Programarea Calculatoarelor

Cursul 10: Tipurile de date structură, uniune și enumerare. Nume simbolice pentru tipuri de date

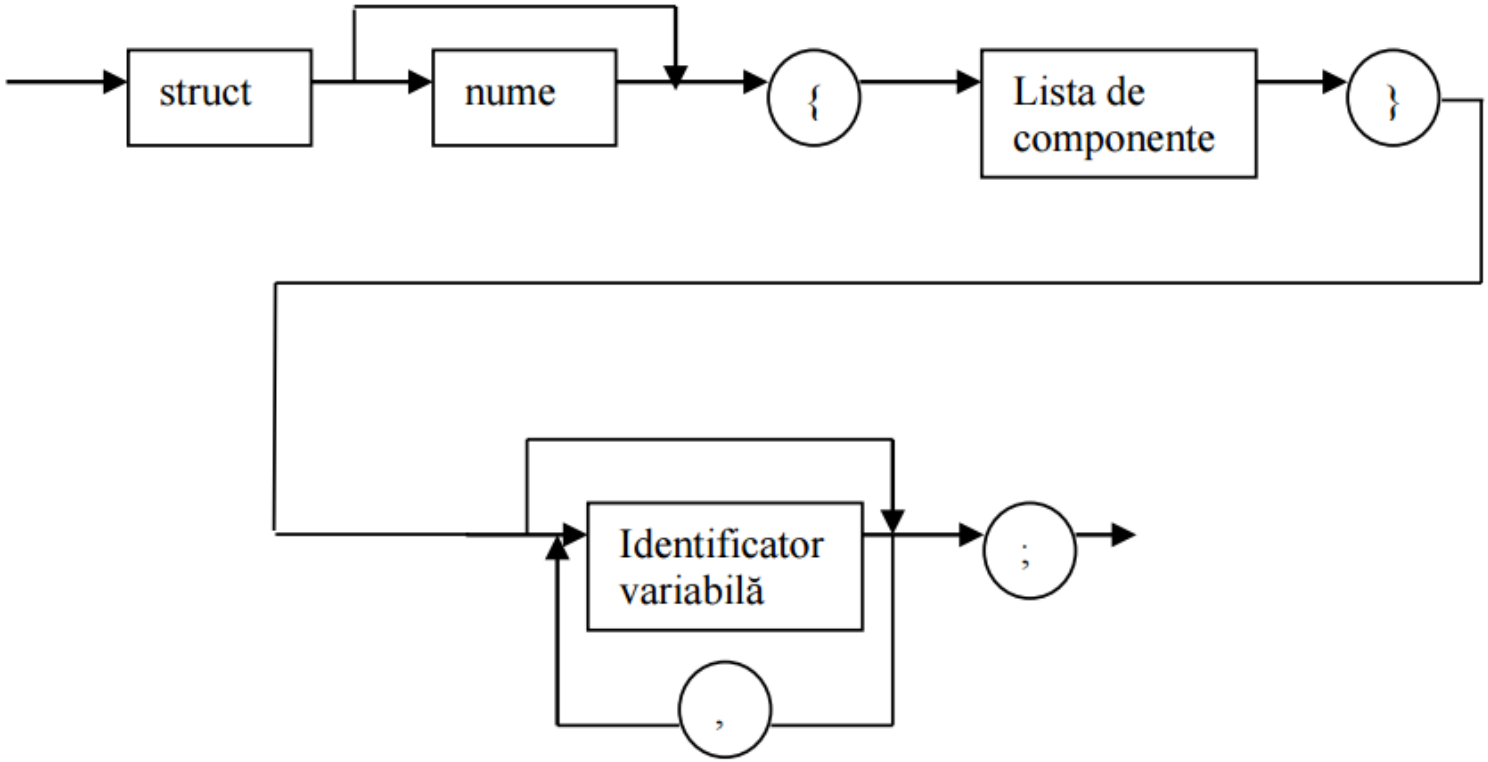
Razvan Itu
Universitatea Tehnică din Cluj-Napoca
Departamentul Calculatoare

Structuri

- Sunt colecții de componente (variabile) înrudite și agregate sub un singur nume
- Pot conține componente (variabile) având diferite tipuri de date
- Similare cu tipul de date **record** din limbajul PASCAL
- Frecvent utilizate pentru definirea de înregistrări care urmează a fi stocate în fișiere
- Combinate cu pointeri pot genera liste înlănțuite, stive, cozi, arbori, etc.
- Tipul de date structură poate fi declarat atât
  - Local – cu vizibilitate doar în acea funcție
  - Global – cu vizibilitate în toate funcțiile

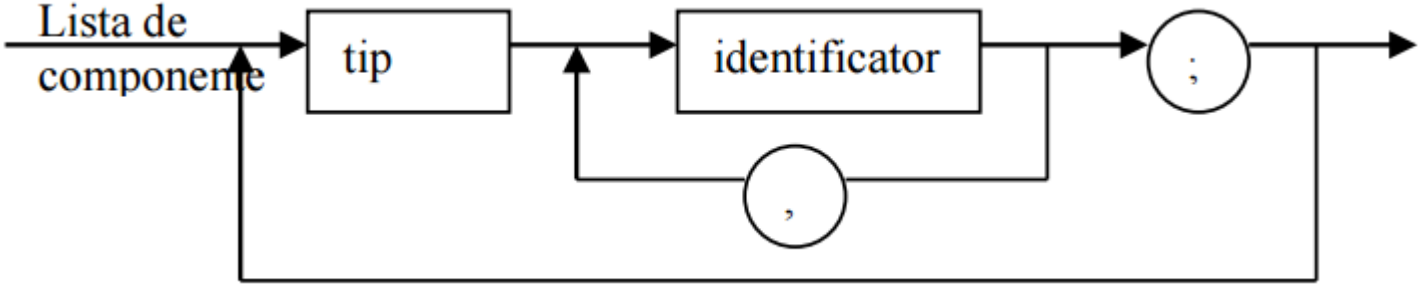
Definiția unei structuri

- În această definiție **nume** și **identificator variabilă** nu pot lipsi simultan:

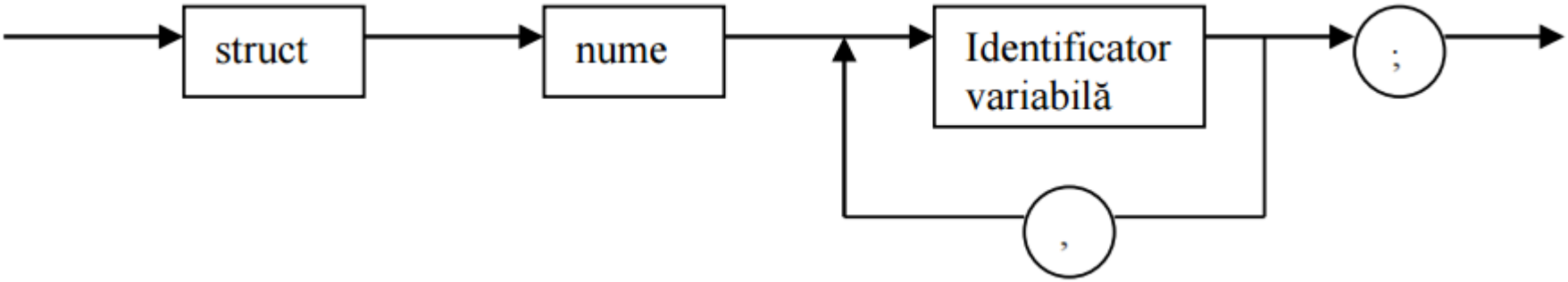


Lista de componente și variabile de tip structură

- Lista de componente poate conține unul sau mai mulți identificatori:



- O variabilă de tipul structurii se declară astfel:



Structuri - exemple

- Structura **student** cu variabile declarate (trei studenți)

```
struct student
{
   int numarmatricol;
   char nume[25];
   char CNP[14];
   float nota;
} a, b, c;
```

- Structura **student** și variabile declarate ulterior (trei studenți)

```
struct student
{
   int numarmatricol;
   char nume[25];
   char CNP[14];
   float nota;
};
struct student a, b, c;
student a, b, c; // In limbajul C++ poate lipsi cuvantul struct
```

Structuri - exemple

- Structură fără nume și trei variabile declarate

```
struct
{
   int numarmatricol;
   char nume[25];
   char CNP[14];
   float nota;
} a, b, c;
```

  - Observație: ulterior nu se mai pot declara alte variabile de tipul structurii
- Definirea unei structuri nu ocupă memorie ci doar creează un tip nou de date
  - Variabilele declarate de tipul structurii respective ocupă memorie
  - Dimensiunea memoriei ocupată de o astfel de variabilă este aproximativ suma dimensiunilor de memorie ocupată de fiecare componentă
    - Zona de memorie ocupată este în final aliniată (se introduc, dacă este necesar, octeți de umplutură – *padding bytes*) pentru a facilita accesul la date

Structuri

- O structură nu poate conține o componentă de tipul structurii înseși (definire recurentă)

```
struct student {
   int numarmatricol;
   char nume[25];
   char CNP[14];
   float nota;
   struct student s; // definire recurenta – nu este permisa!
};
```

- O structură poate conține o componentă care este pointer de tipul structurii înseși

```
struct student {
   int numarmatricol;
   char nume[25];
   char CNP[14];
   float nota;
   struct student *s; // pointer de tipul structurii
};
```

Structuri

- Definirea de structuri recursive prin intermediul pointerilor la structura însăși
  - Liste dinamice, arbori, etc.
  - Exemplu de structură de arbore binar de studenți

```
struct student {
   int numarmatricol;
   char nume[25];
   char CNP[14];
   float nota;
   struct student *fiustang; /* pointer catre studentul
                                fiu-stang */
   struct student *fiudrept; /* pointer catre studentul
                                fiu-drept */
};
```

Operații permise cu structuri

- Asignarea unei variabile structură la o altă variabilă structură de același tip
- Obținerea adresei unei variabile structură (**&**)
- Accesul la componentele (membrii) unei structuri
  - Calificare (operatorul **.**)
  - Utilizând dereferențierea pointerilor și calificare (operatorul **->**)
- Utilizarea operatorului **sizeof** pentru determinarea dimensiunii unei structuri
- Inițializarea componentelor unei variabile structură este similară cu inițializarea tablourilor
  - Membrii variabilelor globale sunt inițializați cu zero
  - Membrii variabilelor locale automate sunt neinițializați
- În C, ca și orice alt argument, trimiterea structurilor ca și argumente la apelul funcțiilor se face prin valoare

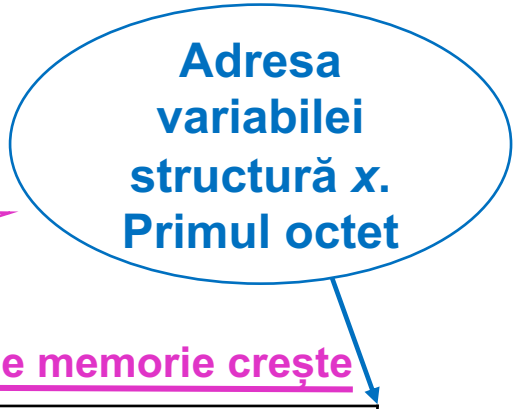
Operații permise cu structuri - exemplu

```
struct student {
        int numarmatricol;
        char nume[25];
        char CNP[14];
        float nota;
    } a;
struct student b;
struct student c={14526,"Popescu Alin","1960314121785",7.58f};
printf("%d %d\n",sizeof(b),sizeof(struct student)); //48 48
b=c;
a.numarmatricol=13154;
strcpy(a.nume,"Ionescu Emil");
strcpy(a.CNP,"1951201011143");
a.nota=5.54f;
struct student *pa=&a;
pa->nota=9.82f;
printf("%d %s %s %.2f\n",a.numarmatricol,a.nume,pa->CNP,(*pa).nota);
                                // 13154 Ionescu Emil 1951201011143 9.82
```

Memorarea componentelor unei variabile structură

- Componentele unei variabile de tipul unei structuri sunt alocate în memorie una după cealaltă
  - Sunt inserați, dacă sunt necesari, octeți de umplutură (*padding bytes*) pentru alinierea datelor în memorie
- *Little-endian* – definește ordinea de memorare a octeților unei valori: octetul mai puțin semnificativ este memorat primul, …, octetul cel mai semnificativ este memorat ultimul
- Exemplu de structură:

```
struct student {
        int numarmatricol;
        char nume[25];
        char CNP[14];
        float nota;
} x;
```

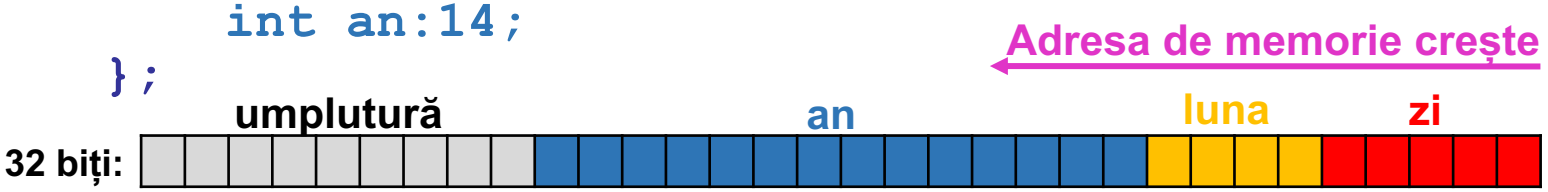


48 octeți: nota(4) umplutură(1) CNP(13…0) nume(24…0) numarmatricol(4)

Structuri cu câmpuri pe biți (*bit fields*) (1)

- Câmp pe biți
  - Membru al unei structuri, a cărui dimensiune este clar specificată, în număr de biți, ca și constantă întreagă după numele câmpului și caracterul `:`
  - Ajută la economisirea memoriei utilizate; trebuie să fie de tip întreg cu sau fără semn; nu pot fi accesați biții individual
- Exemplu de variabilă de structură cu trei câmpuri pe biți

```
struct data {
    unsigned int zi:5;
    unsigned int luna:4;
    int an:14;
};
```

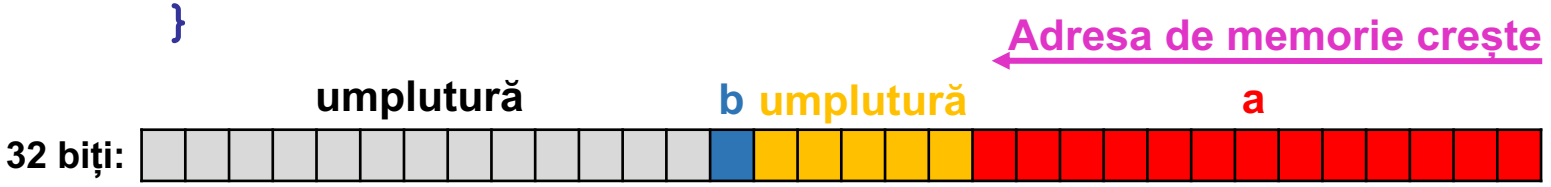


  - Dimensiunea memoriei ocupate de câmpuri este de 5+4+14=23 biți, însă `sizeof(struct data)` este 4 octeți (32 de biți) din motive de aliniere a datelor în memorie (9 biți de umplutură)
  - Observație: dacă toate cele trei câmpuri ar fi fost declarate de tip `int` sau `unsigned int` atunci `sizeof(struct data)` ar fi fost 12 octeți (96 de biți)

Structuri cu câmpuri pe biți (*bit fields*) (2)

- Câmpuri pe biți fără nume
  - Utilizați ca biți de umplutură (*padding bits*) în structură
  - Nu se poate stoca nimic în acești biți de umplutură
- Exemplu de variabilă de structură cu două câmpuri pe biți și alți biți de umplutură

```
struct bfn {
    unsigned int a:13;
    unsigned int :5; // biți de umplutură
    int b:1;
}
```



  - Dimensiunea memoriei ocupate de câmpuri este 13+5+1=19 biți, însă `sizeof(struct bfn)` este 4 octeți (32 de biți) din motive de aliniere a datelor în memorie (13 biți de umplutură)

Structuri cu câmpuri pe biți (*bit fields*) (3)

- Câmpuri pe biți fără nume și dimensiune zero
  - Aliniază conținutul curent de biți din structură (prin inserarea de biți de umplutură) pentru ca următorul câmp să înceapă într-o nouă unitate de memorie
  - Exemplu de variabilă de structură cu următoarele câmpuri

```
struct bfnzero {
    unsigned int a:13;
    int b:27;
    unsigned int :0; //inserează biți de umplutură
    int c:17;
    unsigned char d:2;
    char :0; // inserează biți de umplutură
    char e:3;
};
```

Adresa de memorie crește

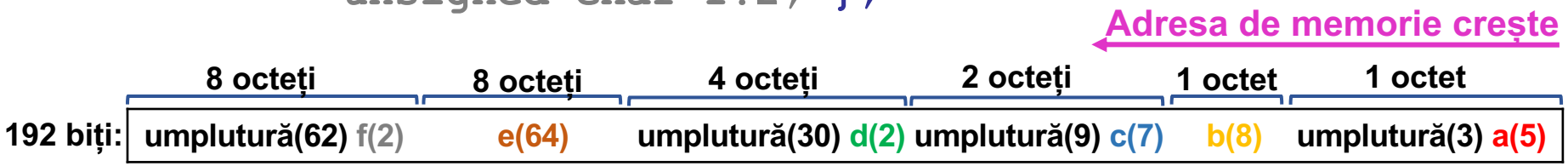
	3 octeți	1 octet	4 octeți	4 octeți	4 octeți
128 biți:	umplutură(21)	e(3)	umplutură(6)	d(2)	umplutură(15)	c(17)	umplutură(5)	b(27)	umplutură(19)	a(13)

  - Dimensiunea totală a memoriei ocupate de câmpurile utile este 62 de biți, însă `sizeof(struct bfnzero)` este 16 octeți (128 de biți) din motive de aliniere a datelor în memorie (biți de umplutură)

Structuri cu câmpuri pe biți (*bit fields*) (4)

- Structuri care conțin atât câmpuri obișnuite cât și câmpuri pe biți
  - Orice câmp obișnuit este aliniat automat la începutul unei noi unități de memorie (prin inserarea înainte a biților de umplutură)
  - Exemplu de variabilă de structură cu următoarele câmpuri

```
struct mixta {
    unsigned int a:5;
    char b;
    unsigned char c:7;
    unsigned int d:2;
    double e;
    unsigned char f:2; };
```

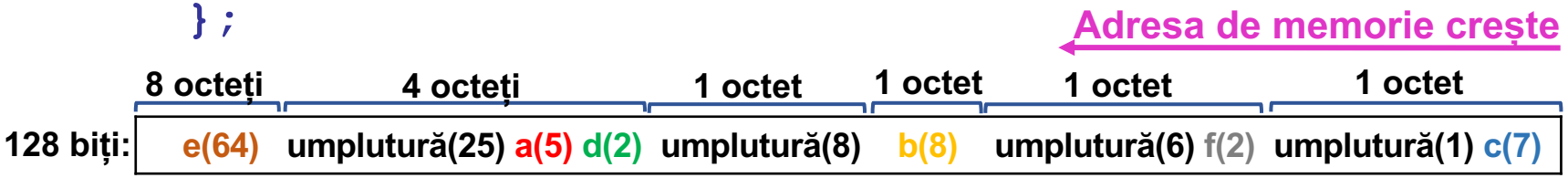


  - Dimensiunea totală a memoriei ocupate de câmpurile utile este 88 biți, însă `sizeof(struct mixta)` este 24 octeți (192 de biți) din motive de aliniere a datelor în memorie (biți de umplutură)
  - Observație: s-ar putea economisi memorie prin rearanjarea câmpurilor în structură

Structuri cu câmpuri pe biți (*bit fields*) (5)

- Structuri care conțin atât câmpuri obișnuite cât și câmpuri pe biți
  - Exemplul anterior cu câmpurile rearanjate

```
struct mixta {
    unsigned char c:7;
    unsigned char f:2;
    char b;
    unsigned int d:2;
    unsigned int a:5;
    double e;
};
```



  - În această configurație `sizeof(struct mixta)` este 16 octeți (128 de biți) din motive de aliniere a datelor în memorie (biți de umplutură)
  - S-a economisit memorie!

Structuri cu câmpuri pe biți (*bit fields*) (6)

- Accesul la câmpurile pe biți este analog ca și la câmpurile obișnuite
  - Singura diferență constă în faptul că nu se poate accesa adresa unui câmp pe biți
    - Este imposibil întrucât cea mai mică unitate de memorie accesibilă este octetul
    - Asignarea de valori se va face prin intermediul altei variabile
- Exemplu

```
struct mixta v;
/* scanf("%d", &v.d); => eroare de compilare */
int x;
scanf("%d",&x);
v.d = x;
```

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
- Definire – la fel ca și o structură, dar folosind **union**
- Operațiile permise cu structuri sunt permise și cu uniuni
  - Excepție: la inițializarea unei uniuni doar primul membru poate fi inițializat

Uniuni - exemplu

```
union heterogen {
      int x;
      double y;
      char z[14];
    } m;
union heterogen n={0x41424344};
printf("%d %d\n",sizeof(n),sizeof(union heterogen)); //16 16
printf("%x %d\n",n.x,n.x); //41424344 1094861636
char *pc=(char*)&n;
printf("%c%c%c%c%c\n",*pc,*(pc+1),*(pc+2),*(pc+3),*(pc+4)); //DCBA
printf("%s\n",n.z); //DCBA
m=n;
union heterogen *pm=&m;
pm->y=7.50;
strcpy(pm->z,"student");
printf("%d %f %s\n",m.x,(*pm).y,pm->z); //1685419123 0.000000 student
```

Little-endian

Adresa variabilei uniune *n*. Primul octet

Adresa de memorie crește

16 octeți:	12 octeți (inițializați cu 0)	4 octeți
	0x0 0x0 ..... 0x0	0x41 0x42 0x43 0x44

Enumerări

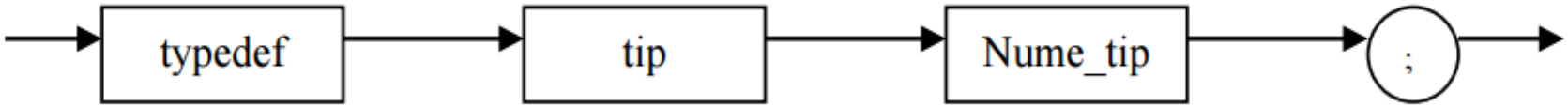
- O enumerare conține un set de constante întregi reprezentate prin identificatori
- Permite folosirea unor nume sugestive pentru valori numerice
- Constantele sunt asemănătoare constantelor simbolice și au valori setate automat
  - Valorile încep implicit de la 0 și sunt incrementate cu 1
  - Se pot 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 - exemplu

```
enum culoare {alb, negru=14, verde, albastru, rosu=30};
printf("%d %d %d %d %d\n",alb,negru,verde,albastru,rosu);
                                          // 0 14 15 16 30
enum culoare x=negru;
enum culoare y=albastru;
int z=x+y;
printf("%d %d %d\n",x,y,z); //14 16 30
x=alb;
x=40000; // nu garanteaza ca se poate stoca corect valoarea in x
printf("%d\n",x); //40000
```

Nume simbolice pentru tipuri de date

- Atribuie un nume simbolic unui tip predefinit sau unui tip utilizator
- Un nume de tip se definește utilizând cuvântul cheie **typedef**



- În cazul tipurilor de date structură, uniune și enumerare
  - Definirea unui nume de tip face ca specificarea cuvântului rezervat **struct**, **union** sau respectiv **enum** să nu mai fie necesară la declararea unei noi variabile

Nume simbolice pentru tipuri de date - exemplu

```
typedef int intreg;
typedef enum {false,true} boolean;
typedef struct {
    double real;
    double imag;
} complex;

intreg i=20;
complex k[2];
k[0].real=5.245;
k[0].imag=6.51;
k[1]=k[0];
boolean d;
d=(i>k[0].real+k[1].imag)?true:false;
printf("%d\n",d); //1
```