

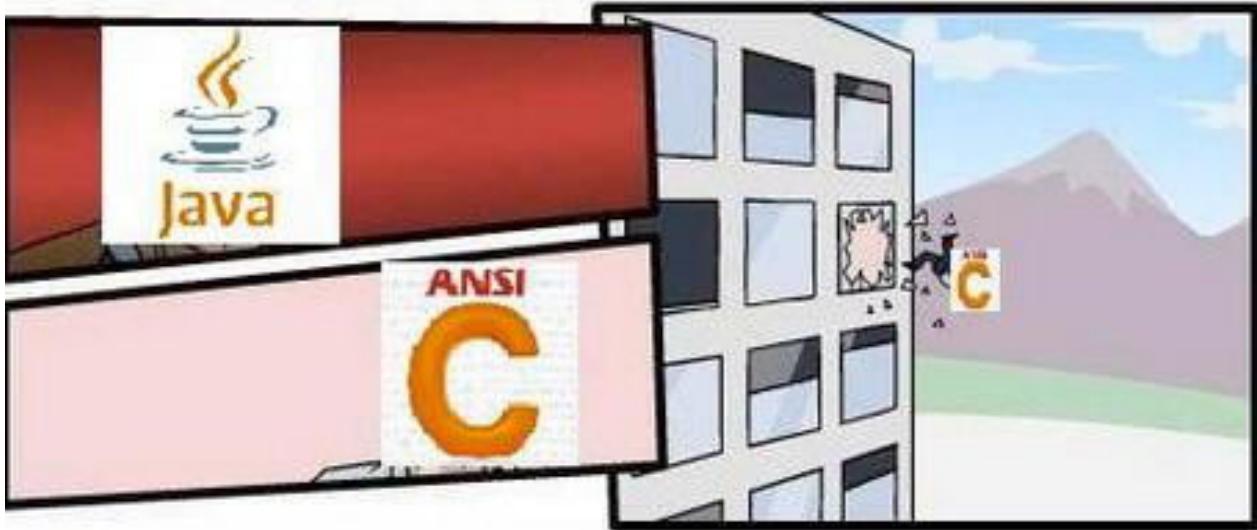
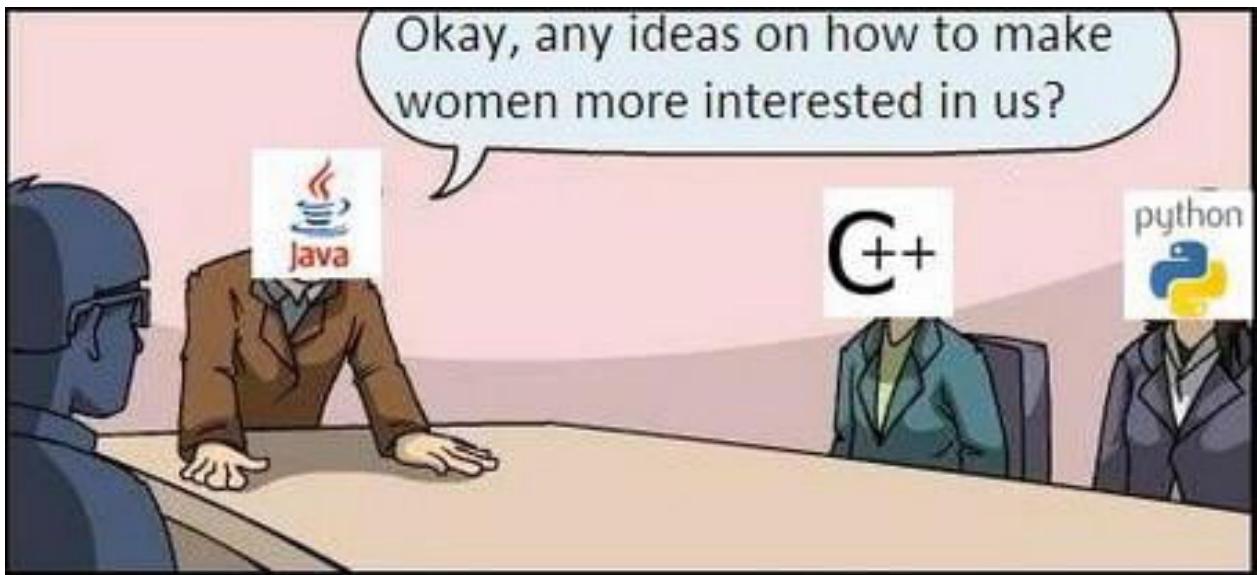
Tutoriat 1 POO

Bianca-Mihaela Stan, Stăncioiu Silviu

March 2021

Contents

1	Pointeri si referinte	3
1.1	Adrese de memorie	3
1.2	Alocare dinamica	6
1.3	Aritmetica pointerilor	13
1.4	Referințe	15
1.5	Const	20
1.6	Rvalues si lvalues	20
2	Incapsularea	21
2.1	Struct-ul din C	21
2.2	Clasele din C++	22
2.3	Specificatori de acces	22
2.4	Getters si setters	25
2.5	Obiectele	26
2.6	Constructori	28
2.6.1	Constructorul de initializare	28
2.6.2	Constructorul de copiere	29
2.6.3	Destructorul	33
2.6.4	Constructorii impliciti	35
2.7	Lista de initializare (Member initializer lists)	38
2.7.1	Ordinea in care se initializeaza datele	41
3	Compozitia	41
4	Exercitii	42
5	Resurse	47



1 Pointeri și referințe

1.1 Adrese de memorie

Pe partea asta de pointeri și referințe nu prea se pune accentul în liceu (mai ales la clasele care nu au fost la mate-info), și nici pe semestrul 1 de la facultate întrucât s-a făcut Python în loc de C.

Pointerii sunt niște numere care reprezintă unde se află anumite obiecte în memorie. Practic sunt adrese de memorie, adrese de memorie ca cele cu care lucram în MIPS. Deci în MIPS foloseam niște pointeri undercover.

Pentru a înțelege cum funcționează adresele de memorie, uitați-vă în tutoriatul de ASC.

Să presupunem că suntem în MIPS și dorim să aflăm adresa de memorie a unei valori din memorie, iar apoi să o afișăm pe ecran, în acest caz am avea:

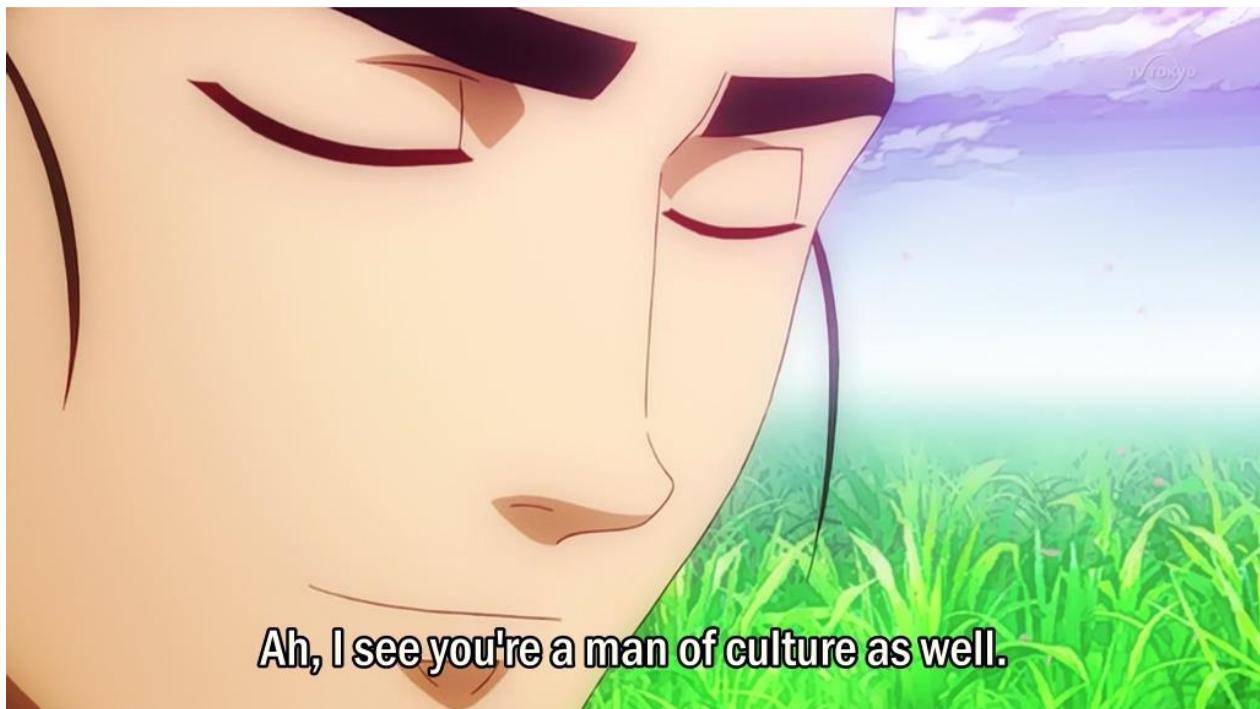
```
# EXEMPLUL 1
.data
    x:.word 10
.text
main:
    la $t0, x      # obtine adresa de memorie a lui x si o pune in registrul t0

    move $a0, $t0 # afiseaza adresa de memorie a lui x pe ecran
    li $v0, 1
    syscall

    li $v0, 10
    syscall
```



Când un student din anul 1 mai stie MIPS după ce a luat examenul de ASC.



Se observă că am folosit instrucțiunea *la* pentru a lua adresa lui *x* din memorie și că adresa de memorie este de fapt doar un număr întreg. Orice adresă de memorie este un număr întreg. Deci putem salva o adresă de memorie într-un *int* din C++ de exemplu. Pentru a nu exista confuzie între numere normale și adrese de memorie, în C/C++, adresele de memorie se numesc pointeri, iar tipurile de date pentru adrese de memorie sunt postfixate cu *. De exemplu, o adresă de memorie pentru un *int* se va stoca în o variabilă de tipul *int**.

Exemplu de adresă de memorie:

```
int *a = 0x37FFFF;
```

Acum, vom vrea să implementăm exemplul din MIPS de mai sus în C++. Adică să afișăm pe ecran adresa unei variabile din memorie.

Pentru asta vom folosi următorul cod:

```
//EXEMPLUL 2
#include <iostream>

using namespace std;

int x = 10;

int main()
{
```

```

int* addr = &x; // obtine adresa de memorie al lui x si o salveaza
// in variabila de tip pointer addr.

cout << addr;

return 0;
}

```

Observăm că acum pentru a afla o adresă de memorie nu avem o instrucțiune precum *la* din MIPS, în schimb avem operatorul **&**. Cu operatorul **&** putem afla adresa de memorie a oricărei variabile. Putem afla chiar și adresa de memorie a unei variabile care stochează o adresă de memorie, haha.

Ok, avem o adresă de memorie, acum noi cum facem să citim sau să modificăm ce se află la acea adresă de memorie? Pentru asta avem operatorul *****. Folosim acest operator asupra unei variabile de tip pointer pentru a citi/ modifică ce se află la acea adresă de memorie.

Exemplu:

```

//EXEMPLUL 3
int b = 20;
int *a = &b; // salveaza adresa de memorie a lui b in a

int c = *a; // salveaza valoarea care se afla la adresa de memorie
// indicata de a in variabila c
*a = 10; // pune valoarea 10 la adresa de memorie catre care
// indica pointerul a

```

Practic, pentru *int*, operatorul ***** este echivalentul lui *sw* și *lw* din MIPS, intrucât ne permite atât să modificăm valoarea de la o adresă de memorie, cât și să aflăm ce se află la acea adresă de memorie.

Fie o variabilă de tip *int* în memorie. Să se afișeze valoarea ei, să se modifice valoarea ei, iar apoi să se afișeze noua valoare folosind doar pointeri și operatorii **&**, respectiv *****.

```

//EXEMPLUL 4
#include <iostream>

using namespace std;

int x = 10;

int main()
{
    int *addr = &x;

    cout << *addr << endl; // afiseaza valoarea care se afla la adresa de
    // memorie catre care pointeaza addr, adica
    // valoarea lui x, adica 10.

    *addr = 100; // modifica valoarea care se afla la adresa de
    // memorie addr, acum la acea adresa de memorie
    // se va afla valoarea 100. Se observa ca si x
    // are valoarea 100 acum deoarece am modificat
    // valoarea care se afla la adresa lui de
    // memorie.

```

```

cout << *addr << endl; // afiseaza valoarea care se afla la adresa de
// memorie catre care pointeaza addr, adica
// valoarea lui x, adica 100.

return 0;
}

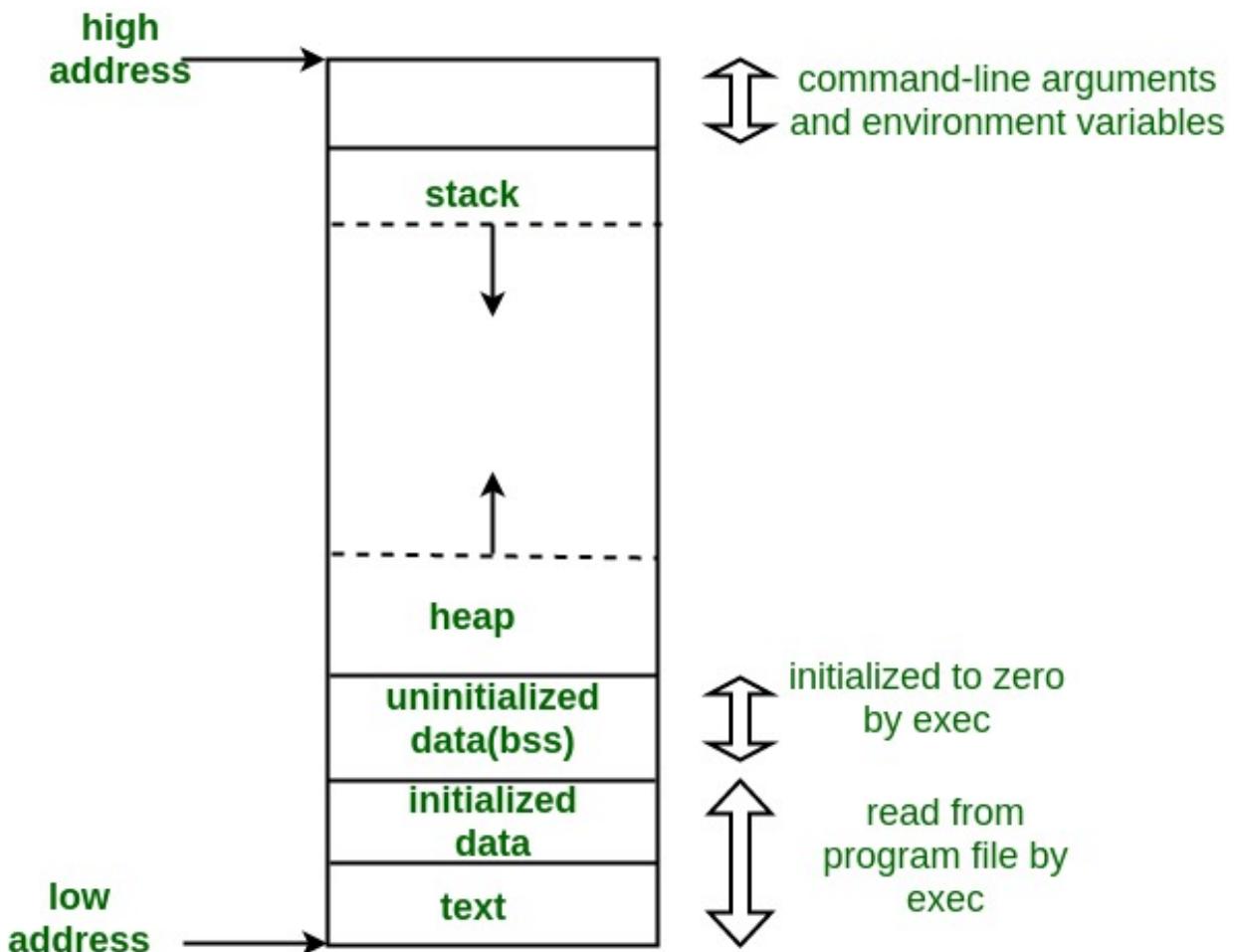
```

Ok, cool, chestiile astea merg și pe alte tipuri de date, sau doar pe int? Wellp, merg pe toate tipurile de date, pe orice tip de date operatorii acestia se comportă la fel.

1.2 Alocare dinamica

In MIPS nu am acoperit alocarea dinamică de memorie... să că o vom acoperi acum în C++. :) După cum știți, în MIPS noi țineam datele în memorie fie în `.data`, fie pe stivă în cadrul procedurilor. Ambele zone de memorie au o limitare, și anume faptul că nu putem avea pe ele obiecte de o dimensiune decisă la runtime. Putem avea doar obiecte de dimensiune fixă pe ele.

Reamintim diagrama simplistă cu așezarea unui program în memorie:



Observăm în această diagramă că avem la dispoziție și o zonă numită *heap* unde putem stoca date. În această zonă de memorie putem aloca obiecte de dimensiune decisă la runtime, nu o dimensiune fixă (statică). De aici și numele de alocare dinamică, din faptul că putem aloca un număr dinamic de bytes în memorie.

În C++ folosim `new` pentru a aloca o zonă de memorie. `new` ne va returna un pointer către adresa de

memorie nou-alocată. Este de datoria noastră să eliberăm acea zonă de memorie după ce nu o mai folosim, deoarece calculatorul nu face asta automat. Dacă nu eliberăm zonele de memorie pe care le alocam dinamic este posibil să avem *memory leak – uri*, adică memorie care este alocată, dar nu mai este folosită niciodată. Chiar mai grav este când pierdem pointerii către acele zone de memorie întrucât nici dacă am vrea nu am mai putea să o eliberăm din moment ce nu mai avem pointerii. Pentru a elibera zonele de memorie alocate dinamic folosim *delete*, respectiv *delete[]* în cazul array-urile alocate dinamic.

Observații:

- Mereu când alocăm ceva dinamic trebuie să nu uităm să dăm *delete*. (btw, se scade dacă la proiecte/ colocviu aveți *memory leak*-uri)
- Atunci când dăm *delete* la un pointer trebuie să ne asigurăm că memoria de acolo nu a fost deja eliberată. O bună practică este ca pointerii care nu duc către o adresă de memorie validă să aibă una dintre valorile 0, *NULL* sau *nullptr*.
- *new* și *delete* sunt specifice C++, pentru a aloca memorie dinamic în C se folosesc funcții precum *malloc*, *calloc*, *free*...
- Este imposibil să aflăm câți bytes sunt alocați la o adresă de memorie, putem doar deduce în timpul alocării cât am alocat. Acum vă puteți întreba cum știe calculatorul cât să steargă când dăm *delete*, din moment ce nu putem afla numărul de bytes alocați. Well, intern *new* și *delete* sunt de fapt niște syscall-uri către sistemul de operare. Sistemul de operare are un tabel cu cât ocupă fiecare lucru pe care îl alocă, deci știe cât să steargă când vine vorba de eliberat memorie. Noi nu avem acces la aceste informații deoarece depind de implementarea sistemului de operare, ci nu de C/C++.





Exemplu:

Se citesc două numere întregi pe 32 de biți de la tastatură. Să se afișeze pe ecran suma lor. Se va folosi alocare dinamică.

//EXEMPLUL 5

```

#include <iostream>

using namespace std;

int main()
{
    int *a = new int; // aloca memorie cat pentru un int
                      // si pastreaza adresa de memorie
                      // in variabila a

    int *b = new int; // aloca memorie pentru inca un int

    cin >> *a;          // citeste de la tastatura si salveaza
                        // rezultatul la adresa de memorie indicata
                        // de a
    cin >> *b;          // citeste de la tastatura si salveaza
                        // rezultatul la adresa de memorie indicata
                        // de b

    cout << *a + *b;   // afiseaza suma celor doua numere

    delete b;           // elibereaza zona de memorie alocata pentru
                        // al doilea numar
    b = nullptr;        // atribuie lui b valoarea nullptr pentru a
                        // indica faptul ca b nu mai pointeaza
                        // catre nicio zona de memorie. In cazul
                        // acesta nu este necesar deoarece nu
                        // se va mai folosi pointerul b in
                        // continuarea programului. Am pus asta aici
                        // doar pentru ca este un good practice.

    delete a;           // elibereaza zona de memorie alocata
                        // pentru primul numar
    a = nullptr;        // atribuie lui a valoarea nullptr

    return 0;
}

```

Observații

- Când alocăm ceva dinamic există posibilitatea ca alocarea să eșueze, caz în care trebuie să verificăm acest lucru. În exemplul de mai sus nu am luat eșuarea în calcul deoarece este un exemplu didactic banal și s-ar fi complicat codul degeaba. Vom reveni asupra acestui lucru când vom ajunge la excepții.
- Înainte de a da `delete` este nevoie de a ne asigura că există ceva la acea adresă de memorie. Pentru asta folosim acel `nullptr`. Într-o situație reală un `delete` ar arăta în felul urmator:

```

//EXEMPLUL 6
if (ptr)
{
    delete ptr;
    ptr = nullptr;
}

```

Reamintim că un array este caracterizat doar de adresa lui de memorie de început și de lungimea sa (vezi tutoriat ASC).

Pentru a aloca dinamic un array în C++ vom folosi *new tip_de_date[lungime]*. Aceasta ne va returna o adresă de memorie către începutul array-ului nou-alocat.

Exemplu:

Se citește $n \in N$ de la tastatură, urmat de n numere. Să se salveze aceste numere într-un array alocat dinamic, apoi să se afișeze.

```
//EXEMPLUL 7
#include <iostream>

using namespace std;

int main()
{
    int n, *vec;                      // numarul n si un pointer catre
                                         // array

    cin >> n;
    if (n < 0)
    {
        cout << "Numarul citit nu este un numar natural.";
        return 0;
    }

    vec = new int[n];                  // aloca dinamic un array de
                                         // int-uri de dimensiune n si
                                         // pastreaza pointerul catre
                                         // inceputul lui in variabila vec.

    for (int i = 0; i < n; i++)       // side note: eu as folosi auto in
                                         // loc de int in acest caz, dar
                                         // asta este o poveste pentru alt
                                         // tutoriat

        cin >> vec[i];                // se observa ca acum folosim
                                         // array-ul fix ca pe un array
                                         // alocat static

    for (int i = 0; i < n; i++)
        cout << vec[i] << " ";

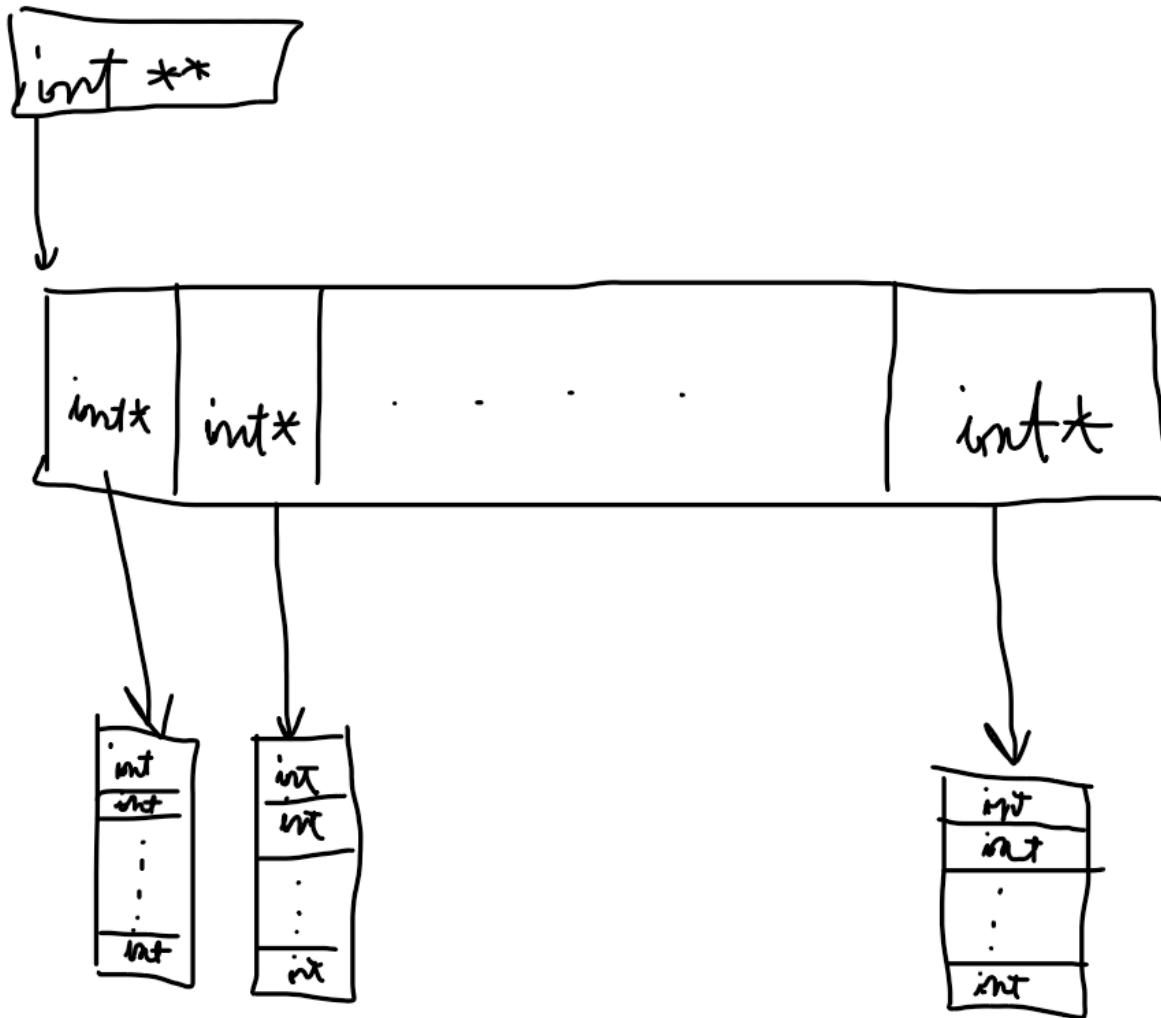
    delete[] vec;                     // elibereaza zona de memorie
                                         // alocata pentru array. Se observa
                                         // ca am folosit delete[] in loc de
                                         // delete, deoarece folosim un
                                         // array de obiecte, ci nu un
                                         // singur obiect.

    vec = nullptr;

    return 0;
}
```

În mod similar, putem aloca dinamic și un tablou bidimensional. Vom avea un array de pointeri care pointează

către array-uri unidimensionale alocate dinamic. Evident, putem aloca dinamic atât tablourile unidimensionale, cât și tabloul de tablouri.



Exemplu:

Se citesc $n, m \in N$ de la tastatură, urmate de un tablou bidimensional de numere întregi cu n linii și m coloane. Să se salveze aceste numere într-un tablou alocat dinamic, iar apoi să se afișeze.

```
//EXEMPLUL 8
#include <iostream>

using namespace std;

int main()
{
    int n, m, **mat;           // numerele n, m si pointerul catre
                               // tabloul nostru.
                               // Se observa ca acum apare de doua
```

```

// ori * inainte de mat. Asta este
// deoarece noi avem un pointer catre
// un tablou cu elemente de tipul
// (int*). Deci una din * este de la
// tipul (int*), iar cealalta este de
// la tabloul de tablouri.

cin >> n >> m;

if (n < 0 || m < 0)
{
    cout << "Dimensiune invalida.";
    return 0;
}

mat = new int*[n];           // aloca un array cu elemente de
                            // tipul (int*) de lungime n
for (int i = 0; i < n; i++)
    mat[i] = new int[m];   // pentru fiecare element din
                            // array-ul de array-uri, alocă
                            // dinamic un array de int-uri

for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        cin >> mat[i][j]; // folosim tabloul ca pe un tablou
                            // bidimensional normal

for (int i = 0; i < n; i++)
{
    for (int j = 0; j < m; j++)
        cout << mat[i][j] << " ";

    cout << endl;
}

for (int i = 0; i < n; i++)
{
    delete[] mat[i];      // stergem fiecare tablou
                            // unidimensional alocat. Daca am fi
                            // facut doar delete[] mat; am fi avut
                            // un memory leak deoarece nu am
                            // fi sters si aceste tablouri
                            // unidimensionale alocate.
                            // Ar fi fost foarte grav
                            // deoarece am fi si pierdut
                            // pointerii catre ele, deci ne-ar
                            // fi fost imposibil sa le mai
                            // stergem.

    mat[i] = nullptr;
}

delete[] mat;              // acum ca am sters tablourile
                            // unidimensionale, putem sterge
                            // in siguranta si tabloul de

```

```
// tablouri  
mat = nullptr;  
  
return 0;  
}
```



Concepție de Programare Orientată pe Obiecte

Tablouri bidimensionale alocate dinamic

1.3 Aritmetică pointerilor

Adresele de memorie fiind doar niste numere, putem realiza operații matematice asupra lor. Țineți minte cum făceam la MIPS, pentru a accesa un element de la o poziție dintr-un array era necesar să adunăm la adresa sa de memorie indexul pe care vrem să-l accesăm, înmulțit cu dimensiunea tipului de date utilizat (în cazul word-urilor era 4). Același lucru rămâne valabil și în C/C++. Noi în C/C++ suntem obișnuiți să folosim următoarea sintaxă pentru a accesa un element de la o poziție dintr-un array:

```
v[index];
```

Totuși, intern se adună la adresa de memorie indexul înmulțit cu dimensiunea tipului de date. Practic linia de mai sus se va traduce în:

```
*(v + index);
```

Observați că nu am mai înmulțit cu dimensiunea tipului de date. Asta deoarece compilatorul de C++ își dă seama automat despre ce tipuri de date este vorba, deci noi nu este nevoie să mai înmulțim cu dimensiunea tipului de date.

Observații:

- Pentru a afla câți bytes ocupă un tip de date se folosește *sizeof*. Exemplu:

```
sizeof(int) // va returna 4 (pentru ca int este stocat pe 4  
// bytes)
```

- Nu putem folosi *sizeof* pentru a afla câți bytes ocupă un obiect alocat dinamic, *sizeof* ne va returna dimensiunea tipului de date pointer, care este 4 pe sistemele pe 32 de biți, respectiv 8 pe sistemele pe 64 de biți.
- Operația de adunare este comutativa, deci $*(v + index)$ este același lucru cu $*(index + v)$, care este același lucru cu $index[v]$. Deci putem folosi și $index[v]$ pentru a accesa elementul de la indexul *index* din array-ul *v*. Nu recomand asta, dar e așa... un fun fact...
- Dacă transforăm pointerul nostru în pointer pentru alt tip de date, atunci, în cazul în care adunăm un număr la pointerul nostru, acesta e posibil să nu fie înmulțit cu dimensiunea pe care o dorim noi, deci trebuie să avem și asta în calcul. Voi exemplifică la ce mă refer în exemplul de mai jos.

Exemplu:

Se citește $n \in N$ de la tastatură, urmat de n numere întregi. Să se memoreze numerele într-un array alocat dinamic și să se afișeze. Pentru citire se vor aduna indicii la adresa array-ului direct, iar pentru afișare se va face același lucru, doar că pointerului ii se va transforma tipul.

```
//EXEMPLUL 9  
#include <iostream>  
  
using namespace std;  
  
int main()  
{  
    int n, *vec;  
  
    cin >> n;  
  
    if (n < 0)  
    {  
        cout << "Numarul citit nu este natural";  
        return 0;  
    }  
  
    vec = new int[n];  
  
    for (int i = 0; i < n; i++)  
        cin >> *(vec + i);  
        // in loc sa facem vec[i] facem  
        // *(vec + i), adica echivalente
```

```

for (int i = 0; i < n; i++)
{
    void *other_ptr = vec;
    other_ptr += i * sizeof(int);
}

cout << *((int*)other_ptr) << " ";
// convertim din nou la (int*)
// pentru a ajuta cout-ul să
// deducă ce vrem să afișăm, iar
// apoi ne uităm să vedem ce se
// află la acea adresă de memorie
// folosind *.

}

delete[] vec;
vec = nullptr;

return 0;
}

```

Tot ce am vorbit până acum despre pointeri e posibil să nu fie în programa de la Programare Orientată pe Obiecte, dar veți avea nevoie să înțelegeți aceste lucruri foarte bine la acest curs pentru a reuși să promovați. De asemenea, informațiile vă vor fi utile și în anul 2 la cursul de Sisteme de Operare.

1.4 Referințe

În C, dacă voi am să folosim un obiect în mai multe locuri din cod, eram obligați să folosim pointeri. În C++ s-a introdus conceptul de referință, care ne scutește din a mai folosi pointeri pentru a referi obiecte.

Pentru a avea o referință către un obiect trebuie să declarăm o variabilă de tipul obiectului pe care vrem să-l referim, și postfixăm numele tipului de date cu &:

```
//EXEMPLUL 10
int& b = a; // b este o referință către a.
```

Exemplu:

Se dă un număr stocat într-o variabilă globală. Să se modifice valoarea ei folosind o referință.

```
//EXEMPLUL 11
#include <iostream>

using namespace std;
```

```

int x = 10; // variabila globala

int main()
{
    int& y = x; // referinta catre variabila
                  // x

    y = 100;    // modificam obiectul referit,
                  // adica x

    cout << x; // se va afisa valoarea
                  // modificata

    return 0;
}

```

Putem folosi referințe către obiecte și când apelăm proceduri. Adică parametrii procedurilor să fie referințe. În acest caz, putem avea o procedură care să modifice valorile argumentelor cu care a fost apelată, iar modificările să se reflecte și în scope-ul în care a fost apelată.

Exemplu:

Să se scrie două proceduri *swap_1*, respectiv *swap_2* care iau ca parametrii două numere întregi și le interschimbă valorile. Una din ele va folosi referințe.

```

//EXEMPLUL 12
#include <iostream>

using namespace std;

int swap_1(int x, int y) // procedura in care
                         // argumentele sunt
                         // copiate
{
    int t = x;           // interschimba
    x = y;              // valorile lui
    y = t;               // x si y.

                         // modificarile
                         // nu se reflecta
                         // si in exteriorul
                         // functiei
}

int swap_2(int& x, int& y) // procedura in care
                           // argumentele
                           // sunt date ca
                           // referinta
{
    int t = x;           // interschimba
    x = y;              // valorile lui
    y = t;               // x si y.

                         // modificarile
                         // se vor reflecata
}

```

```

        // si in exteriorul
        // functiei
}

int main()
{
    int a = 10, b = 20;

    cout << a << " " << b << endl;

    swap_1(a, b);           // apeleaza swap_1
                           // pe ecran vor
                           // aparea valorile
                           // initiale, deoarece
                           // functia a copiat
                           // argumentele cand
                           // s-a facut apelul
                           // deci in functie
                           // au fost modificate
                           // niste copii ale
                           // lui a si b, nu
                           // variabilele
                           // a si b propriu-zise

    cout << a << " " << b << endl;

    swap_2(a, b);           // apeleaza swap_2
                           // pe ecran vor
                           // aparea valorile
                           // interschimbat,
                           // deoarece functia
                           // a folosit referinte
                           // catre obiectele a
                           // si b declarate
                           // aici, in main.
                           // deci functia chiar
                           // a modificat
                           // valorile noastre
                           // din main.

    cout << a << " " << b << endl;

    return 0;
}

```

După ce apelezi funcția swap_1 și observi că nu interschimbă valorile.



Putem avea funcții care ne returnează referințe. Again, comportamentul este același ca în exemplele de mai sus.

Exemplu:

Se dau două variabile globale de tip *int*. Să se scrie o procedură care primește ca parametru un *bool*, și în funcție de acest argument returnează o referință către una din acele două variabile.

```
//EXEMPLUL 13
#include <iostream>

using namespace std;

int x = 10, y = 20;

int& get_ref(bool t)
{
    if (t)
        return x;           // daca argumentul este true
                            // atunci returnam referinta catre x

    return y;             // altfel returnam referinta catre y
```

```

}

int main()
{
    cout << x << " " << y << endl;

    get_ref(true) = 1;           // obtinem o referinta catre x si ii
                                // atribuim valoarea 1. Acum x va avea
                                // valoarea 1, iar y va avea valoarea 20

    cout << x << " " << y << endl;

    int& ref = get_ref(false); // obtinem o referinta catre y.
                                // de data asta pastram referinta
                                // in variabila de referinta ref.
    ref = 2;                  // modificam valoarea referinteii in 2
                                // deci y va avea acum valoarea 2.

    cout << x << " " << y << endl;

    return 0;
}

```

Observații:

- Dacă în exemplul de mai sus am fi facut `int ref = get_ref(false);`; în loc de a folosi o referință, atunci noi în `ref` am fi avut o copie a lui `y`, iar modificând valoarea lui `ref` nu s-ar schimba și valoarea lui `y`.
- Dacă la sfârșitul unei funcții returnăm o referință către un obiect declarat în acea funcție, atunci cel mai probabil ne vom trezi cu un crash cât casa pentru că obiectul respectiv va fi sters automat la sfârșitul funcției (se dă pop la stivă ca la MIPS), deci vom avea o referință către ceva ce nu mai există.

Puteam avea și referințe constante. Adică referințe care mereu o să reflecte valoarea unui obiect, dar pe care noi nu avem dreptul să le modificăm. Pentru a avea o referință constantă, pur și simplu adăugăm cuvântul `const` înainte de a declara o referință:

```
const int& b = a;
```

Aceste tipuri de referințe sunt foarte utile când vrem să transmitem eficient un parametru către o funcție, fără să mai fie nevoie să îl copiem (deci este mai eficient). Faptul că nu ne lasă să modificăm obiectul este o măsură de siguranță, pentru a ne asigura că nu modificăm lucruri pe care nu vrem să le modificăm.

Exemple:

```
//EXEMPLUL 14
#include <iostream>

using namespace std;

int main()
{
    int a = 5;

    const int& b = a;

    b = 10; // eroare de compilare
}
```

```

    cout << a << endl;

    return 0;
}

```

Eu aş recomanda ca atunci când transmiteți obiecte către o funcție, să le transmiteți ca referințe constante dacă știți că nu le veți modifica. Dacă aveți nevoie să le și modificați, atunci recomand să transmiteți obiectele ca referințe.

TLDR: look at this:

<https://dev.to/erraghavkhanna/pass-by-value-reference-explained-with-single-gif-believe-me-it-s-true-23ki>

1.5 Const

In C++, const specifică ca obiectul sau variabila respectivă nu poate fi modificată. Proprietăți:

- O variabilă constantă trebuie initializată pe aceeași linie cu declararea.
- O variabilă constantă nu poate fi modificată.
- `const int * ptr` înseamnă valoarea care se află la adresa de memorie către care pointează `ptr` trebuie să ramane constantă, însă pointerul poate să se schimbe și să pointeze către altceva.
- `int * const ptr` înseamnă că pointerul nu are voie să mai pointeze către altceva, în schimb valoarea către care pointează se poate modifica.
- `const int &ref` și `int const &ref` sunt absolut echivalente și amândouă indic faptul că `ref` este o referință către o variabilă constantă.

Ne vom reaminti aceste proprietăți ale `const`-ului și cand vom cunoaște concepte mai avansate de C++.

1.6 Rvalues și lvalues

Toate obiectele (și prin obiect ma refer la variabile, expresii, numere, caractere, etc.) din C++ sunt împărțite în 2 categorii:

- lvalues (locator values) = obiecte care ocupă o locație identificabilă în memorie (adică care au adresa)
- rvalues = obiecte care nu ocupă o locație identificabilă în memorie

Acestea sunt 2 noțiuni de care ne putem lovi la erorile de compilare, și este bine să stim ce înseamnă ca să putem identifica problema. Spre exemplu:

```

//EXEMPLUL 15
#include <iostream>
using namespace std;

int main()
{
    int var;
    4 = var;      // eroare: expression must be a modifiable lvalue
    (var + 1) = 4; // eroare: expression must be a modifiable lvalue
    return 0;
}

```

```

//EXEMPLUL 16
#include <iostream>
using namespace std;
int foo() {return 2;}

int main()
{
    foo() = 2;      // eroare: lvalue required as left operand of assignment

    // Ce se intampla aici?
    // foo returneaza o valoare temporara, adica un rvalue.
    // La fel ca in exemplul de mai sus, nu putem face assignment unui rvalue.
    return 0;
}

```

2 Incapsularea

Incapsularea este un principiu din programarea orientata pe obiect care se bazeaza pe doua idei principale:

- Toate variabilele si functiile sunt inglobate intr-o singura structura de date si reprezinta caracteristici ale acesteia.
- Accesul la anumiti membri ai structurii poate fi controlat.

2.1 Struct-ul din C

In C, un struct este o structura de date definita de utilizator care ne permite sa combinam elemente de diferite tipuri.

Sintaxa:

```

struct numele_structurii
{
    tip1 data1;
    tip2 data2;
    ..
    tipn datan;
} [una sau mai multe variabile];

```

Exemplu: Vrem sa tinem informatii (titlu, autor, pret, numar volume vandute) despre carti intr-o singura structura. Cum facem asta? Creem o structura:

```

//EXEMPLUL 17
struct carte
{
    char titlu[100];
    char autor[100];
    float pret;
    int nr_volume_vandute;
} carte1, *carte2;

```

Cum accesezi/modific elementele dintr-o structura? Prin operatorul '':

```

carte1.titlu      = "Amintiri din copilarie";
carte1.autor       = "Ion Creanga";
carte1.pret        = 10.25f;
carte1.nr_volume_vandute = 10000;

```

```
(*carte2).titlu      = "Robinson Crusoe";
//Operatorul -> este syntactic sugar pentru (*).
//Adica (*a).b este echivalent cu a->b.
carte2->autor      = "Daniel Defoe";
carte2->pret        = 12.5f;
carte2->nr_volume_vandute = 50000;
```

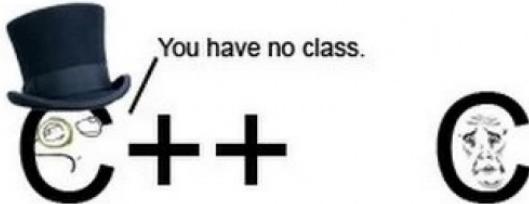
Cum creez un nou element de tip carte?

```
struct carte carte3;
```

Cum creez un pointer la un struct?

```
struct carte *carte4;
```

2.2 Clasele din C++



In C++ se introduce notiunea de clase care vine ca o imbunatatire a notiunii de struct din C. Spre deosebire de **struct**-ul din C, clasele contin atat date cat si functii(denumite acum **metode**). De asemenea, in C++ s-a imbunatatit si notiunea de struct, care permite, la fel ca o clasa, retinerea atat unor date, cat si a unor metode.

Sintaxa unei clase:

```
class numele_clasei
{
[specificator_de_acces]:
    tip1 _data1;
    tip2 _data2;
    ...
    tipn _datan;
[specificator_de_acces]:
    metoda1;
    metoda2;
    ...
    metodam;
};
```

2.3 Specificatori de acces

Pentru a controla accesul la anumiti membri ai unei structuri de date s-au introdus modificatorii de acces. Acestia sunt:

- private = datele si metodele private nu pot fi accesate din afara clasei
- protected = datele si metodele protected nu pot fi accesate din afara clasei, doar din subclase
- public = datele si metodele publice pot fi accesate din afara clasei

Exemplu de declarare a unei clase:

```
//EXEMPLUL 18
#include <iostream>
using namespace std;

class student
{
public:
    void schimba_nume(string nume_nou)
    {
        _nume=nume_nou;
    }
    void afiseaza_cnp()
    {
        cout<<_cnp;
    }
    void afiseaza_nume() { cout << _nume; }

private:
    void schimba_cnp(string cnp_nou)
    {
        _cnp=cnp_nou;
    }

    int      _varsta;
    string   _cnp;
    string   _nume;
};

int main()
{
    student gigel;                      //eroare la compilare:
    gigel._nume="Gigel";                 //'std::__cxx11::string student::_nume'
                                         //is private within this context

    gigel.schimba_nume("Gigel");        //functioneaza, si ii da lui gigel numele Gigel

    cout<<gigel._nume;                  //eroare la compilare:
                                         //std::__cxx11::string student::_nume
                                         //is private within this context

    gigel.afiseaza_nume();              // afiseaza "Gigel"

    gigel.schimba_cnp("0191828763617"); // 'void student::schimba_cnp(std::__cxx11::string)'
                                         //is private within this context

    return 0;
}
```

Diferenta dintre clasele din C++ si structurile din C++ este date de modificatorii de acces impliciti.
Daca scriu clasa:

```
//EXEMPLUL 19
class complex
{
    //fara sa specific un modificador de acces, cel implicit este private
    float _im;
    float _re;
}
```

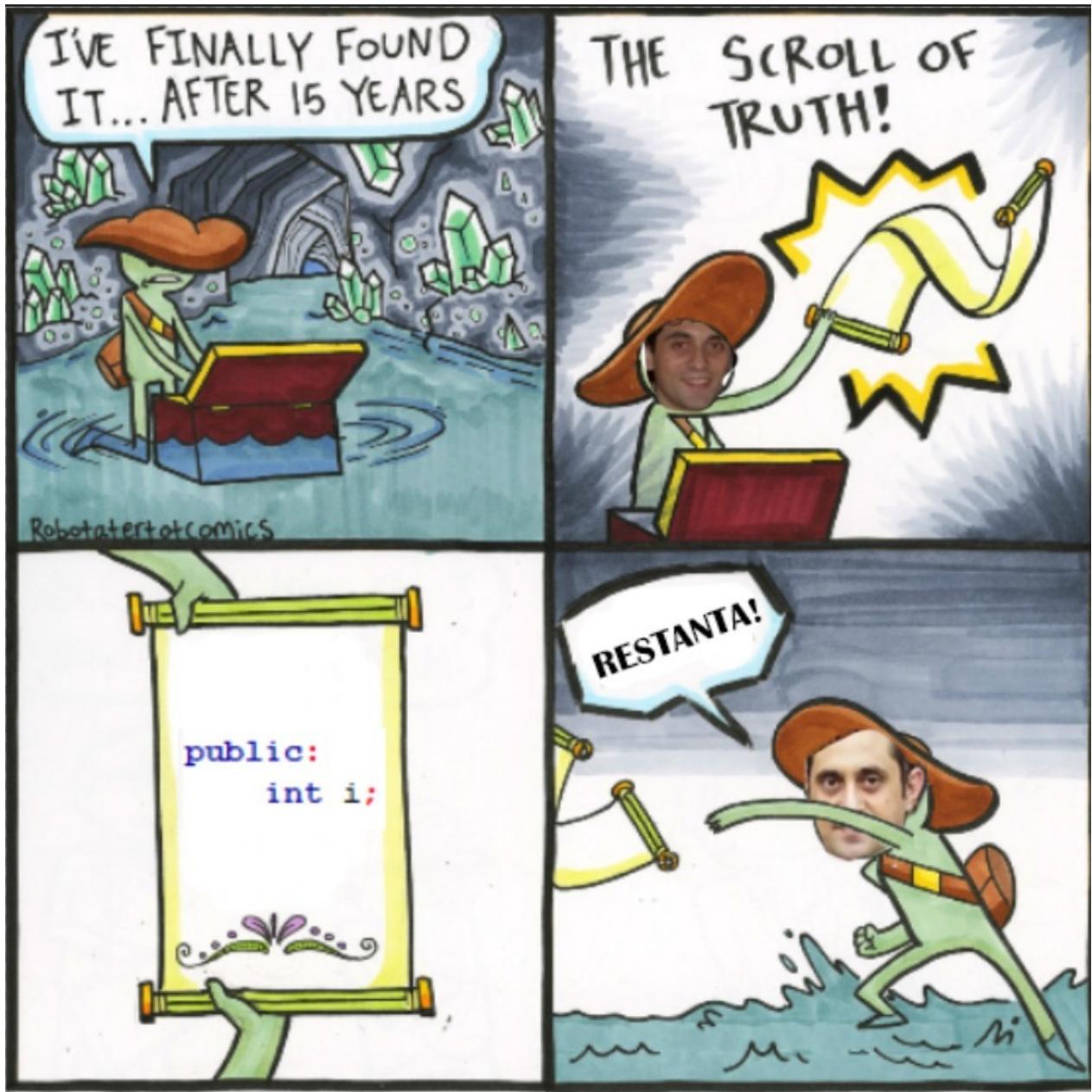
In schimb daca scriu struct-ul din C++:

```
//EXEMPLUL 20
struct complex
{
    //fara sa specific un modificador de acces, cel implicit este public
    float _im;
    float _re;
}
```

Asadar, intr-o clasa, daca modificatorii de acces lipsesc, atunci datele si metodele sunt implicit private. In schimb intr-un struct din C++, daca modificatorii de acces lipsesc, datele si metodele sunt implicit publice.

Nu faceti niciodata datele publice, este automat restanta!

Bun, dar daca toate datele sunt mereu private, cum le accesam?



2.4 Getters si setters

Pentru a putea obtine sau modifica datele dintr-o clasa, avem nevoie de doua tipuri de metode care respectă incapsularea:

- getters = metode publice care întorc valoarea unei date private pentru a putea avea acces la ea în afara clasei
- setters = metode publice care permit modificarea unei date private în afara clasei

Exemplu:

```
//EXEMPLUL 21
#include <iostream>
```

```

#include <string>

using namespace std;

class student
{
public:

    string get_nume() { return _nume; }
    int     get_varsta() { return _varsta; }
    void   set_nume(string new_nume) { _nume = new_nume; }
    void   set_varsta(int new_varsta) { _varsta = new_varsta; }

private:
    string _nume;
    int     _varsta;
};

int main()
{
    student gigel;
    gigel.set_nume("Gigel");
    cout << gigel.get_nume(); //se afiseaza Gigel
    return 0;
}

```

Acum avem o metoda "organizata" si care respecta principiul encapsularii pentru a avea acces sau a modifica datele unui obiect. Dar ce e mai exact un obiect?

2.5 Obiectele

Un obiect este:

- o instantă a unei clase
- o variabilă de tipul unei clase

Cum am vazut și mai devreme, ca să ii dam un nume lui gigel și apoi să afisam datele din gigel avem nevoie de o instantă, un obiect care să apeleze acele metode. Instantele se creează prin constructori.

**Constructor or
something, idk, I'm
not a programmer**



2.6 Constructori

Cand datele sunt publice, putem folosi initializarea agregata pentru a intializa:

```
//EXEMPLUL 22
#include <iostream>
using namespace std;

class foo
{
public:
    int x;
    int y;
};

int main()
{
    foo foo1 = { 4, 5 };      // prin lista de initializare
    foo foo2 { 6, 7 };        // initializare uniforma

    return 0;
}
```

Cum insa noi nu avem voie cu date publice ca ne pica Paun, avem nevoie de constructori.

Constructor = un tip special de metoda care se apeleaza automat atunci cand un obiect este instantiat. Spre deosebire de metodele normale, constructorii au anumite reguli pe care trebuie sa le respectam atunci cand ii scriem:

- Un constructor trebuie sa aiba acelasi nume cu clasa. (conteaza capitalizarea: student ≠ Student)
- Un constructor nu are un tip returnat. (nici macar void)

2.6.1 Constructorul de initializare

Constructor de initializare = o functie care se apeleaza automat la alocare de spatiu pentru un obiect.

```
//EXEMPLUL 23
#include <iostream>
using namespace std;

class complex
{
    float _re;
    float _im;
};

int main()
{
    complex v[2];           //se apeleaza constructorul de initializare de 2 ori
    complex *p;             //nu se apeleaza constructorul de initializare
    p = new complex;         // se apeleaza constructorul de initializare o data
    return 0;
}
```

Constructorii de initializare pot fi:

- cu parametri (parametrizati)
- fara parametri

Exemplu:

```
//EXEMPLUL 24
#include <iostream>
using namespace std;

class lista
{
public:
    lista()           // constructor fara parametri
    {
        _n=0;         // o lista pentru care nu am specificat numarul de parametri este goala
    }
    lista(int nr)    // constructor parametrizat
    {
        int _n=nr;    // creez o lista Cu nr elemente
        _a = new int[nr];
    }
private:
    int* _a;
    int _n;
};

int main()
{
    lista lista1(5);
    return 0;
}
```

2.6.2 Constructorul de copiere

Constructor de copiere (copy constructor, de obicei prescurtat CC) = o functie care se apeleaza cand aloc zona pentru un obiect si ii atribui valoarea unui obiect deja existent.

```
//EXEMPLUL 25
#include <iostream>
using namespace std;

class complex
{
    float _re;
    float _im;
} c1;

void f (complex p) {}

int main()
{
    complex c1;
    complex c2(c1);      // se apeleaza constructorul de copiere
    complex c3=c1;       // se apeleaza operatorul de atribuire (copierea se face bit cu bit)
    f(c1);               // se aloca zona pe stiva pentru un obiect si ii atribui valoarea
```

```

        // unui obiect existent <=> constructor de copiere
    return 0;
}

```

Cum arata un copy constructor?

```
class_name (const class_name &old_obj);
```

De ce arata asa? Haideti sa pornim de la un copy constructor care nu are nici const nici transmitere a parametrului prin referinta:

```

//EXEMPLUL 26
#include<iostream>
using namespace std;

class complex
{
public:
    complex() {};
    complex(float x, float y)
    {
        _re=x;
        _im=y;
    }
    complex (complex old_obj)
    {
        _re=old_obj._re;
        this->_im=old_obj._im;
    }
private:
    float _re;
    float _im;
};

int main()
{
    complex c1(4,5);
    complex c2(c1);           // Se apeleaza constructorul de copiere.
                            // Constructorul de copiere primeste parametrul prin valoare.
                            // Deci este nevoie alocarea pe stiva a unei zone de memorie
                            // pentru obiectul de tip complex si apoi apelarea copy
                            // constructor-ului pentru a se copia valoarea acolo.
                            // => copy constructorul apeleaza copy constructorul
                            // => ciclu infinit (mare crash)
                            // => trebuie sa transmitem prin referinta

    return 0;
}

```

Am aflat deci de ce ne trebuie referinta. Insa de ce ne trebuie si const?

- In primul rand, are sens din punct de vedere logic: daca treaba copy constructorului este doar sa creeze un obiect identic cu cel dat, nu vrem sa il modificam si pe cel dat.
- Vrem sa putem face copii si ale obiectelor const.

```

//EXEMPLUL 27
#include<iostream>

```

```

using namespace std;

class complex
{
public:
    complex() {};
    complex(float x, float y)
    {
        _re=x;
        _im=y;
    }
    complex (complex &old_obj)
    {
        _re=old_obj._re;
        this->_im=old_obj._im;
    }
private:
    float _re;
    float _im;
};

int main()
{
    const complex c1(4,5);
    complex c2(c1);           // binding 'const complex' to reference
                           // of type 'complex&' discards qualifiers
    return 0;
}

```

- Vrem sa putem face copii si dupa obiecte temporare:

```

//EXEMPLUL 28
#include<iostream>
using namespace std;

class Test
{
public:
    Test(Test &t) { /* Copy data members from t */}
    Test()          { /* Initialize data members */ }
};

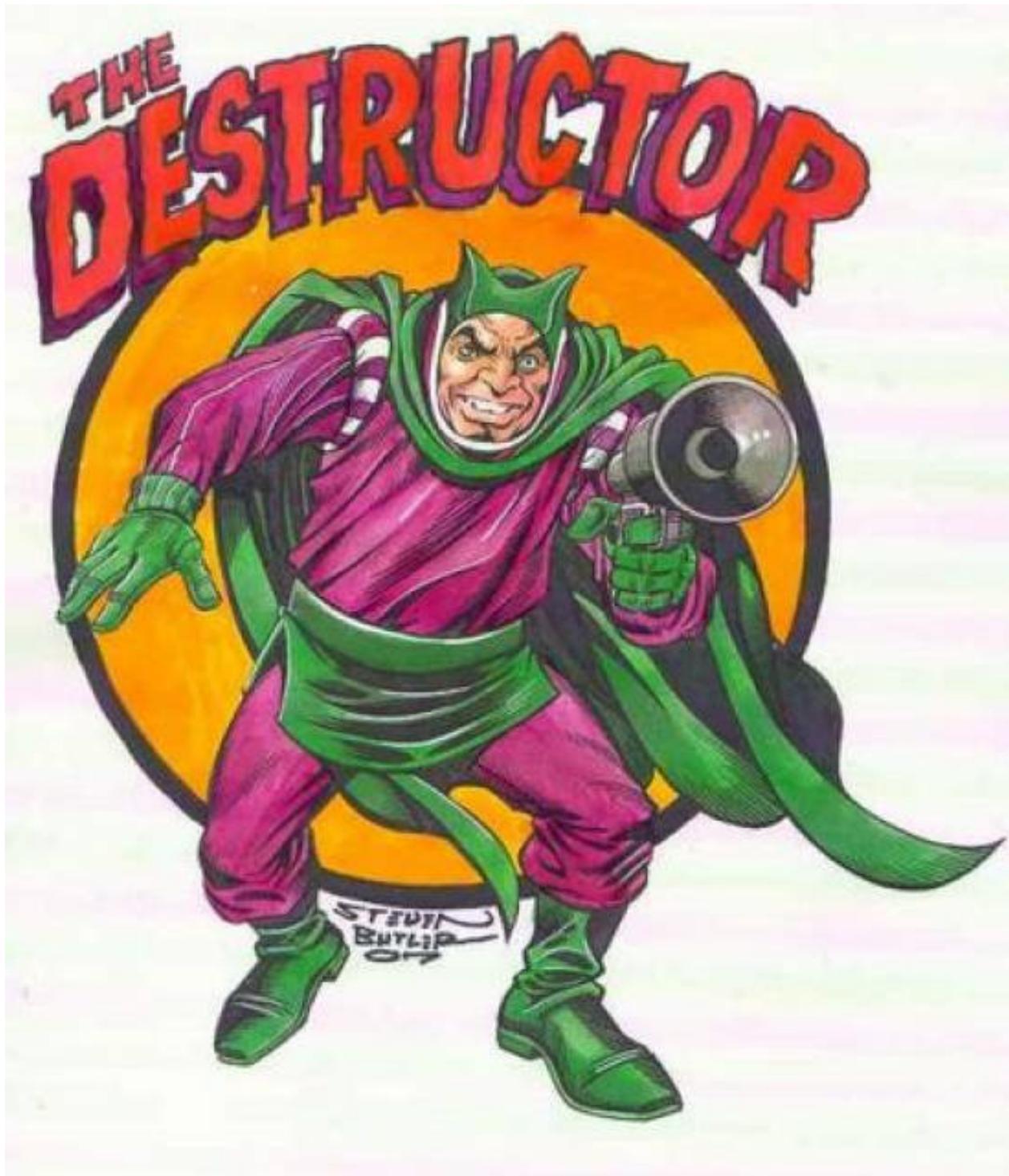
Test fun()
{
    cout << "fun() Called\n";
    Test t;
    return t;
}
private:
/* Class data members */

int main()
{
    Test t1;
    Test t2(fun());      // invalid initialization of non-const
                         // reference of type 'Test&' from an rvalue of type 'Test'
}

```

```
// Reference lifetime extension:  
// Aici fun returneaza o valoare temporara.  
// In C++ putem face bind unei valori temporare  
// doar la un obiect de tipul referinta la const.  
// Deci ne trebuie acel const la copy constructor.  
    return 0;  
}
```

2.6.3 Destructorul



Destructorul este o functie care sterge datele unui obiect. Proprietatile destructorului:

- Acesta este apelat automat atunci cand se distrugе obiectul. Cand se distrugе un obiect? Candiese din scope. Adica:
 - Functia in care a fost creat se termina.
 - Programul se termina.

- Blocul in care a fost creat se termina.
- Se apeleaza un operator de stergere.
- Nu are argumente.
- Nu are niciun tip returnat, nici macar void.
- Nu pot fi declarati static sau const.

Sintaxa:

```
~nume_clasa();
```

Exemplu:

```
//EXEMPLUL 29
class complex
{
public:
~complex()
{
    // Aici destructorul nu trebuie sa faca nimic
    // pentru ca datele din acest obiect nu au fost alocate dinamic.
};

private:
    float _re;
    float _im;
}
```

In schimb aici:

```
//EXEMPLUL 30
#include <iostream>
using namespace std;

class lista
{
public:
    lista()
    {
        _n=0;
    }
    lista(int nr)
    {
        int _n=nr;
        _a = new int[nr];
    }
    ~lista()
    {
        delete[] _a;           // trebuie sa sterg memoria alocata pentru _a,
                               // altfel am memory leak
    }
private:
    int* _a;
    int _n;
};
```

```

int main()
{
    lista lista1(5);
    return 0;
}

```

Destructorul de apeleaza in ordinea inversa a constructorului:

```

//EXEMPLUL 31
#include <iostream>
using namespace std;
class some_class
{
private:
    int _value1;
    char _value2;
};

int main()
{
    some_class object1;           // Se apeleaza constructorul pentru object1.
    some_class object2;           // Se apeleaza constructorul pentru object2.
    some_class object3;           // Se apeleaza constructorul pentru object3.

    return 0;                     // Se apeleaza destructorul pentru object3,
                                  // object2 si object1, in aceasta ordine.
}

```

2.6.4 Constructorii impliciti

In momentul in care scriu o clasa, compilatorul o sa mi defineasca implicit:

- un constructor neparametrizat
- un copy constructor
- un operator =
- un destructor

Acsta este motivul pentru care eu pot scrie programul urmator si acesta sa compileze fara probleme, desi eu nu am definit un constructor de initializare, unul de copiere, un operator = sau un destructor:

```

//EXEMPLUL 32
#include <iostream>
using namespace std;

class some_class
{
private:
    int _value1;
    char _value2;
};

int main()
{
    some_class object1;           // se apeleaza constructorul de initializare implicit
    some_class object2(object1);  // se apeleaza CC-ul implicit
}

```

```

some_class object3=object2;      // se apeleaza operatorul = implicit

return 0;                      // se apeleaza destructorul pentru: object3, object2
                                // si object1 in aceasta ordine
}

```

Constructorul de initializare implicit doar declara datele. Deci asta inseamna ca daca printre datele noastre avem date de tip const sau referinta, suntem obligati sa definim constructorul de initializare, cel implicit nu functioneaza.

```

//EXEMPLUL 33
#include <iostream>
using namespace std;

class some_class
{
private:
    int _value1;
    char _value2;
    const int _value3;
};

int main()
{
    some_class object1;          // note: 'some_class::some_class()' is
                                // implicitly deleted because the
                                // default definition would be ill-formed
    some_class object2(object1);
    some_class object3=object2;

    return 0;
}

```

Constructorul de copiere si operatorul de assignment (=) fac copiere bit cu bit. Cand ne deranjeaza asta? Cand avem pointeri:

```

//EXEMPLUL 34
#include <iostream>
using namespace std;
class some_class
{
public:
    some_class()           { _value3 = new int(7); }
    int get_value3()       { return (*_value3); }
    void set_value3(int x) { *(_value3)=x; }

private:
    int _value1;
    char _value2;
    int* _value3;
};

int main()
{
    some_class object1;        // Se construieste obiectul,
                                // iar object1._value3 = 7.

```

```

cout<<object1.get_value3()<<"\n"; //Se afiseaza 7.

some_class object2(object1); // Se apeleaza copy constructorul
                            // implicit, care face copiere bit cu bit.
                            // Asta inseamna ca pointerul din
                            // object2 va pointa catre aceeasi
                            // adresa ca cel din objec1, nu se va
                            // crea un pointer catre o noua adresa.

cout<<object2.get_value3()<<"\n"; // Se afiseaza 7.

object2.set_value3(5); // Schimbam valoarea aflata la acea
                      // adresa de memorie.
                      // Schimbarea va fi observabila atat in
                      // object2 cat si in object1.

cout<<object1.get_value3()<<"\n"; // Se afiseaza 5.
cout<<object2.get_value3()<<"\n"; // Se afiseaza 5.

some_class object3=object2; // Se apeleaza operatorul = implicit care
                           // face de asemenea copierea bit cu bit.

object3.set_value3(1); // Schimbam valoarea aflata la adresa de
                      // memorie catre care pointeaza toti 3
                      // pointerii. Deci schimbarea se va vedea
                      // in toate cele 3 obiectele.

cout<<object1.get_value3()<<"\n"; // Se afiseaza 1.
cout<<object2.get_value3()<<"\n"; // Se afiseaza 1.
cout<<object3.get_value3()<<"\n"; // Se afiseaza 1.

return 0;
}

```

Atentie! Daca scriem noi constructorul de copiere, pierdem acces la constructorul implicit de initializare. Putem obtine din nou acces la cel implicit daca mentionam = `default`:

```

//EXEMPLUL 35
#include <iostream>
using namespace std;
class some_class
{
public:
    some_class () = default;
    some_class(const some_class& obj) {};
    some_class& operator= (const some_class& obj) {};
private:
    int _value1;
    char _value2;
    int* _value3;
};

int main()
{
    some_class object1;

```

```

    some_class object2(object1);
    some_class object3=object2;
    return 0;
}

```

Care e diferenta dintre cele 2 moduri de apelare a constructorului?

- () foloseste
 - initializarea prin valoare cand parantezele sunt goale
 - initializarea directa cand parantezele nu sunt goale
- foloseste initializarea prin lista, care implica
 - initializare prin valoarea daca parantezele sunt goale
 - initializarea agregata daca obiectul initializat este aggregat

2.7 Lista de initializare (Member initializer lists)

Asa cum am aflat mai sus, unele tipuri de date (precum const-urile si referintele) trebuie sa fie initializate pe aceeasi linie cu declararea. Exemplu:

```
//EXEMPLUL 36
class something
{
public:
    something(int value1, double value2, char value3)
    {
        _value1 = value1; // eroare: nu putem asigna unei variabile constante
        _value2 = value2;
        _value3 = value3;

        //ce facem noi aici pentru _value1 este:
        const int _value1;           // eroare: variabilele constante nu pot fi
                                     // initializate fara valoare
        _value1 = 5;                // eroare: nu pot face assignment la
                                     // variabila constanta
    }
private:
    const int _value1;
    double    _value2;
    char     _value3;
};
```

Si atunci care e solutia? Lista de initializare:

```
//EXEMPLUL 37
class something
{
public:
    something(int value1, double value2, char value3) :
        _value1(value1),
        _value2(value2),
        _value3(value3)
    {
        //nu mai trebuie sa facem nimic aici
```

```

    }
private:
    const int _value1;
    double    _value2;
    char      _value3;
};

```

Atentie! Lista de initializare se poate folosi **DOAR** la constructori.

De ce sa folosim lista de initializare:

- Arata less cluttered si asta este un mare avantaj cand avem de-a face cu clase.
- Eficienta:

```

//EXEMPLUL 38
#include <iostream>
using namespace std;
class example
{
public:
    example()
    {
        cout<<"S-a apelat constructorul fara parametri\n";
    }
    example(int x)
    {
        cout<<"S-a apelat constructorul parametrizat cu parametru "<<x<<"\n";
    }
private:
    int _x;
};

class entity
{
public:
    entity()
    {
        _example = example(8);           // Ca sa asignam lui _example valoarea noului obiect
                                         // creat, este nevoie ca _example sa fie creat.
                                         // => Se apeleaza constructorul fara parametri pentru
                                         // creearea obiectului _example.
                                         // Apoi se apeleaza constructorul parametrizat
                                         // pentru `example(8)`.
                                         // Apoi se face asignarea (assignment).
    }
private:
    example _example;
};

int main()
{
    entity e; // Se afiseaza:
              // S-a apelat constructorul fara parametri
              // S-a apelat constructorul parametrizat cu parametru 8
}

```

```

        return 0;
    }

```

In schimb, daca folosesc lista de initializare, se apeleaza direct constructorul parametrizat:

```

//EXEMPLUL 39
#include <iostream>
using namespace std;
class example
{
public:
    example()
    {
        cout<<"S-a apelat constructorul fara parametri\n";
    }
    example(int x)
    {
        cout<<"S-a apelat constructorul parametrizat cu parametru "<<x<<"\n";
    }
private:
    int _x;
};

class entity
{
public:
    entity() : _example(8) {} // Aici lista de initializare apeleaza direct
                            // constructorul parametrizat pentru _example,
                            // cu parametrul 8.
private:
    example _example;
};

int main()
{
    entity e; // Se afiseaza doar:
              // S-a apelat constructorul parametrizat cu parametru 8
    return 0;
}

```

- Avem cazuri cand suntem obligati sa folosim lista de initializare:

- Cand avem **date constante** in clasa noastra (nu statice constante tho - voi nu stiti ce e aia deocamdata dar sa fie aici pentru cand va uitati inainte de examen).
Noi stim ca variabilele constante trebuie initialize pe aceeasi linie cu declararea. => ne trebuie lista de initializare.
Vezi exemplele 36 si 37.

- Cand avem **referinte** in clasa noastra.

```

//EXEMPLUL 40
class something
{
public:
    // something(int value1)           // Daca scriu asa primesc eroarea:
    // {                                // uninitialized reference member
        //     _value1=value1;           // in 'int&' [-fpermissive]
    }
}

```

```

    // }
    something(int value1) : _value1(value1) {}      // Daca scriu asa
                                                    // initializeaza corect
private:
    int& _value1;
};
int main()
{
    return 0;
}

```

2.7.1 Ordinea in care se initializeaza datele

Surprinzator, ordinea in care se initializeaza datele din lista de initializare nu este ordinea specificata in lista de initializare, ci este ordinea in care datele respective au fost declarate in clasa. Soo, ca sa nu va incurcati, e bine sa scrieti variabilele in lista de initializare in aceeasi ordine in care ati scris variabilele in clasa.

3 Compozitia

Compozitia este un concept fundamental al programarii orientate pe obiect care modeleaza relatii de timpu "has-a" intre obiecte. Ce inseamna asta? Inseamna ca daca avem clasa actor si clasa film, putem spune ca un film are actori.

```

#EXEMPLUL 41
#include <iostream>
using namespace std;

class point2D
{
public:
    point2D() : _x(0), _y(0) {}
    point2D(int x, int y) : _x(x), _y(y) {}
    void set_point2D (int x, int y)
    {
        _x = x;
        _y = y;
    }
private:
    int _x;
    int _y;
};

class creature
{
public:
    creature (const string& name, const point2D & location) :
        _name(name),
        _location(location)
    {
    }

    void move_to(int x, int y)
    {
        _location.set_point2D(x, y);
    }
}

```

```

        }
private:
    string _name;
    point2D _location;
};

int main()
{
    //creature creature1("troll", (4, 7));      // Aici nu stie ca 4 si 7 sunt pentru
                                                // initializarea punctului.

    point2D point1(4,7);                      // Dar pot sa initializez asa.
    creature creature1("troll", point1);

    creature creature2{"bear", {1,1}};          // Sau asa.
    return 0;
}

```

4 Exercitii

Toate exercitiile au enuntul: "Functioneaza? Daca da, ce afiseaza, daca nu, de ce nu si modificati o linie astfel incat sa mearga." Sunt genul de exercitii pe care o sa le primiti la examen.

Exercitiul 1

```

1 #include <iostream>
2 using namespace std;
3 class cls
4 {
5 public:
6     cls()           { x=3;           }
7     void f(cls &c) { cout << c.x; }
8     int x;
9 };
10 int main()
11 {
12     cls d;
13     f(d);
14     return 0;
15 }

```

Exercitiul 2

```

1 #include<iostream>
2 using namespace std;
3 class cls
4 {
5 public:
6     cls(int i=0, int j=0) { x=i; y=j; }
7     int x;
8     int y;
9 };
10 int main()

```

```

11  {
12      cls a, b, c[3]={cls(1,1), cls(2,2), a};
13      cout << c[2].x;
14      return 0;
15  }

```

Exercitiul 3

```

1 #include <iostream>
2 using namespace std;
3 class cls
4 {
5 public:
6     int x, y;
7     cls(int i=2, int j=3)
8     {
9         x=i+j/2;
10        y=i-j/2;
11    }
12 };
13 int main()
14 {
15     cls a, b, c=a;
16     cout << a.x;
17     return
18 }

```

Exercitiul 4

```

1 #include <iostream>
2 using namespace std;
3 class Test
4 {
5 public:
6     Test();
7 };
8 Test::Test()
9 {
10     cout<<"Constructor Called \n";
11 }
12 int main()
13 {
14     cout<<"Start \n";
15     Test t1();
16     cout<<"End \n";
17     return 0;
18 }

```

Exercitiul 5

```
1 #include <iostream>
2 using namespace std;
3 class Test
4 {
5 public:
6     Test();
7 };
8 Test::Test()
9 {
10     cout<<"Constructor Called \n";
11 }
12 int main()
13 {
14     cout<<"Start \n";
15     Test t1();
16     cout<<"End \n";
17     return 0;
18 }
```

Exercitiul 6

```
1 #include <iostream>
2 using namespace std;
3 class Point
4 {
5 private:
6     int x;
7     int y;
8 public:
9     Point(int i, int j);
10 };
11 Point::Point(int i = 0, int j = 0)
12 {
13     x = i;
14     y = j;
15     cout << "Constructor called";
16 }
17 int main()
18 {
19     Point t1, *t2;
20     return 0;
21 }
```

Exercitiul 7

```
1 #include <iostream>
2 using namespace std;
3 class Test
4 {
5     int value;
```

```

6  public:
7      Test(int v);
8  };
9 Test::Test(int v)
10 {
11     value = v;
12 }
13 int main()
14 {
15     Test t[100];
16     return 0;
17 }
```

Exercitiul 8

```

1 #include <iostream>
2 using namespace std;
3 class Test
4 {
5     int &t;
6 public:
7     Test (int &x) { t = x; }
8     int getT() { return t; }
9 };
10 int main()
11 {
12     int x = 20;
13     Test t1(x);
14     cout << t1.getT() << " ";
15     x = 30;
16     cout << t1.getT() << endl;
17     return 0;
18 }
```

Cum rezolvam?

Modific linia 7 astfel: `Test (int &x) : t(x) {}.` Acum programul ruleaza si afiseaza 20 30.

Exercitiul 9

```

1 #include <iostream>
2 using namespace std;
3 class Fraction
4 {
5 private:
6     int den;
7     int num;
8 public:
9     void print() { cout << num << "/" << den; }
10    Fraction() { num = 1; den = 1; }
11    int &Den() { return den; }
12    int &Num() { return num; }
13 };
14 int main()
```

```

15  {
16      Fraction f1;
17      f1.Num() = 7;
18      f1.Den() = 9;
19      f1.print();
20      return 0;
21  }

```

Exercitiul 10

```

1 #include <iostream>
2 using namespace std;
3 class Test
4 {
5     private:
6         int x;
7     public:
8         void setX (int x) { Test::x = x; }
9         void print() { cout << "x = " << x << endl; }
10    };
11 int main()
12 {
13     Test obj;
14     int x = 40;
15     obj.setX(x);
16     obj.print();
17     return 0;
18 }

```

Exercitiul 11

```

1 #include <iostream>
2 using namespace std;
3 class A
4 {
5     int id;
6     public:
7     A (int i) { id = i; }
8     void print () { cout << id << endl; }
9 };
10 int main()
11 {
12     A a[2];
13     a[0].print();
14     a[1].print();
15     return 0;
16 }

```

Nu functioneaza, din acelasi motiv ca la Exercitiul 7, nu are constructor fara parametri. Putem fie sa adaugam un constructor fara parametri, fie sa adaugam o valoare implicita pentru parametrul i din constructorul parametrizat.

Daca alegem prima varianta, adica sa adaug linia `A () {}`, programul va afisa 2 valori garbage (diferite in

functie de compilator, si momentul de timp):

3280896
4194432

Daca in schimb adaug parametru default, adica sa modific linia 7 astfel:

A (int i=0) id = i;

Programul va afisa 2 de 0:

0
0

In general, si la examen, daca vedeti 2 solutii posibile de rezolvare, scrieti-le pe amandoua, discutati care e mai buna, etc. Arata ca aveti o intelegerere de ansamblu asupra problemei si asta ajuta mult intr-un examen.

5 Resurse

- un site super bun cu concepte de OOP cand aveti nevoie sa va clarificati o notiune: learncpp.com
- this guy care are un playlist de C++, la fel, pentru cand vreti sa va clarificati o notiune anume: [Cherno](#)
- this guy as well: [CppNuts](#)
- cartea asta pe care v-a dat-o si la curs, e utila daca nu reusiti sa fiti atenti nici la curs nici la tutoriat: [Thinking in C++ de Bruce Eckel](#)
- [Acum video care explică foarte bine conceptele de programare orientată pe obiecte în C++](#)