# Tehnici de programare - TP

## Cursul 4 – Structuri. Uniuni. Structuri cu câmpuri pe biți


**Ș.l. dr. ing. Cătălin Iapă**

**catalin.iapa@cs.upt.ro**

# Structuri de date abstracte: Stiva, Coada

Structuri

Uniuni

Structuri cu câmpuri pe biți

# Să ne amintim: Datele în programare

Un program gestionează date.

Datele pot fi:

- date simple: int, float, char etc.

- structuri de date simple: vector, matrice, struct, union etc.

- structuri de date abstracte: stiva, coada, dicționarul etc.

# Să ne amintim: Datele în programare

Structurile de date sunt utilizate pentru a organiza și stoca date într-un mod structurat și eficient.

Acestea sunt esențiale în dezvoltarea aplicațiilor și a algoritmilor, deoarece permit programatorilor să organizeze și să gestioneze seturi de date într-un mod logic și coerent.

O structură de date poate fi definită ca o colecție de date și o serie de operații care pot fi efectuate pe acestea.

# Să ne amintim: Stiva si Coada

Stiva este o structura de date LIFO (last in, first out), ceea ce inseamna ca ultimul element adaugat in stiva este primul element extras din stiva.

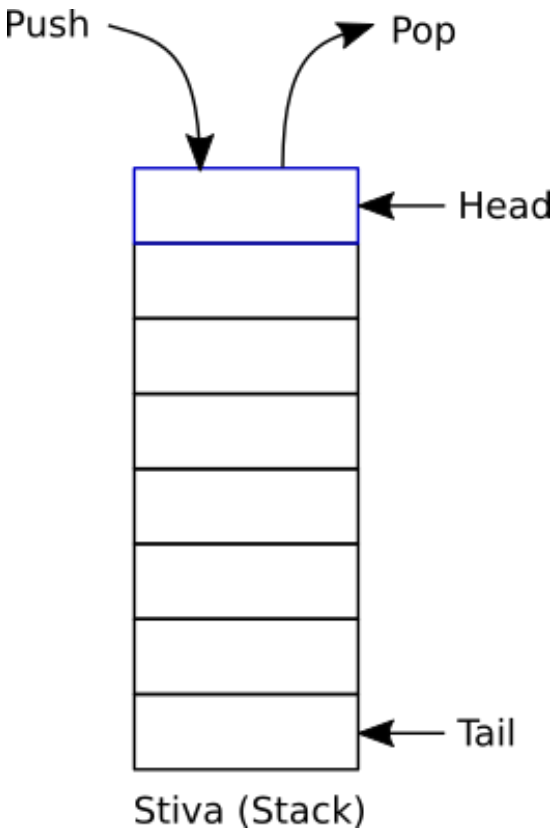
Coada este o structura de date FIFO (first in, first out), ceea ce inseamna ca primul element adaugat in coada o sa fie primul element extras din coada.

# Să ne amintim: Stiva si Coada

# Să ne amintim: Stiva si Coada

Stiva este asemanatoare cu o stiva de farfurii: se poate adauga o farfurie noua doar in varful stivei si nu se poate scoate decat ultima farfurie adaugata.

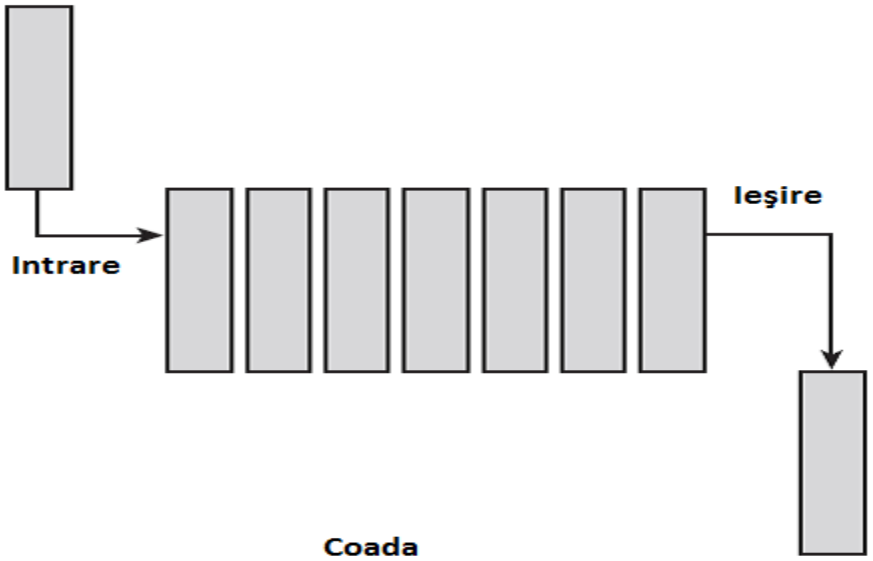
Un exemplu de folosire a stivei in programare este pentru a urmari apelurile recursive.



# Să ne amintim: Stiva si Coada

Coada este asemanatoare cu o coada la supermarket: primul client care ajunge la coada este primul care este servit si paraseste coada.

In programare, coada este folosita pentru a gestiona sarcinile in ordinea in care au fost adaugate sau pentru a stoca elementele in asteptare pana cand sunt procesate.



Structuri de date abstracte: Stiva, Coada

**Structuri**

Uniuni

Structuri cu câmpuri pe biți


WHEN YOU START PROGRAMMING
AFTER A WHILE
Hello World!
@pycoders
Data Structures

# Structuri

Structurile sunt utilizate pentru a stoca un grup de variabile sub un singur nume.

Acestea ne permit să definim tipuri de date personalizate.

# Definirea unei structuri

Pentru a defini o structură în C vom folosi cuvântul cheie struct.

În interiorul unei structuri se pot defini doar variabile, nu și funcții.

Structurile se pot defini oriunde într-un program, inclusiv în interiorul funcțiilor, dar, în general, se definesc în exteriorul lor, pentru a fi accesibile de oriunde din program.

# Definirea unei structuri

La fel ca în cazul variabilelor și a funcțiilor, structurile trebuie mai întâi definite și apoi folosite.

Componentele unei structuri pot fi oricât de complexe: vectori, matrici, alte structuri, etc.

Definirea unei structuri creează un nou tip de date, pe care îl putem folosi la fel ca pe tipurile de date predefinite (ex: **int**, **float**, **char**).

# Definirea unei structuri

```
struct Produs{
   char nume[200];
   float pret;
   int stoc;
};
```

Cuvântul cheie **struct** definește o nouă structură de date având numele dat și conținând între acolade (**{}**) toate definițiile componentelor sale.

# Accesarea componentelor unei structuri

Variabilele de tip structură se pot folosi în mod unitar, ca un întreg, sau se poate opera asupra componentelor lor specifice.

Componentele unei structuri se mai numesc câmpuri ale structurii, membrii ai structurii sau atribute ale structurii.

Componentele structurii pot fi accesate prin intermediul operatorului punct (.) - utilizăm numele variabilei de tipul structurii, urmat de punct și numele membrului structurii.

*struct Produs p1, p2;*

*p1=p2;*
*p2.pret = 7.5;*

# Dimensiunea unei structuri

Dimensiunea unei structuri este dată de dimensiunea cumulată a tuturor câmpurilor structurii. (în unele cazuri, pentru eficiența accesării memoriei, poate lăsa și câmpuri goale de memorie care se adaugă în plus la dimensiunea totală)

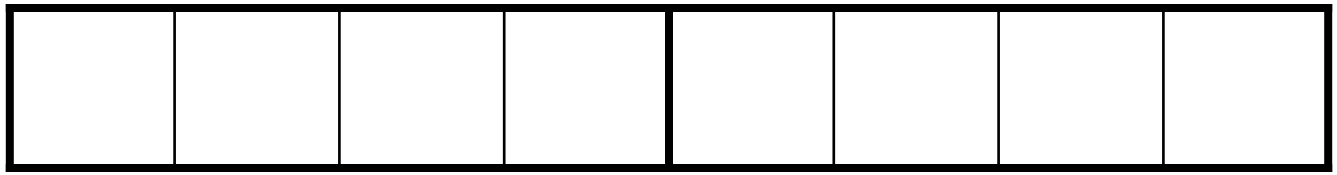
Aflăm dimensiunea în bytes a unui element de tip structură la fel ca la orice alte date, cu *sizeof(struct nume);*

```
struct Produs{
   float pret;
   int stoc;
} a;
```

*Variabila a ocupa 8 bytes în memorie ( 4 bytes de la float + 4 bytes de la int).*

# Dimensiunea unei structuri

```
struct Produs{
   float pret;
   int stoc;
} a;
```



# Inițializarea unei structuri

O structură se inițializează dând valori între acolade pentru câmpurile structurii, în ordinea apariției lor în declarație.

```
struct Elem
{
        int n;
        char text[10];
};
int main()
{
        struct Elem str3 = {78, "test"};
        return 0;
}
```

# Vectori cu structuri

Putem aloca vectori care să aibă elemente de tipul structurii. Accesarea câmpurilor structurilor se face accesând mai întâi elementul din vector și apoi se pune punct și numele câmpului.

```
struct Elem {
        int n;
        char text[10];
};
int main() {
        struct Elem v[10];
        v[3].n=3;
        strcpy(v[3].text, ”sir”);
        scanf(“%d”, &v[4].n);
        printf(“%s”, v[3].text);
        return 0;
}
```

# Pointeri la structuri

Putem declara pointeri spre elemente de tipul structurii și să îi folosim la fel ca un alt tip de pointer. Pentru a lucre cu elemente de tip structură, în acest caz, mai întâi trebuie să alocăm maorie dinamic. În acest caz, pentru a accesa câmpurile structurii vom folosi una din cele 2 sintaxe:

*(*p).text* *sau* *p->text*

```
struct Elem {
        int n;
        char text[10];
};
int main() {
        struct Elem *p;
        p = (struct Elem*) malloc (1 * sizeof(struct Elem));
        p->n=3;
        strcpy(p->text, ”sir”);
        return 0;
}
```

# Transmiterea ca parametru sau returnarea valorii

Putem folosi tipuri de date structură ca și argumente la funcții sau ca și valoare returnată:

```
void afisare (struct Elem s){
        printf("%d", s.n);
        printf("%s", s.text);
}
int main()
{
        struct Elem str = {78, "test"};
        afisare(str);
        return 0;
}
```

Structuri de date abstracte: Stiva, Coada

Structuri

**Uniuni**

Structuri cu câmpuri pe biți

# Uniuni

În C, o uniune este un tip de date similar ca și sintaxă cu structura, iar rolul ei este de a economisi spațiu în memorie atunci când avem nevoie să stocăm doar un singur tip de date la un moment dat.

Uniunile au o diferență importantă față de structuri: toate câmpurile unei uniuni împart același spațiu de memorie.

Inițializarea, utilizarea, asignarea, restricțiile, utilizarea cu pointeri sunt la fel cu cele din cazul structurilor.

# Uniuni

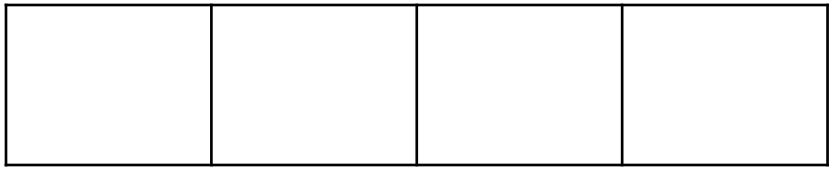
Uniunile pot fi utilizate pentru a interpreta aceeași locație de memorie în diferite moduri, în funcție de tipul de date care este stocat în uniune. De exemplu, o uniune poate fi utilizată pentru a interpreta aceeași secvență de biți ca un număr întreg sau ca un șir de caractere.

```
union int_sau_string {
        int a;
        char s[4];
};
```

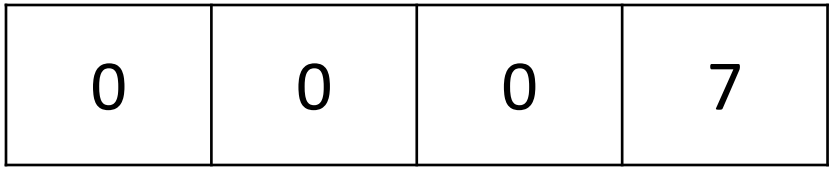
Dimensiunea pe care o va ocupa un element de tipul uniunii este dimensiunea maximă a câmpurilor. În cazul de mai sus dimensiunea stucturii va fi de 4 bytes.

# Uniuni

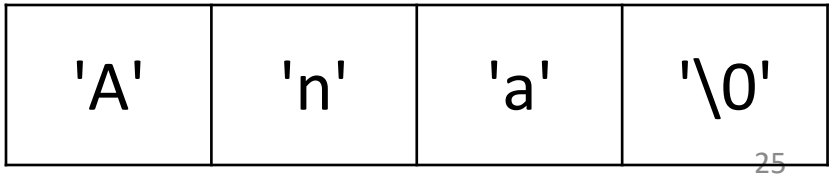
```
union int_sau_string {
        int a;
        char s[4];
} x;
```



```
x.a=7;
```




```
strcpy(x.s, "Ana");
```



# Uniuni

O uniune nu știe prin ea însăși ce membru al ei este folosit la un moment dat. De aceea, în general trebuie folosită o altă variabilă în care să ținem minte ce membru folosim dintr-o uniune.

```
// cantitatea unui produs: numar de bucati sau greutatea, in kg
union Cantitate{
   int nBucati;
   double kg;
   };
 struct Produs{
   int um;        // unitate de masura: 0-la bucata, 1-kilograme
   union Cantitate c;
   };
```

# Uniuni

În caz că o uniune se folosește doar în interiorul unei structuri, putem să declarăm acea uniune și câmpul de tipul său direct în interiorul structurii respective. Mai mult decât atât, putem să omitem numele uniunii, folosind astfel o uniune *anonimă*.

```
struct Produs{
    int um;      // unitate de masura: 0- bucata, 1-kg
    union{
        int nBucati;
        double kg;
        }c;          // cantitate
    };
```

# Uniuni

Ca orice structură, *Produs* mai poate conține și alte câmpuri, în afară de uniune și de selectorul său. De exemplu, putem să adăugăm câmpuri care sunt comune pentru orice fel de produs:

```
struct Produs{
   char nume[20];
   double pret;
   int um;        // unitate de masura: 0- bucata, 1- kg
   union{
      int nBucati;
      double kg;
      }c;          // cantitate
   };
```

# Tipuri de probleme cu uniuni

Tipurile de probleme în care e eficient să folosim uniuni este acela în care anumite caracteristici se folosesc pentru un anumit tip, iar alte caracteristici se folosesc pentru alte tipuri.

Exemplu:

Să se realizeze o aplicaţie pentru gestionarea unei biblioteci. Se vor memora următoarele informaţii pentru fiecare înregistrare:

- Titlu, Autor, Număr de exemplare, Preț, Număr de pagini
- **Dacă e Carte sau Revistă**
- **Dacă e carte se va memora ISBN, sir de 13 caractere, exemplu: 9783161484100, iar dacă e revistă se va memora anul în care a apărut primul număr.**

Structuri de date abstracte: Stiva, Coada

Structuri

Uniuni

**Structuri cu câmpuri pe biți**

# Structuri cu câmpuri pe biți

Structurile cu câmpuri pe biți sunt o modalitate eficientă de a stoca date în memorie, reducând astfel consumul de memorie.

Sunt utile în situațiile în care o variabilă este de dimensiunea unui tip de date (int sau char, eventual unsigned), dar nu toți biții acesteia sunt folosiți în timpul execuției programului.

# Structuri cu câmpuri pe biți

```
struct Data{
   int zi;              // ocupă 4 bytes
   int luna;            // ocupă 4 bytes
   int an;              // ocupă 4 bytes
   };                   // total structură 12 bytes = 96 biți
```

Dacă analizăm mai atent lucrurile, observăm că fiecare dintre cele trei câmpuri ar putea fi memorat pe mai puţin de patru octeţi.

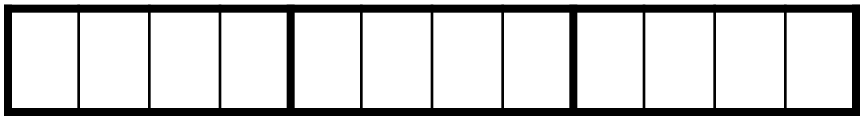
- câmpul *zi* va lua valori între 1 şi 31, deci ar putea fi memorat pe 5 biţi (cea mai mică putere a lui 2 mai mare decât 31 este $32=2^5$).
- câmpul *luna* va lua valori între 1 şi 12, deci ar putea fi memorat pe 4 biţi (cea mai mică putere a lui 2 mai mare decât 12 este $16=2^4$).
- câmpul *an* să presupunem că va lua valori între -9999 şi 9999, deci poate fi reprezentat pe 15 biţi (avem din start un bit de semn deoarece vrem să putem reţine şi numere negative, iar cea mai mică putere a lui 2 mai mare decât 9999 este 16384=2^14; rezultă în total 15 biţi).

# Structuri cu câmpuri pe biți

**Structură normală**

```
struct Data{
    int zi;
    int luna;
    int an;
    };
```

Dimensiune: 12 bytes = 96 biți



**Structură cu câmpuri pe biți**

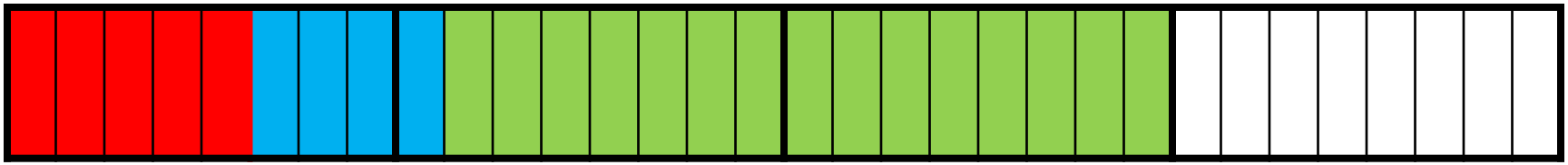
```
struct DataBiti {
    unsigned int zi:5;
    unsigned int luna:4;
    int an:15;
    };
```

Dimensiune: 24 biți (se mărește la multiplu de 32 biți, 4 bytes, 1 int)

# Structuri cu câmpuri pe biți

```
struct DataBiti {
    unsigned int zi:5;
    unsigned int luna:4;
    int an:15;
    };
```

Dimensiune: 24 biți (se mărește la multiplu de 32 biți, 4 bytes, 1 int)



zi luna an spațiu nefolosit

# Structuri cu câmpuri pe biți

Într-o structură pot alterna câmpurile definite pe biţi cu cele definite clasic.

```
struct Panificatie {
    unsigned int tip:2;
    unsigned int nCereale:3;
    float greutate;
    };
```

Pentru a se eficientiza efectiv spațiul de memorie e necesar ca toate câmpurile definite pe biți să nu fie despărțite de alte câmpuri, toate câmpurile pe biți să fie scrise grupat, unul după altul.

# Structuri cu câmpuri pe biți

Dacă vom lucra doar cu numere pozitive e necesar să specificăm asta prin modificatorul unsigned.

Un câmp definit pe doi biţi cu modificatorul *unsigned* va reţine valori de la 0 până la 3. Dacă nu apare modificatorul *unsigned*, atunci câmpul este cu semn şi va reţine valori de la -2 până la 1.

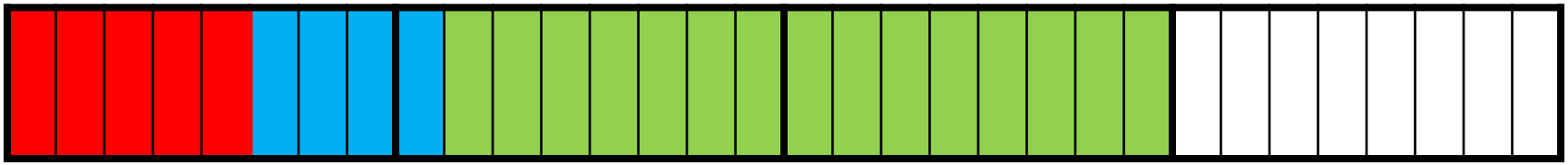
```
struct Panificatie {
    unsigned int tip:2;
    unsigned int nCereale:3;
    float greutate;
    };
```

# Structuri cu câmpuri pe biți

Un câmp pe biți nu are adresă. Din cauză că el poate fi plasat în memorie în interiorul unui octet, nu se poate lucra cu adresa lui, deoarece adresele de memorie sunt adrese de octeți.

Pentru a citi valori în câmpuri pe biți folosind funcția *scanf*, se poate face citirea într-o variabilă auxiliară, iar apoi se poate copia valoarea din variabila auxiliară în câmpul structurii.

```
struct Elem s;
   int aux;
   scanf("%d", &aux);
   s.a = aux;
```



Vă mulțumesc!