Programarea Calculatoarelor
Cursul 10

Structuri
Definiții de tip
Uniuni
Enumerări

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Obiective

- Ce este o structură?
  - Definiție, utilitate, sintaxă
  - Interacțiune cu funcții, tablouri, pointeri, alocare dinamică
- Cum se poate defini un tip de către utilizator?
- Ce este o uniune?
- Ce este o enumerare?

Structuri

- Definiție
  - O structură este o colecție de variabile înrudite și agregate sub un singur nume
  - Componentele se numesc și câmpuri = eng. fields
  - Este practic un tip compus din mai multe tipuri de bază
- Utilitate
  - Definire tip pentru a permite operații pe un anumit obiect
  - Este precursorul conceptului de obiect pentru limbaje orientate pe obiecte

Structuri - definiție

- Se definește folosind cuvântul cheie struct

```
struct nume_struct{
    tip1 camp1;
    tip2 camp2;
    …
    tipn campn;
};
```

se termină cu ;

- nume_struct este numele structurii, poate fi orice identificator valid
  - fără spațiu, nu începe cu un număr
- `tip1, tip2, … tipn` sunt tipuri și `camp1, camp2, … campn` sunt numele câmpurilor
  - se pot grupa împreună la fel ca la declararea variabilelor

Structuri declarare - exemple simple

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

struct complex{
    double re, im;
};

struct mystring{
    char* s;
    int len;
};

struct student{
    int nrmatricol;
    char nume[25];
    char cnp[15];
    float nota;
};
```

- punct cu coordonate întregi x și y
- număr complex cu parte reală și imaginară
- structură pentru stocarea unui string alocat dinamic cu un câmp pentru menținerea lungimii
- structură care reprezintă o înregistrare din catalog pentru un student
- o structură poate conține tablouri sau alte structuri, chiar și pointeri la structuri de același tip (definire recursivă)

Structuri - declarare variabile de tip structură

- Se poate face direct după definiție

```
struct nume_struct{
   tip1 camp1;
   tip2 camp2;
   …
   tipn campn;
} var1, var2, var3;
```

se termină cu ;

- Sau ulterior pe un rând separat, asemănător unei declarații de variabile de un tip fundamental dar împreună cu cuvântul cheie struct

```
struct nume_struct var1, var2, var3;
```

Definire de tipuri

- O structură poate fi privită ca un nou tip definit de utilizator
- În limbajul C este nevoie de o definiție de tip pentru a folosi o structură ca un tip
  - se recomandă fiindcă produce cod mai scurt și mai clar
- Un tip se definește folosind cuvântul cheie `typedef`

```
typedef tip_original nume_tip_nou;
```

- `tip_original` este numele tipului original
  - poate fi o structură
- `nume_tip_nou` este numele nou stabilit de programator

Definire de tipuri - exemple simple

```
typedef unsigned long size_t;
```

- definiție existentă în biblioteca standard în `crtdefs.h`

```
typedef long long ll;
```

- putem prescurta tipuri cu nume lungi în cod

```
typedef struct complex{
    double re, im;
}complex;
```

- numele acordat este identic cu numele original dar permite omiterea cuvântului cheie struct la declararea variabilelor

```
typedef struct mystring{
    char* s;
    int len;
}MyString;
```

- numele original este mystring, numele acordat este diferit = MyString

```
typedef struct{
    int nrmatricol;
    char nume[25];
    char cnp[15];
    float nota;
}student;
```

- numele structurii poate să lipsească
- apare doar numele acordat

Definire de tipuri + declarare variabile

- Definirea structurilor folosind definire de tip permite omiterea cuvântului cheie `struct` și folosirea structurii exact ca un tip nativ din C
- Definirea unei structuri nu ocupă memorie ci doar creează un tip nou de date

Definire de tipuri + declarare variabile - exemple simple

```
typedef long long ll;
ll x, y;
```

- putem prescurta tipuri cu nume lung în cod
- se folosește după definire exact ca un tip obișnuit

```
typedef struct{
    double re, im;
}complex;
complex z, w;
```

- structură anonimă doar cu nume acordat de `typedef`
- putem declara variabile fără a include cuvântul cheie `struct`

```
typedef struct student{
    int nrmatricol;
    char nume[25];
    char cnp[15];
    float nota;
    struct student* urmator;
}Student;
Student s;
```

- conține un pointer la aceeași structură
- aici suntem obligați să folosim cuvântul cheie `struct` și numele original
- dacă este aceeași structură atunci poate fi doar pointer

Inițializare structuri - exemple simple

- Se poate realiza folosind sintaxa cu inițializator {} folosită și la tablouri
  - se furnizează componentele în ordinea declarării lor

```
typedef struct{
    double re, im;
}complex;
complex z = {0, 1};
complex w = {-5};
```

- structură anonimă, doar cu nume acordat de typedef
- putem declara și inițializa variabile fără a include cuvântul cheie struct
- la fel ca la tablouri, câmpurile nespecificate sunt inițializate cu 0 dacă apare inițializatorul {}, altfel nu

```
typedef struct student{
    int nrmatricol;
    char nume[25];
    char cnp[15];
    float nota;
    struct student* urmator;
}Student;
Student s = {5, "Alin", "502...4", 10.f, NULL};
Student s2 = {6, "Alina", "603...5", 9.5f, &s};
```

- se inițializează în mod corect și componentele de tip string sau pointeri

Operații pe structuri

- Asignare - operatorul =
  - se copiază întregul conținut al structurii - include tablouri, pointeri
- Accesare câmpuri - operatorul .
  - se folosește după o variabilă de tip structură pentru a accesa un câmp specific
  - se poate folosi la inițializare pentru a seta doar un câmp specific
- Accesare câmpuri - operatorul ->
  - se folosește după o variabilă de tip pointer la o structură pentru a accesa un câmp specific
  - variantă prescurtată pentru dereferențiere și apoi operatorul .
- Adresă - operatorul &
  - preia adresa din memorie a structurii
  - se referă la adresa primului câmp din structură

Accesare câmpuri - exemple simple

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

typedef struct{
   double re, im;
}complex;
complex z = {.im = 1};

typedef struct student{
   int nrmatricol;
   char nume[25];
   char cnp[15];
   float nota;
   struct student* urmator;
}Student;

int main(){
   complex w;
   w.re = 1;
   w.im = -1;

   Student s;
   strcpy(s.nume, "Alin");
   Student s2 = {};
   s2.urmator = &s;
   return 0;
```

- structură anonimă doar cu nume acordat de `typedef`
- inițializator cu accesare câmp specific, restul câmpurilor este inițializat cu 0

- accesare și asignare câmpuri, variabila originală `w` este neinițializată

- *s* este neinițializat
- câmpurile se schimbă conform regulilor
- câmpurile lui *s2* sunt inițializate la 0
- se modifică câmpul *următor* care este de tip pointer la student

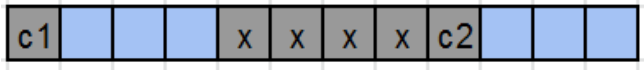
Dimensiunea unei structuri

- Definiția structurii nu alocă spațiu pentru structură
  - acest lucru se întâmplă în momentul în care declarăm o variabilă
- Se poate obține folosind operatorul `sizeof`
  - aplicat pe numele structurii sau pe o variabilă de tip struct
- Este aproximativ suma dimensiunilor componentelor
- Se urmărește alinierea datelor în memorie
  - Depinde de compilator și setări de împachetare
  - Fiecare `tip` începe de o adresă care este multiplu de sizeof(tip) relativ la inceputul structurii
  - dimensiunea finală a structurii este divizibilă cu tipul cel mai mare
  - o reordonare a câmpurilor poate rezulta în altă dimensiune
  - se recomandă definirea câmpurilor cu tipuri de dimensiune mică (char, short) unul lângă celălalt

Dimensiunea unei structuri - exemple simple

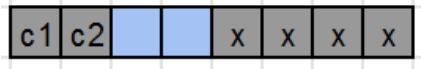
```c
#include <stdio.h>
typedef struct{
   char c1;
   int x;
   char c2;
} s1;
```

- după primul caracter se introduc 3 octeți de umplutură pentru a alinia câmpul x la o adresă care este multiplu de 4
- după al doilea caracter se introduc din nou 3 octeți de umplutură pentru a avea următoarea adresă liberă tot multiplu de 4



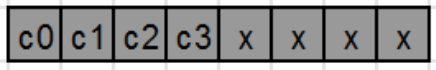

```c
typedef struct{
   char c1, c2;
   int x;
} s2;
```

- se introduc 2 octeți de umplutură după cele două caractere




```c
typedef struct{
   char c[4];
   int x;
} s3;
```

- spațiul pentru structură este folosit la maxim, 4 octeți pentru șirul de caractere și 4 octeți pentru int




```c
int main(){
   printf("%d\n", sizeof(s1));
   printf("%d\n", sizeof(s2));
   s3 var;
   printf("%d\n", sizeof(var));
   return 0;
}
```

- 12
- 8
- 8

Interacțiune structuri și funcții

- Se pot trimite și returna de la funcții
- La trimiterea unei structuri ca argument la o funcție
  - se trimite structura prin valoare
  - se copiază întregul conținut
    - dacă include tablouri sau string-uri ele se copiază în întregime
    - dacă include pointeri se copiază doar pointerul
  - pentru evitarea copierii se poate trimite structura ca pointer
    - evităm duplicarea unei zone extinse (tablouri mari)
- La returnarea unei structuri
  - se copiază întregul conținut

Tablouri de structuri

- Un tablou de structuri alocat static se declară ca

```
nume_struct nume_tablou[nr_elemente];
```

- Un tablou de structuri se aloca dinamic cu:

```
nume_struct* nume_tablou =
    calloc(nr_elemente, sizeof(nume_struct));
```

- unde `nume_struct` este un tip de structură definit prin `typedef`, `nr_elemente` poate să lipsească la tabloul static dacă se inițializează
- `nume_tablou` arată la adresa primului element de tip `nume_struct`
- elementele succesive ocupă poziții din memorie la incremente de `sizeof(nume_struct)`

Alocare dinamică structuri

- Se respectă regulile și recomandările prezentate la alocare dinamică
- O singură structură se alocă dinamic dacă
  - este de dimensiune mare
- Un tablou de structuri se alocă dinamic dacă
  - este de dimensiune mare - tipic mai mare ca 1000 elemente
  - este returnat de la o funcție - pentru a rămâne valabil
    - trebuie dezalocat după folosire
  - își schimbă dimensiunea în timpul programului
  - nu are dimensiune cunoscută la începutul programului
    - tablouri de structuri de dimensiune redusă se pot aloca pe stivă cu dimensiune variabilă cu `nume_struct a[n];`

Structuri cu câmpuri pe biți

- Câmp pe biți
  - câmp al unei structuri al cărui dimensiune este clar specificată în număr de biți sub forma unei constante după nume și caracterul :
  - permite economisirea spațiului ocupat de structură
  - se poate folosi doar cu tipuri întregi cu sau fără semn
  - nu se poate accesa adresa unui câmp

```
struct nume_struct{
    tip1 var1 : nbiti1;
    tip2 var2 : nbiti2;
    …
    tipn varn : nbitin;
}
```

Structuri cu câmpuri pe biți - exemplu

```c
#include <stdio.h>
typedef struct{
   unsigned int zi : 5;
   unsigned int luna : 4;
   unsigned int an : 12;
} date;

int main(){
   printf("%d\n", sizeof(date));
   date d;
   int zi;
   scanf("%d", &zi);
   d.zi = zi;
   d.luna = 12;
   d.an = 2020;

   printf("%u:%02u:%02u\n", d.an, d.luna, d.zi);
   return 0;
}
```

- zi pe 5 biți = 0 : 31
- luna pe 4 biți = 0 : 15
- an pe 12 biți = 0 : 4095

- 21 biți = 2 octeți + 5 biți - umplut la 4 octeți pentru aliniere
- nu se poate citi direct la adresa &d.zi
- afișăm și completăm cu 0-uri unde este nevoie

Uniuni

- O uniune este o zonă de memorie care poate conține o varietate de componente la momente diferite de timp
- Conține numai o singură componentă (membru) la un anumit moment
- Membrii unei uniuni partajează aceeași zonă de memorie
- Ajută la economisirea spațiului de memorie utilizat
- Poate fi accesat numai ultimul membru scris în acea uniune
- Zona de memorie rezervată are dimensiunea componentei care necesită cea mai multă memorie pentru reprezentare

Uniuni

- Definire
  - la fel ca o structură, dar folosind cuvântul cheie `union`

```
union nume_uniune{
    tip1 camp1;
    tip2 camp2;
    …
    tipn campn;
};
```

- Operațiile permise cu structuri sunt permise și cu uniuni
  - Excepție: la inițializarea unei uniuni doar primul membru poate fi inițializat

Uniuni - exemplu

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

union eterogen {
   int x;
   double y;
   char z[14];
} m;

int main(){
   union eterogen n={0x41424344};
   printf("%d %d\n",
          sizeof(n),sizeof(union eterogen));
   printf("%x %d\n",n.x,n.x);
   char* pc=(char*)&n;
   printf("%c%c%c%c%c\n",
          *pc,*(pc+1),*(pc+2),*(pc+3),*(pc+4));
   printf("%s\n",n.z);
   m=n;
   union eterogen* pm=&m;
   pm->y=7.50;
   strcpy(pm->z,"student");
   printf("%d %f %s\n",m.x,(*pm).y,pm->z);
   return 0;
```

- definiție uniune fără typedef
- *x* ocupă 4 octeți
- *y* ocupă 8 octeți
- *z* ocupă 14 octeți
- *m* este o variabilă globală

- se inițializează *x* cu numărul hexazecimal

- 16 - maximul dintre 4, 8, 14 rotunjit sus la cel mai apropiat multiplu de 8
- `41424344 1094861636` - câmpul *x* în hexazecimal și zecimal
- DCBA - luăm adresa lui *n* și interpretăm ca un pointer la caracter - bytes în ordin inversă
- DCBA - interpretat ca șir de caractere - există caracter nul deoarece am inițializat primii 4 bytes

- `1685419123 0.000000 student`
- $1685419123 = 's' + 't'*2^8 + 'u'*2^{16} + 'd'*2^{24}$
- string-ul interpretat ca double este un număr mic

Enumerări

- O enumerare conține un set de constante întregi reprezentate prin identificatori
- Permite folosirea unor nume sugestive pentru valori numerice
- Constantele sunt asemănătoare constantelor simbolice și au valori setate automat
- Valorile încep implicit de la 0 și sunt incrementate cu 1
- Se pot seta valori explicite prin asignare cu operatorul =
- Numele constantelor trebuie să fie unice
- Variabilele de tip enumerare își pot asuma doar una din valorile constante din set
- Nu se poate garanta că reprezentarea pe tipul întreg a unei variabile de tipul enumerare poate fi folosită pentru a stoca alt întreg

Enumerări

- Definire
  - folosind cuvântul cheie enum se enumeră numele fiecărui element posibil din enumerare
  - se începe numerotarea lor de la 0
  - valorile pot lipsi, în acest caz se continuă numerotarea lor

```
enum nume_uniune{
    nume0 = val0,
    nume1 = val1,
    …,
    numen = valn;
};
```

Enumerări - exemplu

```c
#include <stdio.h>

typedef enum {false, true} boolean;
enum culoare {alb, negru=14, verde,
     albastru, rosu=30};

int main(){
   enum culoare x=negru;
   enum culoare y=albastru;
   printf("%d %d %d %d %d\n",
         alb,negru,verde,albastru,rosu);

   boolean b = (0.3 == 0.1 + 0.2);
   printf("%d\n", b);
   b = (0.5 == 0.25 + 0.25);
   printf("%d %d\n", b, true);
   return 0;
}
```

- enumerare folosind `typedef`, false = 0, true = 1
- enumerare pentru culori, fiecare nume este asociat la un număr

- la declarare avem nevoie de `enum`

- 0 14 15 16 30

- nu avem nevoie de `enum`
- 0 - din cauza reprezentării inexacte a lui 0.3

- 1 1 - fiindcă numerele se reprezintă exact

Analiză problemă - aria unui poligon general

- Se dă un șir de *n* puncte 2D
- Să se determine aria poligonului format din punctele din șir, în ordinea dată
- Prima dată considerăm variante mai simple
  - aria unui triunghi format din puncte 2D
  - aria unui poligon convex

Analiză problemă - aria unui poligon general

- Aria unui triunghi 2D
  - formula lui Heron
  - formula cu determinant / produs scalar

$$\begin{vmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 \\ x_1 & x_2 - x_1 & x_3 - x_1 \\ y_1 & y_2 - y_1 & y_3 - y_1 \end{vmatrix}$$

  - determinantul este egal cu mărimea produsului vectorial între vectorii de la (x1, y1) la (x2, y2) și respectiv de la (x1, y1) la (x3, y3)
  - este egal cu +/- 2 *aria triunghiului
- Aria unui poligon
  - Orice poligon se poate descompune în triunghiuri
  - Aria poligonului este suma ariilor triunghiurilor

Analiză problemă - aria unui poligon general

- Formăm triunghiuri cu fiecare muchie din poligon și originea
- Fiecare triunghi contribuie la aria poligonului
  - cu semn pozitiv sau negativ, dat de semnul produsului vectorial
- Suma produsurilor vectoriale ale vectorilor consecutivi

$$\mathbf{A} = \frac{1}{2}\left|\sum_{i=1}^{n} x_i(y_{i+1} - y_{i-1})\right| = \frac{1}{2}\left|\sum_{i=1}^{n} y_i(x_{i+1} - x_{i-1})\right| = \frac{1}{2}\left|\sum_{i=1}^{n}(x_i y_{i+1} - x_{i+1} y_i)\right|$$

$$= \frac{1}{2}\left|\sum_{i=1}^{n}(x_{i+1} + x_i)(y_{i+1} - y_i)\right| = \frac{1}{2}\left|\sum_{i=1}^{n}\det\begin{pmatrix} x_i & x_{i+1} \\ y_i & y_{i+1} \end{pmatrix}\right|$$

https://en.wikipedia.org/wiki/Shoelace_formula

Aria unui poligon general - implementare

```c
#include <stdio.h>
#include <math.h>

typedef struct {
   double x, y;
} point;

double det(point p, point q){
   return p.x * q.y - p.y * q.x;
}

double area(point* v, int n){
   double area = 0;
   for(int i=0; i<n; i++){
       int j = (i+1)%n;
       area += det(v[i], v[j]);
   }
   return 0.5 * fabs(area);
}
```

```c
int main(){
   point v[] = {
           {0, 0},
           {0, 1},
           {1, 1},
           {1, 0}
   };
   int n = sizeof(v)/sizeof(point);
   printf("%f\n", area(v, n));
   return 0;
}
```

- tablou de puncte declarat și inițializat cu un pătrat de arie 1
- funcție pentru determinant
- fiecare pereche consecutivă
- indexare de la 0

- valoare absolută pentru numere flotante