Programare Orientata pe Obiecte

- POO -

Titular curs : conf. dr. ing. Andreea Udrea

Contact : andreea.udrea@upb.ro

Birou: ED 009

Organizare:

Curs – 3h/sapt. ; **Laborator** – 2h/sapt. (Platforma https://ocw.cs.pub.ro/courses/poo-is)

Nota finala:

Laborator **- 30/110pcte** - media notelor obtinute la fiecare laborator (cerinta laborator va fi postata dupa curs, cu termen de rezolvare pana vinerea 23:59)

* O cerinta de laborator nepredata la timp = 0 (nu se va reface in alta saptamana)

Teme de semestru: - 2*15= **30/110pcte**

*Deadline: Tema 1 – saptamana 9 Tema 2 – saptamana 13

**Enunturile pentru teme le veti primi cu min 3 saptamani inainte de deadline

Activitati curs - 2x5%=**10/110pcte** - grile 15-20 intrebari (saptamanile 9, 14)

•Pentru a se intra in examenul final, sunt necesare 30 din 70 de puncte aferente activitatilor de semestru.

Examen **- 40/110pcte** - 2 probleme (2*2pcte)

•Pentru promovare trebuie sa se obtina minim 20 pcte la examenul final.

IMPORTANT: O cerinta de laborator/tema de semestru copiata = (-1)*punctajul ei

Pentru test antiplagiat se va folosi programul Moss.

- Tema 1 se va baza pe o parte din materia pe care ati facut-o la SDA (dar cu implementare orientata pe obiect).

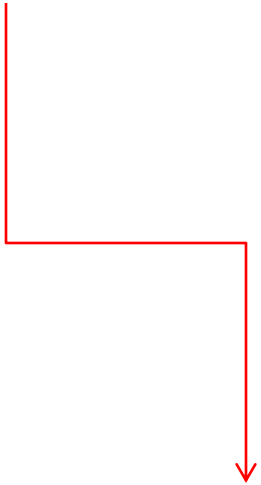
- Materialele necesare legate de structuri de date liniare (liste, cozi, stive) si arbori (BST) vor fi disponibile pe Moodle.

❖ Cursurile din saptamanile 9 si 14 (cele cu predarea Temelor de semestru) as vrea sa fie recuperate in avans cel mai probabil pana in saptamana 7 – zi sugerata -vineri).

- Examen final – la inceputul sesiunii.

Necesar:

Cunostinte medii de C si structuri de date si algoritmi



Abilitati la absolvirea cursului:

Modificare si testare programe C++

Concepere si implementare aplicatii C++

Modificare si testare programe Java

POO?

- Paradigma de programare (model de limbaj de programare) organizat in jurul obiectelor mai repede decat in jurul actiunilor; in jurul datelor mai repede decat in jurul procedurilor.

*un program in viziunea voastra de pana acum …

procedura logica ce primeste date de intrare, le proceseaza si produce date de iesire.

Paradigme de programare

- procedurala/imperativa (calculul presupune instructiuni ce modifica starea unui program – ex: C, C++, Java)
- orientata pe obiecte (SmallTalk, C++, Java, C#)
- declarativa (calculul se face fara a defini fluxul de control in amanunt, ex: SQL)
- functionala (calculul presupune evaluarea de functii matematice fara a se pune accent pe stare – ex: Haskell, Lisp, Python).

Exista limbaje realizate astfel incat sa suporte

- doar o paradigma de programare: SmallTalk , Haskell
- mai multe paradigme: C++, Java, C#, Visual Basic, Perl, Python, Ruby

De ce POO?

- probleme mari ->subprobleme (rezolvare separata) ->module individuale inteligibile-> **modularizare**

- terminologia problemei se regaseste in solutia software/modulele sunt usor de inteles -> **abstractizare/ aplicatii usor inteligibile**

- functionalitatile de nivel scazut sunt protejate prin -> **incapsulare**

- interfetele permit combinarea modulelor -> sisteme noi -> **compozabilitate-design structurat**

- dezvoltare de la programe mici si simple -> module complexe ->**ierarhizare**

- mentenanta si modificarile unor module nu afecteaza ierarhia-> **continuitate**

Unde?

Raspandire larga, necesar in multe domenii (aplicatii web, mobile, desktop; aplicatii industriale, medicale, uz curent; jocuri…)-> **Angajare**

De ce C++?

1. C++ - > 1979 si ramane limbajul cu cea mai buna performanta/cost
("performance/$")

Performanta – eficienta in termeni de timp de executie, memorie ocupata; interactiune harware (mobile, desktop, data center); dezvoltare facila de aplicatii

->1989-1999 - C/C++ - "mainstream"
- standardizare ISO ('98), imbunatatire compilatoare

->1999-2009 - decada "productivity vs. performance"
- .net, C#, Visual C++, Java
- s-au dovedit a nu fi la fel de robuste (esec Windows Vista)

-> 2009 - … - revenire la interesul pentru "performance/$ " si robustete

2. Conceptele de POO sunt cel mai bine exprimate in C++ (dpdv educativ)
Daca se cunoaste C++ atunci se poate trece usor la Java/C#, programare vizuala, programare dispozitive mobile…etc
! Angajare

Cuprins:

- Limbaje interpretate si limbaje compilate
- Cuvinte rezervate; variabile; managementul memoriei
- Asemanari si deosebiri C/C++
- Introducere in conceptele POO: Clase; Obiecte; Constructori; Destructori
- Supradefinirea operatorilor si functii friend; Sistemul de intrari/iesiri
- Agregare si derivare; mostenire; supraincarcarea functiilor
- Functii si clase virtuale; polimorfism
- Functii si clase Template/Sablon
- Functii si clase abstracte
- STL
- Paralela Java – C++

Azi

- Limbaje interpretate si limbaje compilate
- Compilarea in C/C++
- Variabile si cuvinte cheie
- Managementul memoriei

Procesul de “traducere”

limbajele de programare - traduceri (scop – rezolva o **problema**)

programator

ceva usor de inteles de catre om: **cod sursa**

translator

ceva ce poate sa fie executat de calculator: **instructiuni masina**

calculator

Translatoarele pot fi:

- *interpretoare (=>lb. interpretate)*
- *compilatoare (=>lb. compilate)*

Q: *C?*

Interpretor

•traduce **codul sursa** -> **activitati** (ex. grupuri de instructiuni masina)

•le executa imediat (interpretare <-> executie)

Lb. interpretate: BASIC, PHP ,etc.

•traduce si executa linie dupa linie codul sursa =>

=> retraduce orice instructiune chiar daca aceasta a fost tradusa anterior =>

=> **LENT**

•BASIC a fost compilat din motive de viteza.

Avantaje :

•tranzitia: scriere cod -> executie - imediata

•codul sursa este mereu **disponibil**; interpretorul poate fi extrem de specific cand intalneste o eroare

•usurinta in interactiune si dezvoltare rapida a programelor (nu neaparat executie)

•independenta de platforma

Dezavantaje :

•interpretorul (sau o versiune redusa) **trebuie sa se gaseasca in memorie** pentru a executa codul

•pana si cel mai rapid interpretor poate introduce **restrictii de viteza** inacceptabile

•majoritatea interpretoarelor presupun ca intreagul cod sursa sa fie introdus in interpretor => **limitari de spatiu**

•in general, apar **limitari** in cazul realizarii **proiectelor de dimensiuni mari**

Compilator

•traduce **codul sursa** -> **limbaj de asamblare** sau **instructiuni masina**

•produsul final -> fisier ce contine cod masina

•realizeaza un proces cu mai multi pasi; tranzitia cod scris -> cod executat - semnificativ mai lunga

•programele generate necesita **mai putin spatiu pentru a rula** si **ruleaza mai repede**

Limbaje compilate : C, C++, C#, Fortran, Pascal, etc

Unele permit compilarea independenta a diferitelor bucati de cod.

Aceste bucati sunt eventual combinate intr-un program final executabil de catre o unealta numita *linker* (editor de legaturi).

Acest proces se numeste *compilare separata*.

Avantaje:

•daca un program depaseste limitele compilatorului sau mediului de compilare -> poate fi impartit in mai multe fragmente

•programele pot fi construite si testate fragment cu fragment

•cand un fragment functioneaza, poate fi salvat si folosit in continuare

•colectiile de astfel de fragmente testate pot fi integrate in biblioteci *(libraries)* ce pot fi folosite de alti programatori

•crearea unei piese -> ascunde complexitatea pieselor componente

•aceste elemente faciliteaza crearea de aplicatii de dimensiuni mari

•elementele de debugging oferite de compilator au fost semnificativ imbunatatite de-a lungul timpului; compilatoarele vechi generau doar cod masina, debuggingul era facut de compilator cu functii de tip print

Limbaje de nivel scazut – in general compilate

eficienta la nivel de memorie si durata de executie

VS

Limbaje de nivel inalt – in general interpretate

dezvoltare facila, independenta de platforma

- in ultima perioada cele doua abordari au fost combinate pentru a oferi cat mai multe avantaje

- ex. Python, Java: codul sursa -> limbaj intermediar -> executat de un interpretor mult mai rapid

C si C++ : Compilarea

I. rularea unui *preprocesor* pe codul sursa

Preprocesorul - un program simplu

- inlocuieste bucati de cod sablon/tipar din codul sursa cu alte sabloane definite de programator (*directive preprocesor*)

Directivele preprocesor (ex. #define MAX(x,y) ((x)>(y)?(x):(y))) sunt folosite pentru a scadea volumul de cod scris si a face codul usor de citit (code readability).

❖ Codul pre-procesat este, de obicei, scris intr-un fisier intermediar.

!!! Designul lb. C++ a avut in vedere descurajarea utilizarii excesive a preprocesorului, deoarece poate introduce erori subtile.

II. Compilatoarele lucreaza de obicei in 2 pasi

Primul pas: parsarea codului preprocesat (codul sursa este spart in mai multe unitati mici pe care compilatorul le organizeaza intr-o structura numita arbore)

Ex: in expresia "**A + B**" elementele '**A**', '**+,**' si '**B**' sunt noduri ale arborelui de parsare.

Al doilea pas: generarea de cod (*code generator)* se parcurge arborele si genereaza fie cod in limbaj de asamblare pentru fiecare nod al arborelui.

Rezultatul final este un **modul obiect** (in general, un fisier cu extensia **.o** sau **.obj**).

OBS !!! Notiunea de modul obiect (MO) premerge pe cea de POO.

Aici "obiect" este echivalent cu obiectiv/ "goal" in termeni de compilare.

Editorul de legaturi (*linker)*

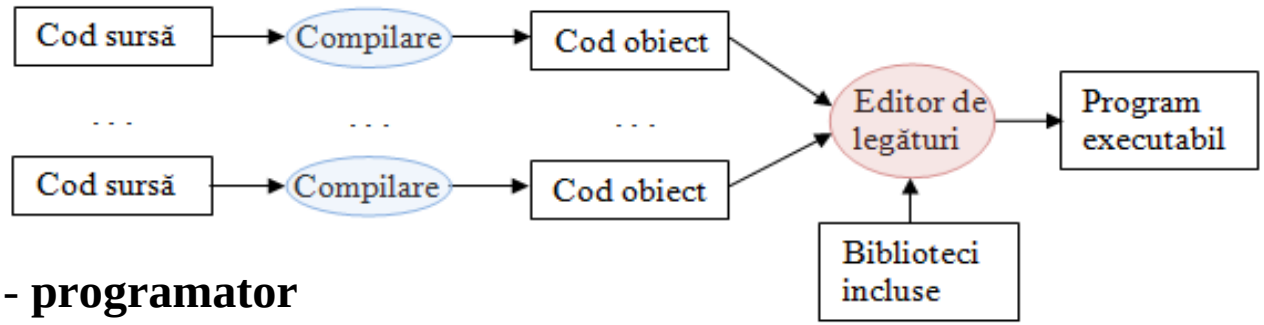
•combina o lista de MO intr-un program executabil care poate fi incarcat si rulat de sistemul de operare

•rezolva o referire facuta de o functie sau variabila dintr-un MO catre o functie sau variabila din alt MO (se asigura ca toate functiile si datele externe solicitate exista pe durata compilarii)

•adauga un modul obiect special pentru activitatile de start-up/pornire (initializare date in C)

•cauta in biblioteci pentru a rezolva referintele de tip #include

De la codul sursa la program executabil



- **Etape**
  - **editarea** codului sursa - **programator**
    - salvarea fisierului cu extensia *.c /.cpp*
  - **preprocesarea - preprocesor**
    - efectuarea directivelor de preprocesare (#define MAX 100)
    - ca un editor – sunt inlocuite directivele preprocesor in codul sursa
  - **compilarea (C - limbaj compilat) - compilator**
    - verificarea sintaxei
    - transformare in modul/cod obiect (limbaj masina) cu extensia *.o /.obj*
  - **editarea legaturilor - editor de legaturi (linker)**
    - combinarea modulului obiect cu alte module obiect (ex: al bibliotecilor asociate fisierelor header incluse)
    - transformarea adreselor simbolice in adrese reale

C si C++: Variabile, tipuri de date si cuvinte rezervate

date intrare -> **program** -> date iesire

variabile + **operatori** -> expresii + **instructiuni** -> descriu pasii unui **program**

Variabilele

- ocupa un anumit spatiul in memoria calculatorului
- fiecare are un nume convenabil in codul sursa (ex. rezultat, suma, nume, vector)
- in momentul rularii/executiei (runtime) fiecare variabila foloseste pentru un anumit timp o zona/ spatiu din memorie ca sa isi stocheze valoarea

❖ *Cand folosim termenul **"a aloca" (allocate)** – indicam ca unei variabile i se pune la dispozitie un spatiu de memorie pentru a isi stoca valoarea.*

❖ *Spatiul ocupat de o variabila este eliberat (**deallocated**) cand sistemul ii revendica spatiul de memorie si, deci, ea nu mai are o zona unde sa isi stocheze valoarea.*

❖ *Durata de la momentul alocarii la momentul eliberarii spatiului de memorie se numeste **durata de viata (lifetime)**.*

Definirea variabilelor - diferente intre C si C++

Asemanari: - trebuie definite inainte de a fi utilizate.

Deosebiri: - in **C** - definite la inceputul sectiunii in care vor fi vizibile pentru ca un bloc intreg de memorie a fie asignat de catre compilator

- in C++ - definite oriunde in zona de vizibilitate (pe larg la Cursul 2)

La **crearea** unei variabile trebuie specificate o serie de optiuni:

- durata de viata (globala/locala)
- cum i se aloca spatiu (static/dinamic)
- cum este tratata de compilator (cuvinte cheie: extern, static, const si combinatii intre acestea)
- ce tip are

Variabilele pot sa fie : de un tip de date de baza

de un tip de date derivat / definit de utilizator

Tipuri de date de baza (fundamentale)

- **int** – numere intregi
- **float** – numere reale reprezentate in virgula mobila (floating point)
- **double** – numere reale mari in virgula mobila (de doua ori mai mult spatiu rezervat ca in cazul tipului float)
- **char** –defineste caractere
- **void** - tip special - multimea valorilor e vida; specifica absenta oricarei valori

Modificatori de tip (schimba proprietatile tipului de date curent – cresc/scad spatiul ocupat)

- **short**
- **long**
- **signed (default)**
- **unsigned**

short int < int < long int
float < double < long double

Pentru a afla cat spatiu are alocat o variabila – se foloseste operatorul **sizeof**.

Spatiul ocupat de o variabila poate sa difere de la un sistem de operare/compilator la altul.

Tipuri de date derivate /definite de utilizator

- Tablouri/Vectori/Masive /Arrays
- Structuri, uniuni, enumeratii
- Pointeri – Anexa I

Tablouri – stocheaza mai multe elemente de acelasi tip folosind o zona continuua de memorie

Unidimensionale: tip_date **nume**[dimensiune];

Initializare:

```
int v[5] = {19, 10, 8, 17, 9};
int v[] = {19, 10, 8, 17, 9};
```

Ce se intampla daca incerc sa fac atribuirea: v[5]=5; ?

Obs: La accesarea v[5] – nu o sa am eroare de compilare! -> Eroare logica

Multidimensionale: tip_date **nume**[dimensiune_1]…[dimensiune_n];

Initializare:

```
int m[2][3] = {{1, 3, 0}, {-1, 5, 9}};
int m[][3] = {{1, 3, 0}, {-1, 5, 9}};
int m[2][3] = {1, 3, 0, -1, 5, 9};
```


OBS: v si m nu isi modifica lungimea sau adresa de inceput pe toata durata executiei programului.

Ex: – tablou tridimensional – citire /afisare valori

```
#include <stdio.h>
int main() {
int i, j, k, test[2][3][2];

printf(“Dati valorile: \n");
for(i = 0; i < 2; ++i) {
    for (j = 0; j < 3; ++j) {
            for(k = 0; k < 2; ++k ) {
                    scanf("%d", &test[i][j][k]); }
    }
}

// Afisare valori:
for(i = 0; i < 2; ++i) {
     for (j = 0; j < 3; ++j) {
              for(k = 0; k < 2; ++k ) {
                      printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]); }
      }
}
return 0;
}
```

Tipuri de date derivate

Structura – spre deosebire de tablou, stocheaza elemente(campuri) de tipuri diferite, dar tot in locatii consecutive de memorie

```
struct persoana{
      char nume[50];
      int varsta;
 } p;

// main
struct persoana p1, p2, p3[20];
```

```
/*sau*/
typedef struct /*pers*/{
    char nume[50];
    int varsta;
} persoana;

//main
persoana p1, p2, p3[20];
```

```
/*sau*/
typedef struct  pers   persoana;
struct pers{
    char nume[50];
    int varsta;
};

//main
persoana p1, p2, p3[20];
```

Variabile globale

- definite in afara oricarei functii
- accesibile de oriunde din program
- durata de viata este egala cu cea a programului
- daca existenta unei variabile dintr-un fisier este declarata folosind cuvantul **extern** in alt fisier - > poate fi folosita in cel de-al doilea fisier

extern - spune compilatorului ca variabila **globala** exista in alt fisier decat cel compilat.

```
//f1.cpp
#include <cstdlib>
#include <iostream>
using namespace std;

int var_e=10;
void funct(); //definire

int main( )
{
   cout<<var_e<<endl; //10
   funct();                  //11

   return 0;
}
```

```
//f2.cpp
#include <iostream>
using namespace std;

extern int var_e;

void funct()//implementare
{
    cout<<++var_e;
}
```

Spatiul de memorie pentru var_e e creat in urma definirii in **f1.cpp**, aceasi variabila este accesata in **f2.cpp**.
Codul din **f2.cpp** e compilat separat de cel din **f1.cpp**, deci compilatorul trebuie informat ca variabila exista in alta parte prin declaratia :

extern int var_e;

Variabile locale (automate)

- definite in interiorul unei functii
- accesibile de oriunde din functia respectiva
- durata de viata este egala cu cea a functiei

```
//exemplu
int patrat(int numar)
{
int rezultat;          //variabila locala
rezultat=numar*numar;
return rezultat;
}
```

- sunt numite locale pentru ca sunt vizivile doar in functie, iar durata lor de viata este legata de functia in care sunt declarate.

- de fiecare data cand functia este **executata**, variabilele sale locale sunt **alocate**.

- la iesirea din functie (finalul executiei codului aferent; return/exit) variabilele locale elibereaza spatiul ocupat.

❖ **parametrul numar** din functia patrat este tot o **entitate locala** si primeste un spatiu de memorie la apelul functiei

❖ diferenta intre **numar** si **rezultat** este ca **numar** va avea initial valoarea copiata in urma apelului functiei:

int x= **patrat(3)**;

```
int patrat(int numar)
{
int rezultat;
rezultat=numar*numar;
return rezultat;
}
```

in timp ce, in acest caz, **rezultat** va avea o valoare aleatoare.

❖ Variabilele locale sunt disponibile/accesibile(scoped) doar in functia din care fac parte : {}

Copii locale

- **Parametrii locali** sunt copii locale ale informatiei cu care se apeleaza functia.
- Acest mecanism se numeste **transmitere prin valoare (pass by value).**

Parametrii sunt variabile locale initializate cu o operatie de tip atribuire la apelul functiei.

Avantaj : Functia va detine propria copie - pe care poate sa o manipuleze fara sa interfere cu variabila din care se face copierea.

Principiul se numeste independenta; separarea componentelor pe cat de mult posibil fiind extrem de importanta.

```
void f(int i, int j){
  i++; j--;
  cout<<i<<" "<<j;
}
```

Dezavantaj : nu exista nici un mecanism prin care modificarile facute in functie sa fie vizibile (in afara de return, si poate exista mai mult de un parametru per apel de functie). (Vom discuta despre transmiterea prin referinta – Cursul 2)

de asemenea, copiile sunt "scumpe" in termeni de memorie.

```
int main(){
  int x=3; y=3;
  f(x,y);
  cout<<x<<" "<<y;
}
```

ERORI posibile: O functie returneaza un pointer catre o variabila locala

```
int * pointer_local()
{
        int temp=10;
        return(&temp); //returneaza un pointer catre variabila locala de tip int
}

void main()
{
        int * ptr=pointer_local();
        ….
        (*ptr)=(*ptr)+1;  //?
}
```

- Functia este in regula pe durata rularii, problema apare in momentul in care se iese din functie deoarece se returneaza un pointer la un int – dar unde este acest int alocat?
- **Problema:** variabila locala temp este alocata doar cat timp functia este executata; la iesirea din functie, spatiul este eliberat automat.
- Cand rulam aceasta mica bucata de cod, eroarea nu este vizibila, dar daca aplicatia se complica, sistemul va face solicitari asupra memoriei ocupate de temp.

Daca totusi vrem ca functia de mai sus sa intoarca un pointer la o variabila locala – ne asiguram ca acea variabila nu va disparea la sfarsitul functiei -> trebuie declarata **static**.

static= spune compilatorului ca o variabila, odata alocata, va ramane alocata cat timp programul este executat

```
int * pointer_local ()
{
      static int temp=10;   //valoarea initiala(la primul apel) este 10
      return(&temp);        //returneaza un pointer catre variabila locala de tip int
}
```

Q: De ce nu s-ar folosi in acest caz variabile globale?

A: Pentru ca acestea ar fi vizibile oriunde si ar putea sa fie modificate incorect, in afara functiei.

Cuvantul cheie **static** (mai multe sensuri)

I. o variabila locala definita static exista pe toata durata programului

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

void func() {
static int i = 0;
i++;
cout << "i = " << i << endl;
}

int main( )
{
   for (int j=0; j<5; j++)
                func();

    return 0;
}
```

Daca variabila i nu era declarata static se afisa:
i=1
i=1
i=1
i=1
i=1

Afisaza:
i=1
i=2
i=3
i=4
i=5

II. O *functie* sau *variabila* globala definita folosind cuvantul cheie static semnalizeaza ca acel nume e rezervat si nu este accesibil/utilizabil in afara fisierului in care apare (*file scope – vizibilitate la nivel de fisier)*

EX: Eroare de legatura (linker error)

```
//f1.cpp
// file scope = doar in acest fisier
static int var_stat;
void func();
int main(){
var_stat = 1;
func();…
}
```

```
//f2.cpp
extern int var_stat;  //EROARE LINKER

void func() {
var_stat = 100;
}
```

Chiar daca **var_stat** este declarata ca **externa,** nu se poate face legatura (linkage) din cauza ca a fost declarata **static** in **f1.cpp**.

III. Intelesul cuvantului **static** in clase va fi explicat ulterior.

Variabile registru

- un tip de variabila locala
- au in fata cuvantul cheie **register** care ii spune compilatorului ca aceasta variabila trebuie sa fie accesata cat mai rapid (folosind registrii de memorie)
- utilizarea registrilor nu este insa garantata, este lasata la latitudinea compilatorului
- NU pot fi declarate ca variabile globale sau statice
- se pot folosi ca argument al unei functii
- nu este recomandata utilizarea lor, functia de optimizare a compilatorului fiind suficient de avansata cat sa asigure plasarea optima in memorie.

Variabile volatile

- cuvantul cheie volatile ii spune compilatorului ca este vorba de o variabila care isi poate modifica valoarea oricand

- se folosesc daca se citesc valori din afara programului si nu exista control din cod asupra valorii lor

Ex: comunicatii cu dispozitive hardware – un senzor anunta o noua valoare; multithreading – o functie la care nu avem acces isi transmite rezultatul

- o variabila **volatile** - este citita oricand e nevoie de valoarea ei
  - modificata oricand este necesar – din exteriorul programului

Locatii pentru stocarea datelor - Memorie, cache, registrii

Calculatorul detine 3 locatii pentru stocarea datelor:

- **memoria** - dimensiune mare comparativ cu celelalte doua locatii
  - fiecare celula de memorie este accesata folosind o adresa
  - locatiile de memorie nu sunt neaparat consecutive

- **memoria Cache** - la nivelul CPU (cache de nivel 1) sau pe placa de baza (cache nivel 2)
  - stocheaza copii ale partilor de memorie utilizate de curand -> in locatii care sa poata sa fie accesate mai repede/usor
  - este “ascunsa” de catre hardware – se lucreaza efectiv cu ea doar in aplicatii pentru nivel kernel

- **registrii** - sunt unitati de stocare in interiorul CPU - cu acces foarte/cel mai rapid (se va intra in amanunte la alte materii: ex. microprocesoare)

C si C++: Spatiul (virtual) de adresa al unui proces

Stiva sau **stocarea automata** contine:

- variabilele locale declarate ca parte a unei functii
- parametrii functiilor
- rezultatele returnate de functiile apelate
- continutul registrilor si rezultatele returnate de functii

cand este apelata o rutina de tip serviciu de intrerupere.

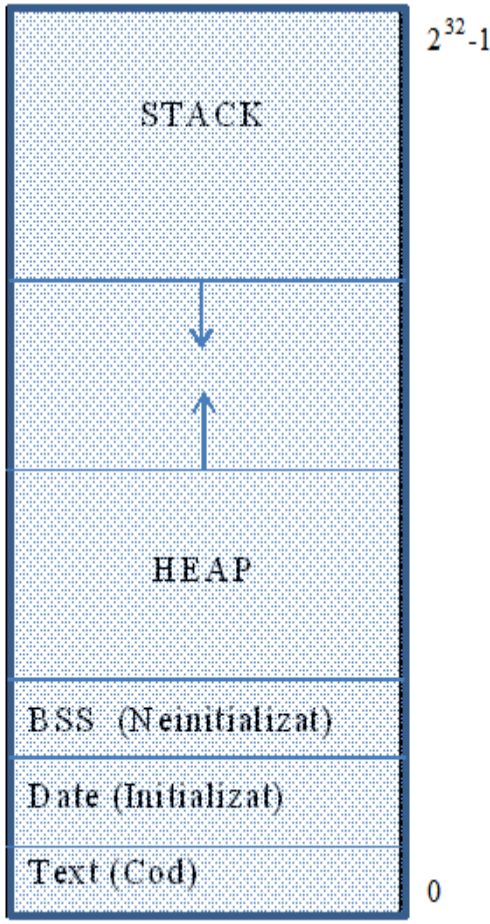
Regiunea **Date** este utilizata pentru stocarea:

- variabilelor **globale initializate**
- variabilelor **statice locale initializate**

Regiunea **Text** este utilizata pentru stocarea de:

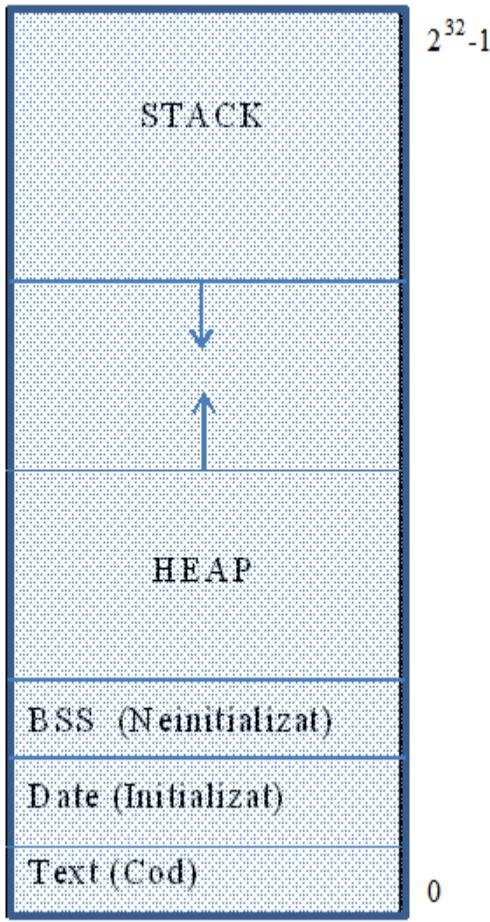
- cod (memoria este alocata de compilator cand incepem un program C/C++)
- cod compilat (assembly)

Diagrama memoriei - regiuni



❖ In C, variabilele globale si statice, fara initializare explicita sunt initializate cu 0 sau pointer null.

❖ Implementarile in C reprezinta in mod curent valorile de zero sau de pointer null utilizand un sablon de biti ce contine numai biti cu valoarea zero.



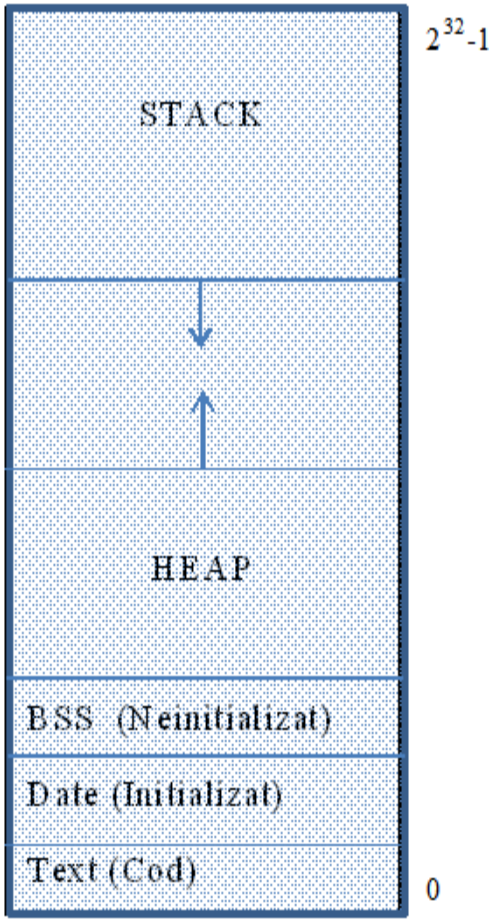
Regiunea **BSS (block started by symbol/block storage segment)** include:

- variabile **globale neinitializate**
- variabile **locale statice neinitializate**

Restul memoriei este lasata la dispozitie pentru alte utilizari – este libera.

Se numeste memorie **Heap (memorie dinamica)**

- este regiunea de memorie alocata dinamic (folosind operatorul new (Curs 2) sau functiile malloc, calloc si realloc)
- este o alternativa la memoria locala de tip stiva.



Memoria de tip stiva (stocare automata) este alocata **automat** la apelul unei functii si **eliberata** automat in momentul iesirii din functie.

Memoria **Heap** este total diferita:

- programatorul cere **explicit alocarea** unui bloc de memorie cu dimensiunea dorita
- blocul ramane alocat atata timp cat nu este **eliberat explicit** (free sau delete(Curs 2))

Nimic **nu se intampla in mod automat**.

Programatorul are astfel tot **controlul,** dar si toata **responsabilitatea** managemenului acestei regiuni de memorie.

- Arhitecturile moderne asigneaza adresa cea mai de jos pentru heap, si cea mai de sus pentru stiva.

Heap

Avantajele : - **durata de viata** - programatorul controleaza exact cand memoria e alocata si eliberata.

- **dimensiunea** spatiului de memorie alocat poate fi controlata in detaliu.

Ex. *Se poate aloca spatiu pentru un vector de intregi de lungime dorita in momentul rularii (run-time) – iar spatiul ocupat va fi exact cel ocupat, pe cand, daca se foloseste memoria locala(stiva), s-ar aloca un spatiu pentru, de exemplu, 100 de intregi, sperand ca aceasta va fi destul sau nu prea mult.*

Dezavantajele - **mai multa munca**: alocarea in heap se face explicit de catre programator

- **mai multe erori posibile -** pot aparea alocari si adresari incorecte

Heap

- o zona mare de memorie pusa la dispozitia programului
- acesta poate solicita blocuri de memorie
- pentru alocarea unui bloc de dimensiune oarecare programul face o cerere explicita prin apelul functiei de alocare heap
- functiile de alocare rezerva un bloc de memorie de dimensiunea ceruta si intoarce un pointer catre aceast spatiu.

- managerul memorie heap aloca blocuri oriunde atata timp cat blocurile nu se suprapun si au cel putin lungimea solicitata.

- la orice moment de timp, unele din zonele memoriei heap sunt alocate programului si se afla in starea de utilizare (**in use)**.

- alte zone nu au fost alocate si se gasesc in starea libera (**free**), ele pot fi alocate in mod sigur

Managerul memoriei heap

- are structuri private de date pentru a tine evidenta zonelor libere si ocupate
- el satisface cererile de alocare si eliberare de spatiu.
- cand programul nu mai foloseste o zona de memorie -> face o cerere de eliberare explicita catre managerul memoriei heap -> acesta isi updateaza structurile de date private astfel incat blocul de memorie eliberat sa fie considerat liber si reutilizat.
- dupa eliberare, pointerul continua sa pointeze catre blocul eliberat (*dangling pointer*), iar programul trebuie sa NU acceseze obiectul care ocupa spatiul tocmai eliberat.

!!!Pointerul inca exista, dar nu trebuie folosit.
Este indicat ca pointerul respectiv sa fie facut **null** imediat dupa eliberarea spatiului pentru a semnaliza explicit ca acel pointer nu mai e valid.

- Initial, toata memoria heap e goala/ libera.
- Are o dimensiune mare **DAR** poate sa fie ocupata in totaliate -> noi cereri de alocare nu mai pot fi satisfacute -> functia de alocare va intoarce aceasta stare in rularea programului –> NULL sau va arunca o exceptie de runtime specifica limbajului.

REMEMBER

void* malloc(size_t dim) - aloca un spatiu continuu de dimensiune *dim* octeti; returneaza un pointer de tipul *void* catre inceputul blocului alocat(daca s-a realizat alocarea cu succes) sau *NULL* daca nu se poate aloca memorie;

```
int *v;

v=(int*)malloc(3*sizeof(int));
```

void* calloc (size_t n , size_t dim) - aloca spatiu continuu pentru *n* elemente, fiecare necesitand '*dim*'' octeti. Spatiul alocat este initializat cu zero. Daca s-a alocat, returneaza un pointer catre inceputul spatiului alocat, altfel returneaza pointerul *NULL*.

```
v=(int*)calloc(3, sizeof(int));
```

void* realloc (void *ptr, size_t dim) –aloca dim octeti pentru ptr si copiaza in noua zona de memorie valorile stocate la vechile adrese

```
v=(int*)realloc(v, 4*sizeof(int));
```

Cand se termina operarea cu memoria ce a fost alocata dinamic cu una din functiile de mai sus, spatiul de memorie poate fi eliberat (cedat sistemului) prin aplicarea functiei **free()**

void *free (void *pointer) - elibereaza blocul de memorie referit de pointer

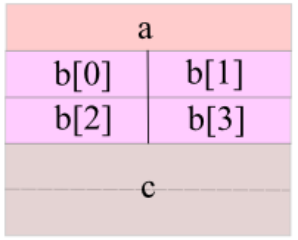
```
free(v);   /*urmat, ideal, de: */ v = NULL:
```


Biblioteci <stdlib.h> si <alloc.h>.

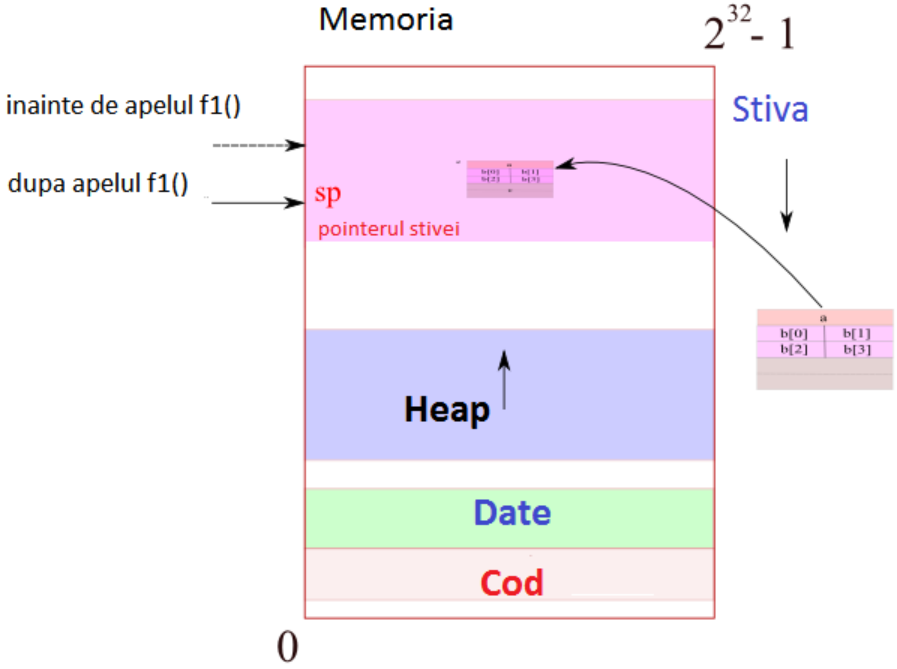
Stack

Pe exemplul de mai jos, vom urmari cum variabilele locale sunt pozitionate in stack:

```
void f1(){
        int a;
        char b[4];
        float c;
        f2();
        f3();
}
```



- *la apelul functie: f1() trebuie pus de-o parte spatiu de memorie: pointerul stivei este decrementat cu dimensiunea variabilelor lui **f1()***

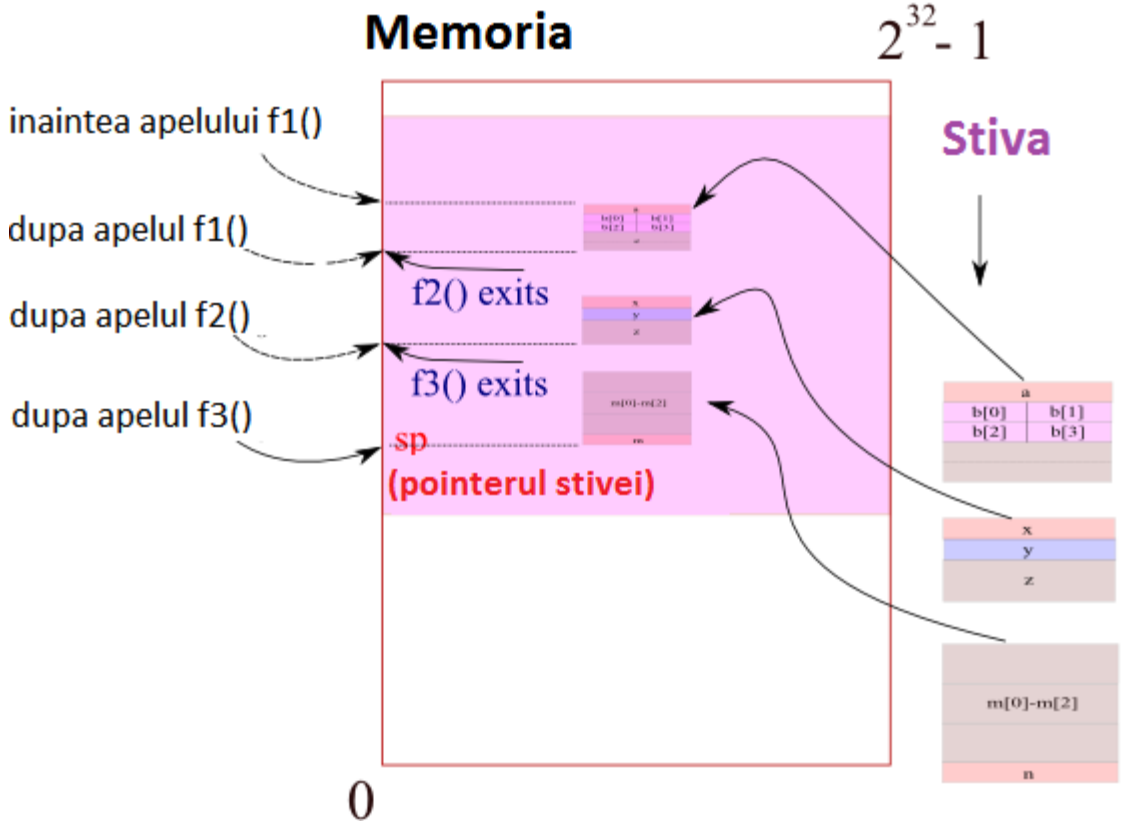


Compilator pt aplicatii

- 32 biti	**- 64 biti**
char : 1 byte	char : 1 byte
int : 4 bytes	int : 4 bytes
float : 4 bytes	float : 4 bytes
double : 8 bytes	double : 8 bytes

*in general; testati pentru siguranta cu sizeof()

```
void f1(){
        int a;
        short b[4];
        float c;
        f2();
        f3();
}

void f2(){
        int x;
        char *y;
        char z[2][2];
        f3();
}

void f3(){
        float m[3];
        int n;
}
```



sp pentru executia lui f1 : f1() -> f1.f2() -> f2.f3() -> f2.f3() se termina ->f1.f2() se termina ->f1.f3()-> f1.f3() se termina->f1 se termina

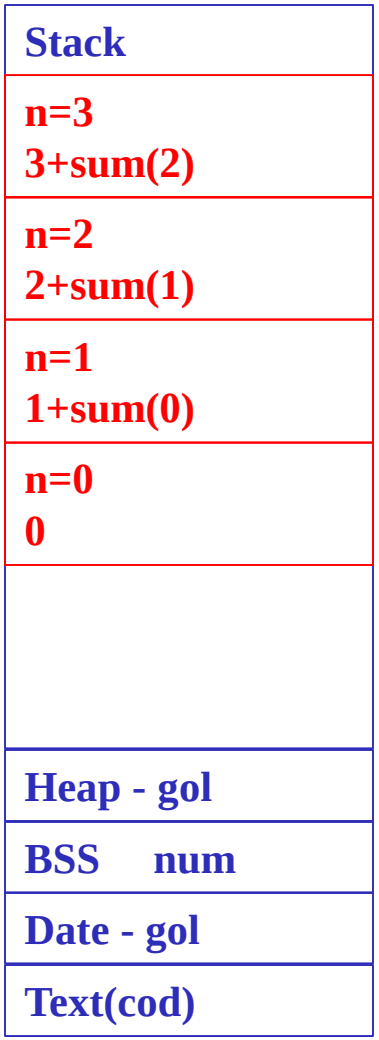
- *!!! dupa f1.f2(), cand se apeleaza f1.f3(),variabilele functiei f3() sunt suprascrise pe zona pana atunci ocupata de variabilele lui f2().*

Problema 1

Cum este ocupata memoria in cazul programului:

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

int num;

int sum(int n)
 {
     if(n==0) return n;
                else
                return n+sum(n-1);
  }

int main()
{
   num=3;
   sum(num);

   return 0;
}
```

sum(3)
=3+sum(2)
=3+2+sum(1)
=3+2+1+sum(0)
=3+2+1+0
=3+2+1
=3+3
=6



Problema 2

Cum e ocupata/eliberata stiva la apelul functiei:

```
int* f()
{
        int temp=10;
        return(&temp);
}
```

Catre ce pointeaza ptr dupa apelul:

```
int * ptr=pointer_local();
```

Problema 3

Cum este ocupata memoria in cazul programului de mai jos?

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

void func() {
static int i = 0;
// zona de date
i++;
cout << "i = " << i << endl;
}

int main()
{   //main – in stack
   int j;
   for (j=0; j<5; j++) func();
// j – in stack; apel func() – stack; iesire din func() x 5 ori – sp nu se modifica

    return 0;
}
```


Dar daca variabila i din func() nu e declarata static?

Problema 4

Pe exemplul despre variabile globale externe se comenteaza definirea si utilizarea variabilei var_e.
Care este efectul?

```
//f1.cpp
#include <cstdlib>
#include <iostream>
using namespace std;

//int var_e=10;
void funct();

int main(int argc, char *argv[])
{
  //  cout<<var_e<<endl; //10
    funct();                    //11

    return 0;
}
```

```
//f2.cpp
#include <iostream>
using namespace std;

extern int var_e;

void funct()
{
    cout<<++var_e;
}
```

[Linker error] undefined reference to `var_e'

Q5: Care sunt formele corecte pentru main()?

static int main()

int main(char** argv, int argc)

int main(int argc, char** argv)

int main()

int main(char* argv[], int argc)

int main(int argc, char* argv[])

void main()

A: int main()
int main(int argc, char* argv[])

- main() trebuie sa returneze un int
- trebuie sa nu aiba parametri sau primii parametrii sa fie:
  (int argc , char* argv[]) – cate argumente se transmit inainte de executie si care sunt ele
- un program care are functia main() declarata inline sau static nu va functiona.

Problema 6

Ce se afisaza la finalul secventei?

Cum se incarca stiva?

```
void modif1(int*x){
    *x=(*x)+1;
}

void modif2(int y){
    y=y+1;
}

int main(int argc, char *argv[]){
    int *x=(int *)malloc(sizeof(int));
    int y=3;
    x=&y;
    modif1(x);
    printf("%d\n",x);
//adresa catre care pointeaza x: 2293316
    printf("%d\n",*x);
//ce se gaseste acolo: 4
    printf("%d\n",&x);
//adresa la care e stocat x: 2293320
    printf("%d\n",&y);
//adreasa la care e stocata valoarea lui y:
//2293316
    printf("%d\n",y);
// valoarea variabilei y: 4
    modif2(y);
    printf("%d\n",y);
//valoarea variabilei y: 4

    return 0;
}
```

Problema 7

1.Alocati dinamic spatiu pentru un vector de dimensiune n de intregi si pentru o matrice lxc de intregi.

2. Alocati astfel incat elementele tablourilor sa fie initializate cu 0.

Problema 8

Alocati dinamic spatiu pentru o variabila numar complex, un vector/matrice de numere complexe – unde, numar complex e o structura.

Problema 9

Realocati spatiu pentru tablourile din Problema 7– dimensiunile noi sunt n-1 si (l+1)*c.

Problema 10

Alocati dinamic spatiu pentru siruri de caractere. Utilizati functiile strlen, strcat, strcmp, strcpy, etc

Anexa I - Tipul pointer

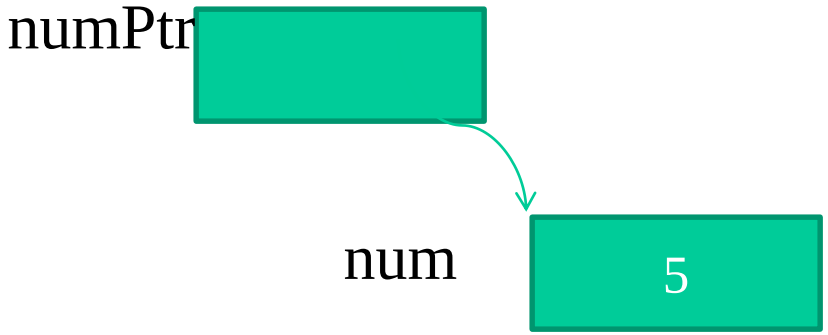
Ce este un pointer?

- Variabilele de tip int, float, char… opereaza destul de intuitiv. O variabila de tip int este ca o cutie (suficient de mare) in care se salveaza o singura valoare de tip intreg – ex: 5.

- Un pointer functioneaza diferit: nu stocheaza direct o valoare, ci o referinta catre o anumita valoare (calea catre/adresa unde e stocata valoarea de interes) .

Tipul pointer

In desen, pointerul numPtr e ca o cutie care contine inceputul sagetii care arata catre valoarea referita la care se pointeaza.



Variabila num contine valoarea 5.

Variabila numPtr (pointerul) contine o referinta catre variabila num.

Valoarea lui numPtr nu este un int, ci referinta la un int.

Dereferentiere
Dereferentierea este o operatie prin care se urmareste referinta unui pointer - ca sa se recupereze valoarea catre care se pointeaza.

Daca dereferentiem numPtr se obtine 5.

Ca sa putem realiza aceasta operatie – trebuie sa fim siguri ca pointerul referentiaza ceva, altfel o sa avem o eroare.

Tipul pointer

Pointerul NULL

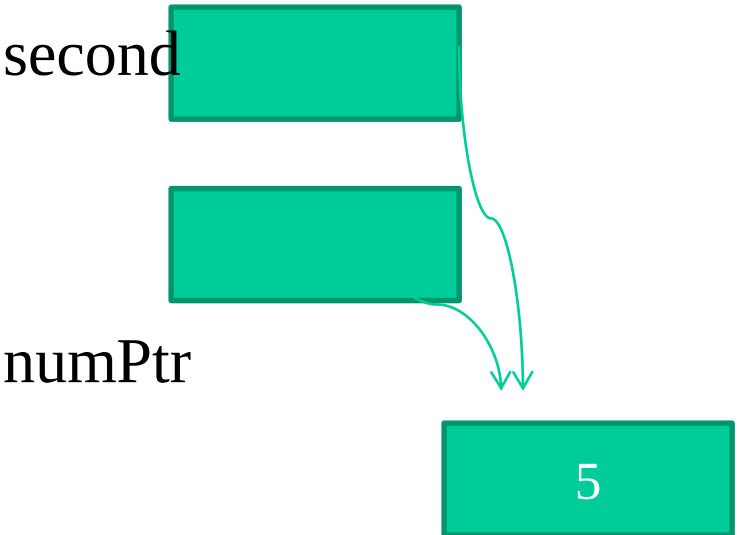
- NULL este o constanta – o valoare pointer speciala - care inseamna “pointeaza catre nimic” = “nu referentiaza nimic”.
- Dereferentierea unui pointer NULL conduce la o eroare la rulare.

Atribuirea de pointeri

- Operatorul = intre 2 pointeri -> ii face sa pointeze catre aceeasi zona de memorie:

second= numPtr

Dupa atribuire, operatorul ==
va returna adevarat
la testul second == numPtr



Operatorul = se foloseste si la atribuirea unei valori nule unui pointer:

numPtr = NULL

Tipul pointer

Pointeri – sintaxa

Un tip pointer in C/C++ se reprezinta ca tipul catre care se pointeaza urmat de *(asterix):

```
int *      // pointer la intreg
float*   // pointer la float
```

Variabilele de tip pointer se declara ca orice alta variabila: tip, nume, posibila initializare.

```
int * numPtr;       //pointer la un intreg – numele este numPtr
int * numPtr2=NULL;       //pointer la un intreg – numele este numPtr2, valoare
                                  // initiala NULL
```

OBS: se ocupa in memorie spatiu pentru pointer, dar nu si pentru elementul catre care se va pointa

Tipul pointer

Operatorul referinta &

Exista mai multe feluri in care se poate calcula referinta catre un/adresa unui element catre care se pointeaza.

Cel mai simplu, aceasta se afla cu operatorul &:

```
int num;
int*numPtr;
num=5;
numPtr=&num;    // calculeaza referinta catre num  (adresa lui num)
                //si o stocheaza in numPtr
```

Operatorul de dereferentiere ***

Operatorul * - dereferentiaza un pointer.

Este un operator unar si se plaseaza la stanga operandului.

*numPtr <=> num <=> *(&num)

Tema: Identificati care e nivelul de precedenta pentru * si & fata de restul operatorilor.

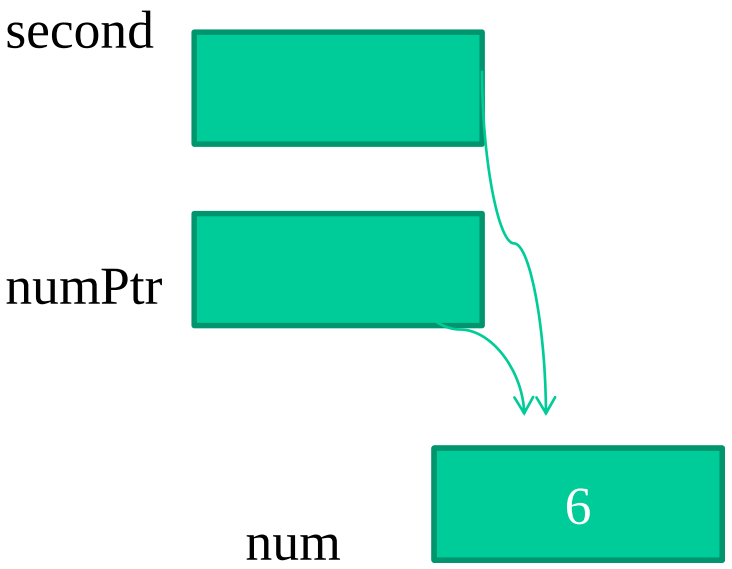
Tipul pointer

Sharing – 2 pointeri care adreseaza aceeasi zona de memorie se spune ca o folosesc in comun.

Copiile profunde VS copiile superficiale

O copie superficiala este rezultatul operatiei de folosire in comun a memoriei:

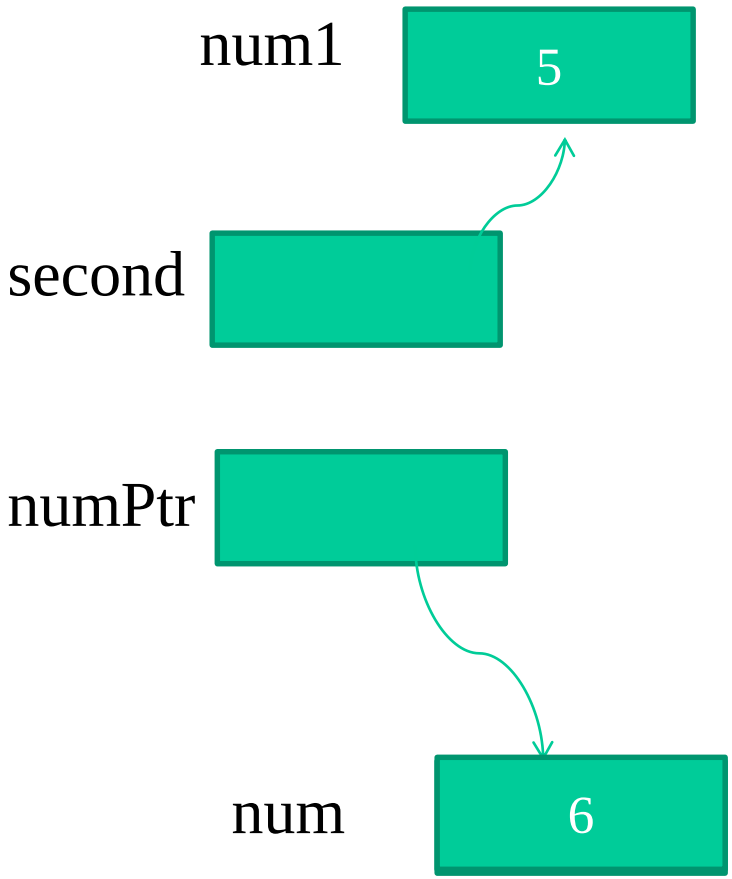
```
int *second, *numPtr;
int num = 5;

numPtr = &num;

second=numPtr;//copie superficiala

num++;

printf("%d %d\n" ,  *second, *numPtr);
```



Tipul pointer

Daca vreau ca fiecare pointer sa pointeze catre o zona de memorie de sine statatoare -> nu folosesc operatorul= intre ei.

```
int *second, *numPtr;

int num, num1 ;

num= num1 = 5;

second=&num1;
numPtr = &num;

num++;

printf("%d %d\n" , *second ,*numPtr);
```



Tipul pointer

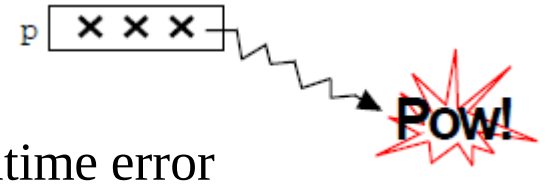
“Bad pointers”

- In momentul in care declar o variabila de tip pointer – ea nu adreseaza/ referentiaza pe nimeni -> pointerul nu e initializat (“bad pointer”).
- O operatie de dereferentiere conduce la o eroare la rulare.
- Ideal ar fi daca aplicatia ar arata eroare imediat, dar se poate intampla ca executia programului sa continue, iar eroarea/”crash”-ul sa apara mai tarziu

```
int *p;
printf(“%d\n” , *p);  // eroare logica – dereferentializarea unui pointer neinitializat
```

sau:

```
int* p;     // am declarat variabila de tip pointer , dar nu am initializat-o cu adresa
            //unei variabile
*p = 5;   //  dereferentierea pointerului neinitializat= >  runtime error
```



Tipul pointer

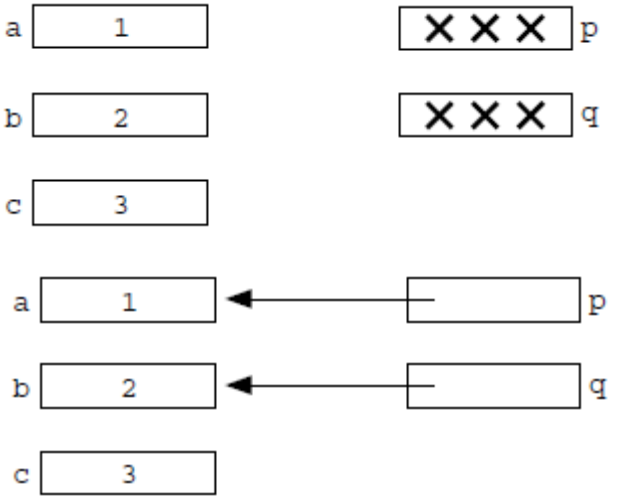
Exemple. Ce se afisaza? Reprezentati grafic.

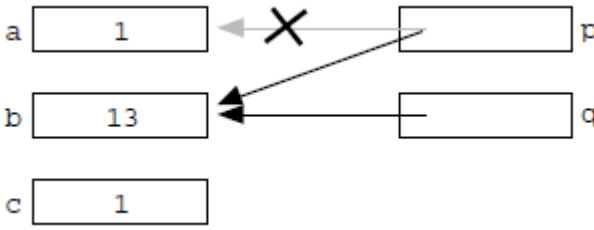
```
int a = 1;
int b = 2;
int c = 3;
int* p;
int* q;

p = &a;
q = &b;

c = *p;

p = q;
*p = 13;

printf("%d %d\n" , *q ,*p);

printf("%d %d %d\n",a, b,c);
```

Tipul pointer

Exemple

```
//main
// variabile: 3 intregi: a, b, c  si 2 pointeri la intregi: p, q
int a = 1;
int b = 2;
int c = 3;
int* p;
int* q;

p = &a; // p pointeaza la a
q = &b; // q pointeaza la b

c = *p; //recuperez valoarea catre care pointeaza p
        // si o stochez in c ; c=1
p = q;   // p si q pointeaza catre acceasi adresa -  a lui b
*p = 13;   // ce se gaseste la adresa referentiata de
           //p (in b) devine 13
printf("%d\n" , *q);   //q si p pointeaza catre aceeasi
                       //memorie - b - care contine 13
```





Tipul pointer

Dimensiunea unei variabile de tip pointer se poate afla cu operatorul **sizeof:**

```
int *p;
char *c;

printf("%d\n" , sizeof(p));
printf("%d\n" , sizeof(c));
```

Deoarece o variabila pointer contine o adresa, toti pointerii ocupa acelasi spatiu : cat sa stocheze o adresa – adica un intreg.

Tipul pointer

Legatura dintre pointeri si tablouri

Numele unui tablou este un pointer constant care are ca valoare adresa primului element din tablou.

Consideram declaratia: *tip_date* **vec[**dim**];**

Urmatoarele expresii sunt echivalente:

vec	&vec[0]	Adresa primului element din tablou
vec+i	&vec[i]	Adresa elementul de pe pozitia i din tablou
*vec	vec[0]	Primul element din tablou
*(vec+i)	vec[i]	Elementul de pe pozitia i din tablou
//vec++	*vec[1]*	*Elementul de pe pozitia 1*

!vec++ - ca instructiune da eroare pentru ca vec e o constanta –adresa de inceput a vectorului

Operatiile cu pointeri reprezinta o alternativa pentru adresarea elementelor din tablouri.

Tipul pointer

Legatura dintre pointeri si tablouri

O matrice este un vector de vectori, deci numele variabilei matrice e un pointer la prima linie din matrice.

Consideram declaratia unei matrice:

tip_date **m[**l**][**c**];**

Urmatoarele expresii sunt echivalente:

m	&m[0]		Adresa primei linii din matrice
m+i	&m[i]		Adresa liniei i din matrice
*m	m[0]	&m[0][0]	Adresa primului element de pe linia 0
*(m+i)	m[i]	&m[i][0]	Adresa primului element de pe linia i
((m+i)+j)		m[i][j]	Elementul de pe linia i si coloana j

Tipul pointer

Rezumat

- Un pointer stocheaza referinta catre o locatie anume din memorie. La acea locatie se salveaza ceva util.
- Operatia de dereferentiere a unui pointer acceseaza valoarea utila.
- Un pointer poate sa fie dereferentiat dupa ce a primit o valoare/adresa catre care sa pointeze => multe erori logice apar din cauza lipsei initializarii pointerului cu o adresa.
- Atribuirea (=) intre 2 pointeri inseamna ca amandoi pointeaza catre acelasi spatiu de memorie (sharing).

Tipul pointer

De ce sa folosim pointeri ?

Pointerii rezolva 2 probleme importante:

1. diferite sectiuni de cod pot sa acceseze aceeasi informatie in mod facil (o sa vedem la functii)

Acelasi efect se poate obtine copiind informatia in toate locurile necesare, dar asta asta implica mai mult timp consumat si mai mult spatiu de memorie ocupat.

2. Permit realizarea de structuri de date complexe – inlantuiri : liste, cozi, stive, arbori, etc