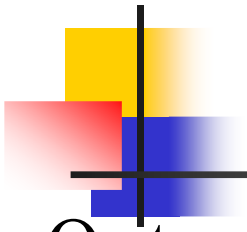
Structuri şi uniuni

1. Bazele structurilor
2. Structurile şi funcțiile
3. Transmiterea structurilor ca argumente
4. Structurile şi pointerii
5. Tablouri de structuri
6. Pointerii spre structuri
7. Typedef
8. Alocarea dinamică
9. Uniuni
10. Câmpuri de biți



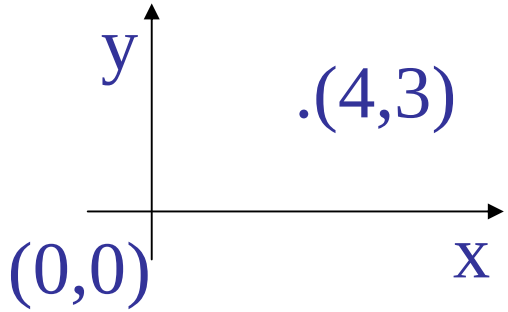
Structuri

- O structură este o colecție de una sau mai multe componente, posibil de tipuri diferite, grupate împreună sub un singur nume pentru uşurința manipulării în ansamblu.
- Structurile ajută la organizarea datelor complicate, în special în programe ample, deoarece permit ca un grup de variabile care au ceva în comun să fie tratate ca o unitate, în loc de a fi considerate entități separate.
- Un exemplu tradițional de structură este o înregistrare din statul de plată: un angajat este descris printr-un set de atribute precum *nume*, *adresă*, *cod numeric personal*, *salariu* etc. Unele dintre acestea pot fi, la rândul lor structuri: un nume are mai multe componente, la fel o adresă şi chiar un salariu.

Structuri

Bazele structurilor

- Să creăm o structură potrivită pentru grafică. Obiectul de bază este un punct, care are o coordonată x şi o coordonată y, ambele întregi.



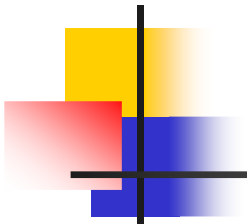
- Cele două componente pot fi plasate într-o structură declarată astfel:

```
struct punct {
    int x;
    int y;
};
```

Structuri

Bazele structurilor

- Cuvântul cheie struct introduce o declarație de structură, care este o listă închisă între acolade.
- După cuvântul struct poate urma un nume opțional, numit *nume generic al structurii* (structure tag) (punct în exemplu). Numele generic reprezintă acest tip de structură şi poate fi folosit ulterior ca prescurtare pentru partea declarației cuprinsă între acolade.
- Variabilele definite într-o structură (componentele structurii) se numesc *membri*.
- Un membru al unei structuri sau numele generic al acesteia şi o variabilă obişnuită (care nu este membru) pot avea acelaşi nume fără a apărea conflicte, deoarece acestea pot fi oricând deosebite prin context.
- În structuri diferite pot să apară aceleaşi nume de membri.



Structuri

Bazele structurilor

- O declarație struct definește un tip. => Acolada închisă care încheie lista de membri poate fi urmată de o listă de variabile, la fel ca în cazul oricărui tip de bază.

```
struct { ... } x, y, z;
```

- O declarație de structură care nu este urmată de o listă de variabile nu alocă spațiu; doar descrie un *șablon de structură*.
- Dacă în declarație apare numele generic, acest nume poate fi folosit mai târziu în definirea unor instanțieri ale structurii.
- Declarația

```
struct punct pt;
```

definește o variabilă pt care este o structură de tipul struct punct.

Structuri
Bazele structurilor

- O structură poate fi inițializată plasând după definiția sa o listă de valori de inițializare pentru membri, fiecare valoare fiind o expresie constantă:

```
struct punct maxpt = { 320, 200 };
```

- Într-o expresie, un membru al unei structuri poate fi accesat printr-o construcție de forma:

```
nume-structura.membru
```

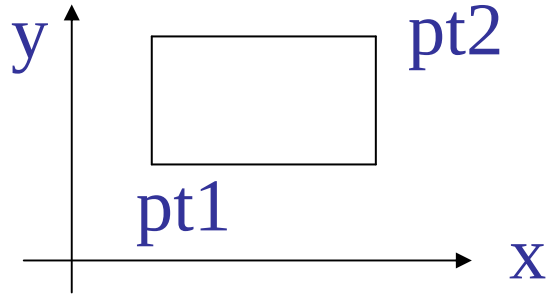
- Operatorul "." (punct) de acces la membrii unei structuri leagă numele structurii de numele membrului.

```
printf("%d,%d", pt.x, pt.y);
dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

Structuri

Bazele structurilor

- Structurile pot fi imbricate (incluse unele în altele).
- Exemplu: o reprezentare a unui dreptunghi este o pereche de puncte care reprezintă colțurile diagonal opuse.




```
struct drept {
        struct punct pt1;
        struct punct pt2;
}
```

Structuri

Bazele structurilor

- Structura drept conține două structuri punct.

Dacă declarăm variabila ecran ca

```
struct drept ecran;
```

atunci

```
ecran.pt1.x
```

se referă la coordonata x a membrului pt1 din ecran.

Structuri

Bazele structurilor

- Exemple de structuri:

```
struct data {
      int zi;
      int luna;
      int an;
};
struct angajat {
     char nume[64];
     int varsta;
     int categ_salarizare;
     float salariu;
     unsigned nr_marca;
} angajat_info, nou_angajat, fost_angajat;
```

Structuri
Structurile şi funcţiile

- Singurele *operații legale* cu o structură sunt:
  - copierea acesteia prin operația de atribuire sau prin transmiterea de argumente funcțiilor şi returnarea de valori din funcții(structura se consideră ca o unitate);
  - invocarea adresei structurii cu operatorul &;
  - accesarea membrilor unei structuri.
- Structurile nu pot fi comparate.
- O structură poate fi inițializată printr-o listă de valori constante pentru membri; o structură automatică poate fi, de asemenea, inițializată printr-o atribuire.

Structuri

Transmiterea structurilor ca argumente

- *Exemple*: funcții pentru lucrul cu punctele şi cu dreptunghiurile.

Există 3 abordări posibile pentru transmiterea parametrilor, fiecare cu avantajele şi dezavantajele sale:

- transmiterea componentelor separat;
- transmiterea unei întregi structuri;
- transmiterea unui pointer către aceasta.

Structuri

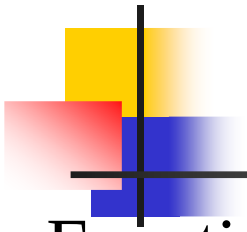
Transmiterea structurilor ca argumente

```
/*creazapunct: creaza un punct din comp. x si y */
struct punct creazapunct(int x, int y)
{
    struct punct temp;
     temp.x = x;
     temp.y = y;
     return temp;
}
```

- Nu există nici un conflict între numele argumentului şi membrul cu acelaşi nume; refolosirea numelor evidențiază mai clar relația dintre aceştia.

Structuri

Transmiterea structurilor ca argumente



- Funcția creeazapunct poate fi folosită pentru a inițializa dinamic orice structură, sau pentru a furniza unei funcții argumente de tip structură:

```
struct drept ecran;
struct punct mijloc;
struct punct creazapunct(int, int);
ecran.pt1 = creazapunct(0, 0);
ecran.pt2 = creazapunct(XMAX, YMAX);
mijloc = creazapunct((ecran.pt1.x + ecran.pt2.x)/2,
                     (ecran.pt1.y + ecran.pt2.y)/2);
```

Structuri

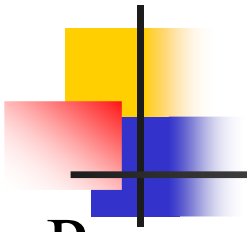
Transmiterea structurilor ca argumente

- *Exemple*: funcții care efectuează operații aritmetice cu puncte.

```
/* ptindrept: returneaza 1 daca p se afla in interiorul
dreptunghiului d si 0 in caz contrar */
int ptindrept(struct punct p, struct drept d)
{
    return p.x >= d.pt1.x && p.x < d.pt2.x
        && p.y >= d.pt1.y && p.y < d.pt2.y;
}
```

Structuri

Transmiterea structurilor ca argumente



- Presupunem că dreptunghiul este reprezentat într-o formă standard în care coordonatele lui pt1 sunt mai mici decît coordonatele lui pt2.

```
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))

/* canondrept: adu coordonatele dreptunghiului la forma canonica */
struct drept canondrept(struct drept d)
{
   struct drept temp;
```

Structuri

Transmiterea structurilor ca argumente

```
    temp.pt1.x = min(d.pt1.x, d.pt2.x);
    temp.pt1.y = min(d.pt1.y, d.pt2.y);
    temp.pt2.x = max(d.pt1.x, d.pt2.x);
    temp.pt2.y = max(d.pt1.y, d.pt2.y);
    return temp;
}
```

- Dacă trebuie să transmitem unei funcții o structură voluminoasă, este mai eficient să transmitem un pointer la structură decât să copiem întreaga structură. Pointerii spre structuri sunt identici cu pointerii spre orice variabilă.

Structuri
Structurile şi pointerii

- Declarația

```
struct punct *pp;
```

precizează că pp este un pointer la o structură punct. Dacă pp indică spre o structură punct, atunci *pp este structura, iar (*pp).x şi (*pp).y sunt membrii. Pentru a putea folosi pointerul pp ca mai sus, acesta trebuie mai întâi inițializat:

```
struct punct origine, *pp;
.....
pp = &origine;
.....
printf("originea este (%d,%d)\n", (*pp).x, (*pp).y);
```

Structuri

Structurile şi pointerii

- În expresia (*pp).x sunt necesare parantezele deoarece precedența operatorului de acces la membrii unei structuri **“.”**, este mai mare decât cea a operatorului *.
- Există o notație alternativă:

Dacă p este un pointer la o structură, atunci

p->membru-al-structurii

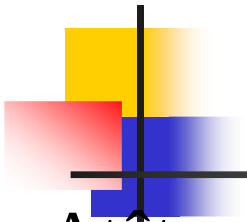
face referire la acel membru.

Am putea scrie cu acelaşi efect:

```
printf(“originea este (%d,%d)\n”, pp->x, pp->y);
```

Structuri

Structurile şi pointerii



- Atât operatorul **.** cât şi operatorul **->** se asociază de la stânga la dreapta, deci dacă avem:

```
struct drept d, *dp = &d;
```

atunci următoarele patru expresii sunt echivalente:

```
d.pt1.x
dp->pt1.x
(d.pt1).x
(dp->pt1).x
```

- Operatorii pentru structuri **.** şi **->**, împreună cu operatorul **()** pentru apelurile de funcții şi cu operatorul **[ ]** pentru indici, se află în vârful ierarhiei precedenței şi sunt foarte strâns legați de operanzii lor.

Structuri

Structurile şi pointerii

- Fiind dată declarația

```
struct {
   int lung;
   char *str;
} *p;
```

atunci:

++p->lung incrementează variabila lung, nu variabila p, deoarece asocierea implicită este ++(p->lung);

(++p)->lung incrementează variabila p înainte de a-l accesa pe lung;

(p++)->lung incrementează variabila p ulterior accesării.

Structuri

Structurile şi pointerii

- În acelaşi mod:

  *p->str preia obiectul spre care indică str;

  *p->str++ incrementează pointerul str după accesarea obiectului spre care indică acesta;

  (*p->str)++ incrementează obiectul spre care indică str;

  *p ++->str incrementează variabila p după accesarea obiectului spre care indică p.

Structuri

Tablouri de structuri

- Vrem să realizăm un program care să numere aparițiile fiecărui cuvânt cheie din C într-un text dat în fişierul de intrare.

Avem nevoie de un tablou de şiruri de caractere care să memoreze numele şi de un tablou de întregi care să memoreze rezultatele(numărul de apariții).

Definim structura:

```
struct cheie {
    char *cuvant;
    int rez;
} tabchei[NCHEI];
```

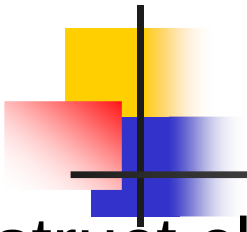
Structuri
Tablouri de structuri

- Putem scrie şi astfel:

```
struct cheie {
    char *cuvant;
    int rez;
};
struct cheie tabchei[NCHEI];
```

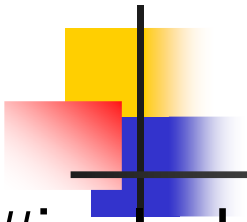
- Deoarece structura tabchei conține un set constant de nume, cel mai simplu este să o facem variabilă externă şi să o inițializăm o dată pentru totdeauna la definire.
- Inițializarea se face printr-o listă de valori de inițializare sub formă de perechi corespunzătoare cu membrii structurii, închise între acolade.

Structuri
Tablouri de structuri



```
struct cheie {
    char *cuvant;
    int rez;
} tabchei[ ] = {
     “auto”, 0,
     “break”, 0,
     “case”, 0,
       .....
     “while”, 0};
```

Acoladele interioare nu sunt necesare: {“auto”, 0},



Structuri

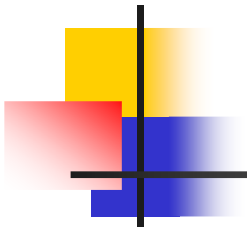
Tablouri de structuri

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXCUVANT 100
int preiacuvant(char *, int);
int cautbin(char *, struct cheie *, int);
/* numara cuvintele cheie C dintr-un text */
main()
{
    int n;
    char cuvant[MAXCUVANT];
```

Structuri

Tablouri de structuri

```
  while (preiacuvant(cuvant, MAXCUVANT) != EOF)
      if (isalpha(cuvant[0]))
          if ((n = cautbin(cuvant, tabchei, NCHEI)) >= 0)
              tabchei[n].rez++;
   for (n = 0; n < NCHEI; n++)
       if (tabchei[n].rez  > 0)
           printf("%4d %s\n",
                   tabchei[n].rez, tabchei[n].cuvant);
    return 0;
}
```



Structuri

Tablouri de structuri

```
/* cautbin: cauta cuvantul in tab[0] ... tab[n-1]  */
int cautbin(char *cuvant, struct cheie tab[ ], int n)
{
    int cond;
    int prim, ultim, mijloc;

    prim = 0;
    ultim = n – 1;
```

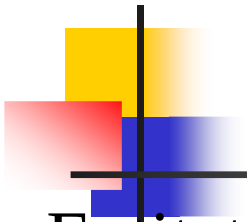
Structuri

Tablouri de structuri

```
  while (prim <= ultim)  {
      mijl = (prim + ultim) / 2;
       if ((cond = strcmp(cuvant, tab[mijl].cuvant)) < 0)
           ultim = mijl – 1;
       else if (cond > 0)
            prim = mijl + 1;
       else
            return mijl;
   }
   return -1;
}
```

Structuri

Tablouri de structuri



- Entitatea NCHEI reprezintă numărul de cuvinte cheie din tabchei.
- Cum o determinăm ? Dimensiunea tabloului este complet determinată în momentul compilării. Numărul de intrări este:

  dimensiunea lui tabchei / *dimensiunea lui struct* cheie

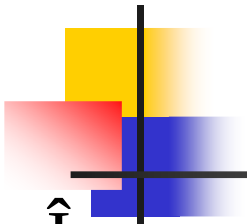
- Limbajul C pune la dispoziție un operator unar care acționează în momentul compilării, numită sizeof, şi care poate fi folosit pentru a a calcula dimensiunea unui obiect.
- Expresiile: sizeof *obiect* şi

  sizeof(*nume tip*)

furnizează dimensiunea în octeți a obiectului (variabilă, tablou sau structură) sau a tipului specificat.

Structuri

Tablouri de structuri



- În program, putem calcula numărul de cuvinte astfel:

- împărțind dimensiunea tabloului la dimensiunea unui element, într-o instrucțiune #define:

```
#define NCHEI (sizeof tabchei / sizeof(struct cheie))
```

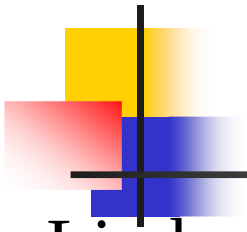
- împărțind dimensiunea tabloului la dimensiunea unui element precizat:

```
#define NCHEI (sizeof tabchei / sizeof tabchei[0])
```

care prezintă avantajul că nu trebuie schimbată dacă se schimbă tipul.

- Operatorul sizeof nu poate fi folosit într-o linie #if, deoarece preprocesorul nu analizează nume de tipuri (expresia din #define nu este evaluată de preprocesor).

Structuri
Typedef



- Limbajul C pune la dispoziție o facilitate numită typedef pentru *crearea de noi nume de tipuri de date*.

- Declarația

```
typedef int Lungime;
```

face numele Lungime sinonim cu int. Tipul Lungime poate fi folosit în declarații, conversii etc., în exact acelaşi mod ca şi int:

```
Lungime lung, maxlung;
Lungime *lungimi[ ];
```

Structuri
Typedef

- Declarația

```
typedef char *Sir;
```

face numele Sir sinonim cu char * (pointer spre caracter);
acesta poate fi folosit în declarații sau conversii:

```
Sir p, ptrlinie[MAXLINII], malloc(int);
int strcmp(Sir, Sir);
p = (Sir) malloc(100);
```

- Tipul declarat într-o construcție typedef apare în poziția unui nume de variabilă şi nu imediat după cuvântul typedef. Din punct de vedere sintactic, typedef este asemănător cu clasele de memorie extern, static etc.

Structuri
Typedef

- O declarație typedef nu creează un tip nou. Ea doar atribuie un nume unui tip existent.
- Motivele principale pentru utilizarea construcțiilor typedef sunt:

1. Creşterea portabilității programelor: dacă se folosesc construcții typedef pentru tipuri de date care pot fi dependente de maşină, numai construcțiile typedef trebuie modificate atunci când programul este mutat pe alt calculator.

2. Furnizarea unei documentații mai bune pentru un program.

Structuri
Alocarea dinamică

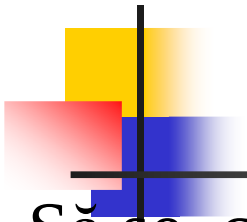
- Se considera tipul structurat persoana, definit în felul următor:

```
struct persoana {
   char nume[30];
   int varsta;
};
```

- O variabilă dinamică de tip pointer la tipul structurat persoana se alocă şi se utilizează în felul urmator:

```
struct persoana * pp;
pp=(persoana *) malloc (sizeof( persoana));
pp->varsta=20;
strcpy(pp->nume, "ion");
```

Structuri
Alocarea dinamică



- Să se citească datele (numele şi vârsta) despre un număr oarecare *n* de persoane şi să se memoreze într-o structură. Să se găsească o structură de date adecvată astfel încât consumul de memorie să fie optim.

- Se defineşte tipul structurat persoana:

```
typedef struct {
        char *  nume;
        int varsta;
} persoana;
```

Structuri
Alocarea dinamică

```
int n;
 persoana * tab;


void citire(void) {
        int i;
        char buf[80];

        printf("Introduceti nr de persoane \n");
        scanf("%d", &n);
        if (!(tab=(persoana *)malloc(n*sizeof(persoana)))) {

            printf("Eroare alocare dinamica memorie \n");
            exit(1);
        }
```

Structuri

Alocarea dinamică

```
    for (i=0; i<n; i++) {
            printf("Introduceti numele persoanei");
            scanf("%s", buf);
            if (!(tab[i].nume=(char *)malloc(strlen(buf)+1))) {
               printf("Eroare alocare dinamica memorie \n");
               exit(1);
            }
                  strcpy(tab[i].nume, buf);
            printf("Introduceti varsta persoanei");
            scanf("%d", &tab[i].varsta);
    }
}
```

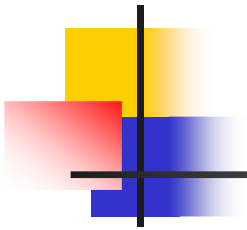
Aplicaţie

Pointeri. Alocare dinamică

- **Problemă:** Să se scrie un program care gestionează date despre un grup de studenți. Pentru fiecare student se memorează numele şi numărul matricol. Programul trebuie să implementeze următoarele operații:
  - citirea numărului de studenți şi a datelor acestora;
  - afişarea datelor tuturor studenților;
  - sortarea listei de studenți în ordinea alfabetică a numelor;
  - sortarea listei de studenți în ordinea crescătoare a numerelor matricole;
  - căutarea unui student pentru care se precizează numele şi afişarea poziției pe care o ocupă acesta în lista ordonată alfabetic după numele studenților;
  - căutarea unui student pentru care se precizează numărul matricol şi afişarea poziției pe care o ocupă acesta în lista ordonată crescător după numărul matricol al studenților;

Aplicaţii

Pointeri. Alocare dinamică

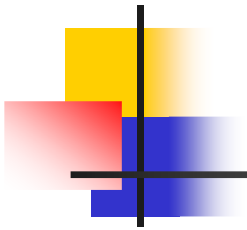


```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

typedef struct {
   char *nume;
   int nr;
} student;
```

Aplicaţii

Pointeri. Alocare dinamică



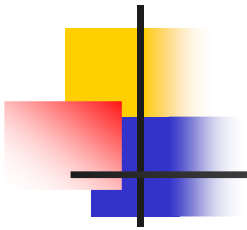
```
#define SIZE sizeof(student)

typedef student *pstud;

void eroare(void) {
   puts("\n  **** eroare alocare dinamica de memorie ****");
   exit(1);
}
```

Aplicaţii

Pointeri. Alocare dinamică



```
void citeste(int *n, pstud *tab) {
  pstud t;
  char sir[40];
  int i;
  printf("\n dati numarul studentilor:");
  scanf("%d", n);
  if(!(t=(pstud)malloc((*n)*SIZE)))  eroare();
  *tab=t;
```

Aplicaţii

Pointeri. Alocare dinamică

```
    for (i=0; i<*n; i++, t++)  {
        printf("\n nume: ");
        scanf("%s", sir);
        /* aloca dinamic spatiu pentru numele studentului */
        if (!(t->nume=(char *) malloc(strlen(sir)+1)))
              eroare ();
        strcpy(t->nume, sir);
        printf("\n numar matricol: ");
        scanf("%d", &t-> nr); }
}
```

Aplicaţii

Pointeri. Alocare dinamică

```
void afiseaza ( int n, pstud tab)
{
    int i;
    puts("\n tabelul cu studenti ");
    for (i=0; i<n; i++, tab++)
       printf("\n%-30s %4d", tab->nume, tab->nr);
}
```

Aplicaţii

Pointeri. Alocare dinamică

```
void elibereaza (pstud tabel, int n) {
   int i;

   for (i=0; i<n; i++)
       free(tabel[ i ].nume);
   free(tabel);
}
```

Aplicaţii

Pointeri. Alocare dinamică

```
void meniu(void)  {
   puts("\n c, C --- citeste tabel studenti");
   puts("\n a, A --- afiseaza tabel studenti");
   puts("\n n, N --- ordoneaza dupa nume");
   puts("\n r, R --- ordoneaza dupa numar matricol");
   puts("\n f, F --- cauta dupa nume");
   puts("\n l, L --- cauta dupa numar matricol");
   puts("\n x, X --- iesire din program");
 }
```

Aplicaţii

Pointeri. Alocare dinamică

```
void main(void)  {
    char opt;
    int n;   /* numarul de studenti  */
    char nume[30];
    student s;
    pstud tabel=NULL; /* adresa tabloului cu studenti */

    while (1)
    {
        meniu();
        opt=tolower(getchar());
```

Aplicaţii

Pointeri. Alocare dinamică

```
switch(opt)  {
  case ‘c’:
    if(tabel)/* daca a existat anterior un alt tabl. in mem.*/
         eliberează(tabel, n);
    citeste(&n, &tabel);  break;
  case ‘a’:
        afiseaza(n, tabel);  break;
  case ‘n’:
        sorteaza_alfabetic(tabel, n);   break;
  case ‘r’:
        sorteaza_dupa_nr_matricol(tabel, n);   break;
```

Aplicaţii

Pointeri. Alocare dinamică

```
case ‘f’:
        printf(“\n dati numele:”);
        scanf(“%s”, nume);
        if (!(s.nume=(char *)malloc(strlen(nume)+1)))
            eroare();
        strcpy(s.nume, nume);
        cauta_dupa_nume(&s, tabel, n);
        free(s.nume); break;
```

Aplicaţii

Pointeri. Alocare dinamică

```
    case ‘l’:
        printf(“\n dati numarul matricol:”);
        scanf(“%d”, &s.nr);
        cauta_dupa_nr_matricol(&s, tabel, n);  break;
     case ‘x’:
         exit(0);
     default:
         puts(“Comanda gresita”);  }
  }
}
```

Uniuni

- O uniune este o structură în care *toți membrii sunt memorați la aceeaşi adresă*. Aceasta înseamnă că, la un moment dat, într-o uniune poate exista doar un singur membru.

- Tipul de dată union a fost inventat pentru a preveni fragmentarea memoriei în bucăți de dimensiuni inutilizabile.

- Tipul de dată union previne fragmentarea prin crearea unei dimensiuni standard pentru anumite date. Spațiul alocat pentru uniune este egal cu dimensiunea elementului maxim al uniunii.

- Uniunile permit ca o porțiune de memorie să fie interpretată ca fiind tipuri de date diferite, întrucît toate aceste tipuri sunt de fapt în *aceeaşi locație de memorie dar la momente de timp diferite*.

Uniuni
Declarare

Declararea unei uniuni.

O uniune se declară în acelaşi mod ca şi o structură. Ea conţine o listă de *membri* de diferite tipuri.

union *nume_uniune {*
tip1 element1;
tip2 element2;
tip3 element3;
} nume_obiect;

- Declararea unei *variabile uniune* se face la fel ca şi declararea unei *variabile structură*.

- O uniune nu poate fi iniţializată decât cu o valoare de tipul primului său membru.

Uniuni
Utilizare

```
union int_sau_real
{
    int membru_int;
    float membru_float;
};
union int_sau_real uniune1, uniune2;
```

- Ca şi în cazul structurilor, membrii unei uniuni pot fi accesați cu operatorii **.** şi **->**.

- Totuşi, spre deosebire de structuri, variabilele uniune1 şi uniune2 pot fi tratate, în timpul execuției programului, *fie* ca variabile întregi, *fie* ca variabile reale în virgulă flotantă.

Uniuni
Utilizare

- De exemplu, dacă scriem:

```
uniune1.membru_int = 5;
```

atunci programul vede uniune1 ca fiind un întreg.

- Dacă apoi vom scrie:

```
uniune1.membru_float = 7.5;
```

valoarea întreagă a variabilei uniune1 se va pierde şi această variabilă va reprezenta în continuare un număr real.

- Concluzie: *o variabilă de tip union nu poate avea, la un moment dat, decât un singur tip: tipul membrului activ în acel moment.*

Uniuni
Utilizare

- O modalitate de a reține ce tip de valoare este memorată în *variabila uniune* este aceea de utiliza pentru fiecare uniune o *variabilă fanion*.

- Se recomandă ca tipul variabilei fanion să fie un *tip enumerare*.

- Exemplu: uniunii int_sau_real îi putem asocia tipul enumerare :

```
enum care_membru {
    INT,
    FLOAT
} ;
```

Uniuni
Exemple

- Variabilele de tip uniune şi de tip enumerare pot fi declarate în pereche:

```
union int_sau_real uniune1;
enum care_membru stare_uniune1;
```

- Exemplu:

```
switch (stare_uniune1)  {
    case INT:
      uniune1.membru_int += 5;  break;
    case FLOAT:
      uniune1.membru_float += 23.222333;  break;
}
```

Uniuni
Exemple

- În vederea unei mai uşoare utilizări, putem grupa aceste variabile într-o structură:

```
struct multitip
{
    enum care_membru stare;
    union int_sau_real numar;
};
struct multitip mt;
```

- Asignările membrilor structurii le facem în pereche:

```
mt.stare = INT;
mt.numar.membru_int = 5;
```

Uniuni
Exemple

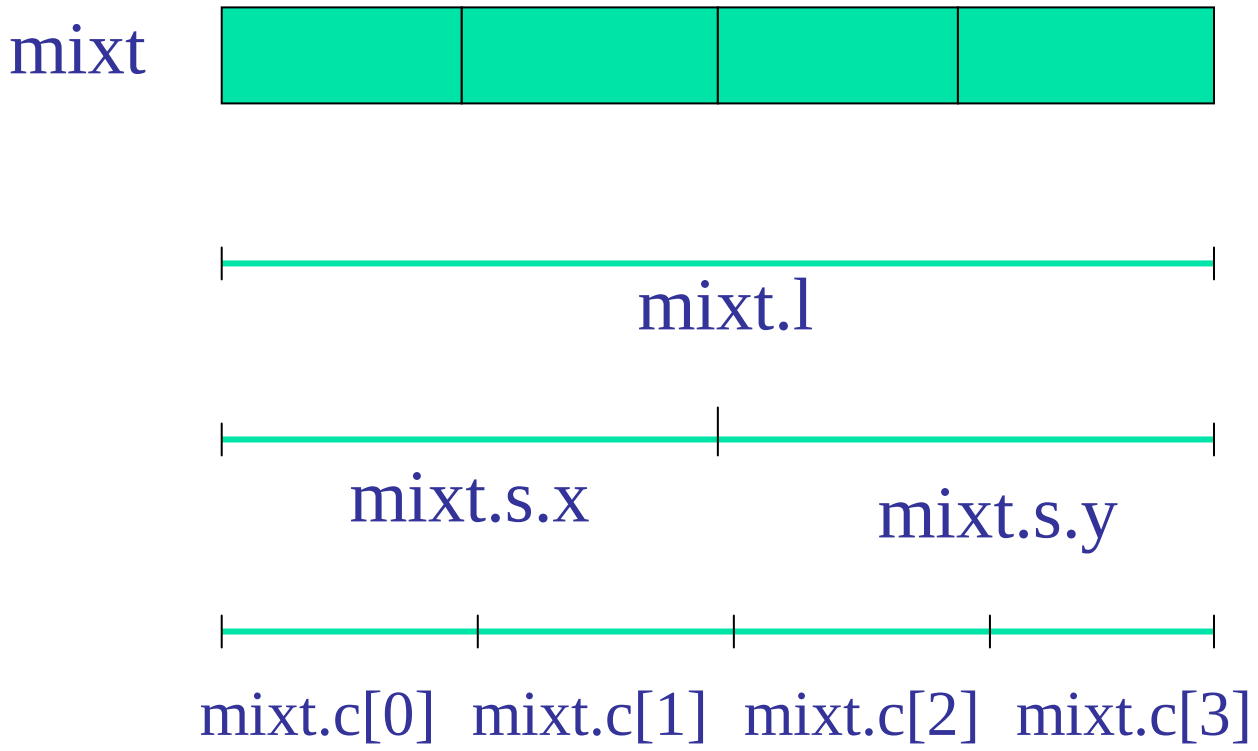
- Într-o uniune putem grupa tipuri elementare împreună cu tablouri sau structuri de elemente de dimensiuni mici.

```
union mixt_t  {
    long l;
    struct  {
        short x;
        short y;
     }  s;
     char c[4];
} mixt;
```

- Structura defineşte trei nume care ne permit să accesăm acelaşi grup de 4 octeți, ca şi tipuri long, short sau char.

Uniuni
Exemple

- Modalitățile de accesare a datelor sunt:



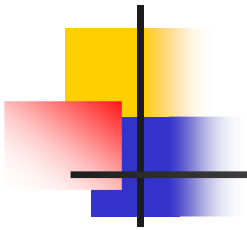
Câmpuri de biţi

- Printre alte facilități la nivel scăzut, C include şi operații pe biți. În acest sens este firească preocuparea de a include şi reprezentarea datelor pe biți.

- Limbajul C conține un mecanism de structurare a datelor la nivel de bit. Se pot aloca, unor câmpuri de structuri sau uniuni, biți individuali sau grupuri de biți dintr-un octet, numite *câmpuri de biți* care pot fi accesate individual în mod direct.

- Un *câmp de biți* este un şir de biți adiacenți conținuți într-un cuvânt calculator. Câmpurile de biți se grupează în structuri. Se pot defini câmpuri de biți doar ca şi **componente de tip întreg fără semn** ale structurilor. În declarația câmpului se specifică şi lungimea acestuia în biți.

Câmpuri de biţi



- *Exemplu*: Descrierea datelor meteorologice. Trebuie să se memoreze data (ziua şi luna), dacă în respectiva zi a plouat, dacă a fost soare, sau dacă a nins. Se alocă câte un bit pentru ploaie, soare şi respectiv ninsoare. Ziua şi luna pot fi reprezentate şi ele ca şi câmpuri de biți, deoarece au valori într-un interval redus.

```
struct meteo {
    unsigned int ziua : 5;
    unsigned int luna : 4;
    unsigned int ploaie :1;
    unsigned int soare :1;
    unsigned int ninsoare :1;
} m;
```

Câmpuri de biţi

- **Avantaj**: câmpurile de biți pot fi accesate în mod direct:

```
m.ziua=15;
m.luna=5;
m.ploaie=1;
m.soare=1;
m.ninsoare=0;
```

- Utilizarea câmpurilor de biți implică următoarele **restricții**:

  - aceştia trebuie definiți în cadrul unei structuri;
  - nu se pot defini tablouri de câmpuri de biți;
  - unui câmp de biți nu i se poate aplica operatorul de adresă.

- O structură poate conține atât câmpuri de date normale cât şi câmpuri de biți.