    cout<< "a = " << a << endl;

    return 0;
}
```

10

20

40

Apelarea functiilor prin referinta cu ajutorul pointerilor

Exemplu:

```
void schimba(int *a, int *b) {
    int aux = *a;
    *a = *b;
    *b = aux;
}

int main() {
    int x{ 100 }, y{ 200 };
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    schimba(&x, &y);
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}
```

```
100
200
200
100
```

Returnarea unui pointer de catre o functie

- Functiile pot returna pointeri (acestia nu trebuie sa pointeze catre variabile locale declarate in cadrul functiei)

type *functie();

```
int *max(int *p1, int *p2) {
     if (*p1 > *p2)
     return p1;
     else
     return p2;
}

int main() {
          int a{ 100 };
          int b{ 200 };
          int *maxim{ nullptr };
          maxim = max(&a, &b);
          cout << *maxim << endl;
          return 0;
}
```

Ce este o referinta?

- Un alias pentru o variabila
- Trebuie sa se refere la o anumita variabila atunci cand este declarata
- Nu poate fi nula
- O data ce au fost initializate nu se pot referi la o alta variabila
- Foarte folositoare ca parametru de functii
- Putem gandi o referinta ca un pointer constant care este in mod automat dereferentiat.

Ce este o referinta?

- Exemplu:

```
int num{ 100 };
int &ref{ num };

cout << num << endl;                100
cout << ref << endl<<endl;          100

num = 200;
cout << num << endl;                200
cout << ref << endl<<endl;          200

ref = 300;
cout << num << endl;                300
cout << ref << endl;                300
```

L-values si R-values

- **L-values**
  - Valori care au nume si admit o adresa
  - Se pot modifica daca nu sunt constante

```
int x{ 100 }; //x este o l-value
x = 1000;
x = 1000 + 20;

string nume; // nume este o l-value
nume = "Simona";

_____________________________________

100 = x;  //100 NU este o l-value

(1000 + 20) = x; // (1000+20) NU este o l-value (nu are o adresa)

"Simona" = nume; //"Simona" NI este o l-value
```

L-values si R-values

- **R-values**
  - Valori care nu admit o adresa si nu li se poate asocia o valoare
  - R-value este o valoare care nu este o l-value
    - Ce se afla in partea dreapta a operatorului de atribuire
    - Un literal
    - O variabila temporara (care nu admite o adresa)

```
int x{ 100 }; //100 este o r-value
x = 1000;    //1000 este o r-value
int y;
y= x + 20; // (x+20) este o r-value

string nume;
nume = "Simona"; // "Simona" este o r-value

int num_max = max(20, 30); //max(20,30) este o r-value
```

L-values si R-values

- **R-values pot fi atribuite lui l-values in mod explicit**

```
int x{ 100 }; //100 este o r-value
int y{ 0 };
y= x + 20; // (x+20) este o r-value

y=100; // r-value 100 este atribuita lui l-value y

x=x+y; // r-value x+y este atribuita lui l-value x
```

L-values si R-values

- **Referintele se refera la l-values**

- **Exemplul1:**

```
int x{ 100 }; //x este o l-value
int  &ref1 = x; // ref1 este referinta la l-value x
ref1=1000;

int  &ref2 = 100; // Eroare: 100 este o r-value
```

- **Exemplul2:**

```
int patratul_unui_numar(int &n) {
return n * n;
}
int main() {
int num{ 10 }; //num este l-value
patratul_unui_numar(num); //OK

patratul_unui_numar(10); //EROARE: 10 este o r-value
}
```