Programarea calculatoarelor
Limbajul C

CURS 9

Alocare dinamică

Structuri, enumerări, uniuni

Observație

- O funcție **nu** poate avea ca rezultat un vector sub forma:
  int [] funcție(…) {…}
- O funcție poate avea ca rezultat un pointer !!
  int *funcție(…) {…}
- De obicei, rezultatul pointer este egal cu unul din argumente, eventual modificat în funcție.

Exemplu corect:

```
// incrementare pointer p
char * incptr ( char * p) {
  return ++p;
}
```

- *Atenție! Acest pointer nu trebuie să conțină adresa unei variabile locale!*

Atenție!

- O variabilă locală are o existentă temporară, garantată numai pe durata executării funcției în care este definită (cu excepția variabilelor locale statice)
- Adresa unei astfel de variabile nu trebuie transmisă în afara funcției, pentru a fi folosită ulterior!!

Exemplu greşit:

```
  // vector cu cifrele unui nr intreg de maxim cinci cifre
int * cifre (int n) {
  int k, c[5];                    // vector local
  for (k=4;k>=0;k--) {
    c[k]=n%10; n=n/10;
  }
  return c;                       // aici este eroarea !
}
//warning la compilare şi rezultate greşite în main!!
```

Funcții cu rezultat pointer

- O funcție care trebuie să transmită ca rezultat un vector poate fi scrisă corect în mai multe feluri:

1. Primeşte ca argument adresa vectorului (*definit şi alocat în altă funcție*) şi depune rezultatele la adresa primită (este soluția recomandată!!)

```
void cifre (int n, int c[ ]) {
  int k;
  for (k=4;k>=0;k--) {
    c[k]=n%10; n=n/10;
  }
}
int main(){
      int a[10];
      ….
      cifre(n,a);
      ….
}
```

Funcții cu rezultat pointer

2. Alocă dinamic memoria pentru vector (cu "malloc")
- această alocare (pe heap) se menține şi la ieşirea din funcție.
- funcția are ca rezultat adresa vectorului alocat în cadrul funcției.
- problema este unde şi când se eliberează memoria alocată.

```
int * cifre (int n) {
  int k, *c;                          // vector local
  c = (int*) malloc (5*sizeof(int));
  for (k=4;k>=0;k--) {
    c[k]=n%10; n=n/10;
  }
  return c;                           // corect
}
```

3. O soluție oarecum echivalentă este utilizarea unui vector local static, care continuă să existe după terminarea funcției.

Exemplu: corect sau greşit?

```
    // extrage un subsir de lungime n de la adresa s
char * substr (char* s, int n) {
  char aux[1000];              // o variabila locala pentru subsirul rezultat
  int m;
  m=strlen(s);                 // cate caractere mai sunt la adresa s
  if (n>m) n=m;
  strncpy(aux,s,n);
  aux[n]=0;                    // terminator de (sub)sir
  return aux;                  // pointer la o variabila locala !
}
    // verificare funcție
void main () {
  char  s[]="abcdef";
  puts (substr(s,3));
  puts (substr (substr(s,4),2));
}
```

- char *aux= (char*) malloc(strlen(s)+1);
- char * aux=strdup(s);

Structură

- O structura este un tip de date care ne permite gruparea unor elemente eterogene.
- Este o colecție de una sau mai multe variabile (câmpuri), grupate sub un singur nume.
- Printr-o declarație *struct* se defineşte un nou tip de date de către utilizator.
- Definire C:

```
struct nume_structura {
  tip nume_camp_1;
  tip nume-camp_2;
  …
} [lista_variabile_structura];
```

Exemple

- structura moment de timp

```
struct time {
   int ora, min, sec;
};
```

- structura activitate

```
struct activ {
   char numeact[30];          // nume activitate
   struct time start;         // ora de incepere
   struct time stop;          // ora de terminare
};
```

Declarare variabile de tip structură

- Declararea unor variabile de un tip structură se poate face fie după declararea tipului structură, fie simultan cu declararea tipului structură.

- Exemple:
- struct time t1, t2, t [100]; // t este vector de structuri
- struct complex {

```
        float re, im;
} c1,c2,c3;
struct complex cv[200];
        // un vector de numere complexe
```

Initializare şi referire elemente structură

- struct data_calendaristica {
  int zi;
  int luna;
  int an;
  } d;

- iniţializare :

struct data_calendaristica azi = { 5, 4, 2006 };
struct complex c1= {1,-1}, c2= {2,3};

- Pentru referire se utilizează operatorul “.”

d.zi, d.luna, d.an

Observații

- In structuri diferite pot exista câmpuri cu acelaşi nume, dar într-o aceeaşi structură numele de câmpuri trebuie să fie diferite.
- Definirea tipului structură (cu sau fără *typedef*) se face la începutul fişierului sursă care conține funcțiile (înaintea primei funcții)
- Câmpurile unei variabile structură nu se pot folosi decât dacă numele câmpului este precedat de numele variabilei structură din care face parte, deoarece există un câmp cu acelasi nume în toate variabilele de un acelaşi tip structură!

  Nu pot spune *zi* ci *d.zi* !!

Observații

- Dacă un câmp este la rândul lui o structură, atunci numele unui câmp poate conține mai multe puncte ce separă numele variabilei şi câmpurile de care aparține (în ordine ierarhică)!

- Exemplu:

```
struct activ a;
printf (“%s începe la %d: %d şi se termina la %d: %d \n”, a.numeact, a.start.ora, a.start.min, a.stop.ora, a.stop.min);
```

Asocierea de sinonime pentru tipuri structuri

- Exemplu fără asociere sinonim:

```
struct Carte
{
    int val;
    char cul[20];
};
struct Carte c1, c2;
```

- Observație: într-un fişier CPP se poate utiliza şi fără a folosi cuvântul cheie *struct*!

```
Carte c1, c2
```

Asocierea de sinonime pentru tipuri structuri

- Se utilizează *typedef*
- Permite atribuirea unui nume oricărui tip
- Numele se poate folosi apoi la fel cu numele tipurilor predefinite ale limbajului.
- Sintaxa declarației *typedef* este la fel cu sintaxa unei declarații de variabilă, dar se declară un nume de tip şi nu un nume de variabilă!

Exemplu

```
typedef struct card
{
      int val;
      char cul[20];
} Carte;
Carte c1, c2;
c1. val = 10;
strcpy (c1. cul, “caro”);
c2 = c1;
```

- Cu typedef structura poate fi şi anonimă (poate lipsi cuvantul card)!

Asocierea de sinonime pentru tipuri structuri

- definire nume tip simultan cu definire tip structură:

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

- definire nume tip după definire tip structură:

```
typedef struct activ activitate;
```

Observație

- Se pot folosi ambele nume ale unui tip structură (cel precedat de *struct* şi cel dat prin *typedef*), care pot fi chiar identice:

- ```
  typedef struct complex  {
          float re; float im;
  } Complex;
  Complex c1;
  struct complex c2, c3;
  ```

- ```
  typedef struct point {
          double x, y;
  } point;
  struct point p[100];
  ```

Utilizarea tipurilor structură

- Un tip structură poate fi folosit în :
  - declararea de variabile structuri sau pointeri la structuri
  - declararea unor argumente formale de funcții (structuri sau pointeri la structuri)
  - declararea unor funcții cu rezultat de tip structură
- Operațiile posibile cu variabile de un tip structură sunt:
  - aplicarea operatorilor & şi sizeof
  - atribuirea între variabile de acelaşi tip structură
    - se foloseşte operatorul de atribuire “=“
    - nu se pot folosi alți operatori ai limbajului
    - trebuie definite funcții pentru operații cu structuri: comparații, operații aritmetice, operații de citire-scriere etc.
  - transmiterea ca argument efectiv la apelarea unei funcții.
  - transmiterea ca rezultat al unei funcții, într-o instrucțiune *return*.

Funcții cu rezultat de tip structură

- O funcție care produce un rezultat de un tip structură poate fi scrisă în două moduri:

1. Funcția are rezultat de tip structură:

```
// citire număr complex (varianta 1)
    complex readx () {
      complex c;
      scanf ( “%f%f ”, &c.re, &c.im);
      return c;
    }
     // utilizare
    complex a[100];
    . . .
    for (i=0;i<n;i++)  a[i]=readx();
```

Funcții cu rezultat de tip structură

2. Funcția depune rezultatul la adresa primită ca argument (pointer la tip structură, pentru a putea modifica):

```
// citire număr complex (varianta 2)
  void readx ( complex * px) {    // pointer la o structură complex
          scanf ( “%f%f ”, &(*px).re, &(*px).im);
  // sau scanf ( “%f%f ”, &px->re, &px->im);
  }
  // utilizare
  complex a[100];
   . . .
  for (i=0;i<n;i++)  readx (&a[i]);
  // adresa variabilei structură a[i]
```

Observații

- Notația px→re este echivalentă cu notația (*px).re
  - Câmpul “re” al structurii de la adresa px

- Pentru structurile care ocupă un număr mare de octeți este mai eficient să se transmită ca argument la funcții adresa structurii (un pointer) în loc să se copieze conținutul structurii la fiecare apel de funcție şi să se ocupe loc în stiva de variabile *auto*, chiar dacă funcția nu face nici o modificare în structura a cărei adresă o primeşte!

Exercitii - Structuri

- Să se defineasca o structura “time" care grupează 3 întregi ce reprezintă ora, minutul şi secunda pentru un moment de timp. Să se scrie funcții pentru:
  1. Verificare corectitudine ora
  2. Citire moment de timp
  3. Scriere moment de timp
  4. Comparare de structuri “time".
- Program pentru citirea şi ordonarea cronologică a unor momente de timp şi afişarea listei ordonate, folosind funcțiile anterioare.

Exemplu: Ordonarea unui vector de structuri “time”

```
#include<stdio.h>
typedef struct time {
    int ora, min, sec;
}time;

void wrtime ( time t) {                    // scrie ora,min,sec
     printf ("%02d:%02d:%02d \n", t.ora,t.min,t.sec);
}

int cmptime (time t1, time t2) {          // compara momente de timp
       int d;
       d=t1.ora - t2.ora;
       if (d) return d;
       d=t1.min - t2.min;                  // <0 daca t1<t2 şi >0 daca t1>t2
       if (d) return d;                    // rezultat negativ sau pozitiv
       return t1.sec - t2.sec;             // rezultat <0 sau =0 sau > 0
}
```

Exemplu - continuare

```
int corect (time t) {              // verifica daca timp plauzibil
     if ( t.ora < 0 || t.ora > 23 ) return 0;
       if ( t.min < 0 || t.min > 59 ) return 0;
       if ( t.sec < 0 || t.sec > 59 ) return 0;
       return 1;                   // plauzibil corect
}

time rdtime () {                   // citire ora
       time t;
       do {
             scanf ("%d%d%d", &t.ora, &t.min,&t.sec);
             if ( ! corect (t))
               printf ("Date gresite, repetati introducerea: \n");
             else break;
       } while (1);
       return t;
}
```

Exemplu - continuare

```
void sort (time a[], int n) {        // ordonare vector de date
      int i,gata; time aux;
      do {
        gata=1;
        for (i=0;i<n-1;i++)
              if (cmptime(a[i],a[i+1]) > 0 ) {
                        aux=a[i]; a[i]=a[i+1]; a[i+1]=aux; gata=0;
              }
      } while ( ! gata);
}
int main () {
       time t[30];
       int n,i;
       printf("introducere n: ");  scanf("%d",&n);
       for(i=0;i<n;i++)  t[i] = rdtime();
       sort (t, n);
       for ( i=0 ;i<n ;i++)   wrtime(t[i]);
       return 1;
}
```

Avantajele utilizării tipurilor structură

- Programele devin mai explicite dacă se folosesc structuri în locul unor variabile separate.
- Se pot defini tipuri de date specifice aplicației iar programul reflectă mai bine universul aplicației.
- Se poate reduce numărul de argumente al unor funcții prin gruparea lor în argumente de tipuri structură şi deci se simplifică utilizarea acelor funcții.
- Se pot utiliza structuri de date extensibile, formate din variabile structură alocate dinamic şi legate între ele prin pointeri (liste înlănțuite, arbori s.a).

Structuri predefinite

- “struct tm” definită în <time.h>
- are componente ce definesc complet un moment de timp:

```
struct tm {
  int     tm_sec, tm_min, tm_hour;  // secunda, minut, ora
  int     tm_mday,  tm_mon,  tm_year;   // zi, luna, an
  int     tm_wday, tm_yday;  // nr zi în saptamana şi în an
  int     tm_isdst; // 1=se modifica ora (iarna/vara), 0= nu
};
```

- Există funcții ce lucrează cu această structură: asctime, localtime, etc

Cum se poate afişa ora şi ziua curentă, folosind numai funcţii standard

```
#include <stdio.h>
#include <time.h>
int main(void) {
  time_t t;                    // time_t  este alt nume pentru  long
  struct tm *area;             // pentru rezultat funcție localtime
  t = time (NULL);             // obtine ora curenta
  area = localtime(&t);        // conversie din time_t în struct tm
  printf ("Local time is: %s", asctime(area));
}
```

Observaţie:

- asctime(struct tm* t) returnează un şir ce reprezintă ziua şi ora din strucura t. Şirul are următorul format: **DDD MMM dd hh:mm:ss YYYY**
- funcţia “time” transmite rezultatul şi prin numele funcţiei şi prin argument: long time (long*), deci se putea apela si: time (&t);

Structuri predefinite

- “struct stat” definită în fişierul <sys/stat.h>
- reuneşte date despre un fişier, cu excepția numelui

```
struct stat {
  short unix [7];      // fără semnificatie în sisteme Windows
  long st_size;        // dimensiune fisier (octeti)
  long st_atime, st_mtime;
            // ultimul acces / ultima modificare
  long st_ctime;      // data de creare
};
```

- Funcția “stat” completează o astfel de structură pentru un fişier cu nume dat:

  int stat (char* filename, struct stat * p);

Exemplu

- Pentru a afla dimensiunea unui fişier normal (care nu este fişier director) vom putea folosi funcția următoare:

```
long filesize (char * filename) {
   struct stat fileattr;
   if (stat (filename, &fileattr) < 0)   // daca fisier negasit
      return -1;
   else  // fisier gasit
      return (fileattr.st_size);
           // campul st_size contine lungimea
}
```

Uniunea (reuniunea) - union

- Defineşte un grup de variabile care nu se memorează simultan ci alternativ.
- Se pot memora diverse tipuri de date la o aceeaşi adresă de memorie.
- Alocarea de memorie se face (de către compilator) în funcție de variabila ce necesită maxim de memorie.

```
union {
   int ival;  long lval;  float fval;  double dval;
 } val;
```

Exemplu

- O uniune face parte de obicei dintr-o structură care mai conține şi un câmp discriminant, care specifică tipul datelor memorate (alternativa selectată la un moment dat).

```
typedef struct numar {
  char tipn;                        // tip numar (un caracter)
  union {
   int ival;  long lval;  float fval;  double dval;
  } v;
} numar;

void write (numar n) {
  switch (n.tipn) {
   case 'i': printf ("%d ", n.v.ival);break;
   case 'l': printf ("%ld ", n.v.lval);break;
   case 'f': printf ("%f ", n.v.fval);break;
   case 'd': printf ("%.15lf ", n.v.dval);
  }
}
```

Exemplu

- În locul construcției *union* se poate folosi o variabilă de tip *void** care va conține adresa unui număr, indiferent de tipul lui. Memoria pentru număr se va aloca dinamic:

```
typedef struct number {
  char tipn;          // tip numar
  void * pv;          // adresa numar
}number;

  // afisare numar
void write (number n) {
  switch (n.tipn) {
   case ‘i’: printf ("%d ", *(int*) n.pv);break;
   ...
   case ‘d’: printf ("%.15lf ",*(double*) n.pv);break;
  }
}
```

Enumerări

- Folosite pentru a da nume simbolice unui şir de valori numerice.
- Sintaxa:

  enum nume_opt { lista_constante } lista_variabile_opt ;
- Observație: constantele pot avea specificate valori (şi o valoare se poate repeta)
- Exemplu:

  enum luni_curs {ian=1, feb, mar, apr, mai, iun, oct=10, nov, dec};
- Implicit, şirul valorilor e crescător cu pasul 1, iar prima valoare e 0
- Un nume de constantă nu poate fi folosit în mai multe enumerări

Enumerări

- Tipurile enumerare sunt tipuri întregi
- Variabilele enumerare se pot folosi la fel cu variabilele întregi
- Cod mai lizibil decât prin declararea separată de constante:
- enum {D, L, Ma, Mc, J, V, S} zi;

/* tip anonim; declară doar var.zi, tipul nu are nume, deci nu mai putem declara altundeva variabile */

sau:

- typedef enum {D, L, Ma, Mc, J, V, S} Zi;

// tip enumerare cu numele Zi; se pot declara variabile

```
int nr_ore_lucru[7]; /* număr de ore pe zi */
for (zi = L; zi <= V; ++zi) nr_ore_lucru[zi] = 8;
```

Exemplu

- Să se scrie un program care declară o strucură pentru un număr generic, folosind o uniune pentru valoarea numărului şi un câmp tip ce va spune ce fel de număr este (întreg – i, întreg lung – l, real –f, real dubla precizie – d).
- Să se scrie o funcție ce citeşte o variabilă de tipul structurii definite
- Să se scrie o funcție ce afişează valoarea unei variabile de tipul structurii definite
- Pentru câmpul discriminant se poate defini un tip enumerare, împreună cu valorile constante (simbolice) pe care le poate avea. Să se refacă programul!

Rezolvare

```
#include<stdio.h>
  // structura pentru un număr de orice tip
typedef struct{
  char tipn;            // tip număr (un caracter)
  union {
   int ival;  long lval;  float fval;  double dval;
  } v; // valoarea numărului
} număr;

// afisare număr
void write (număr n) {
  switch (n.tipn) {
   case 'i': printf ("întreg %d ",n.v.ival);break;
   case 'l': printf ("întreg lung %ld ",n.v.lval);break;
   case 'f': printf ("Real %.5f ",n.v.fval);break;
   case 'd': printf ("Real dublu %.15lf ",n.v.dval);
  }
}
```

Rezolvare

```
număr read (char tip) {
  număr n;
  n.tipn=tip;
  switch (tip) {
   case 'i': scanf ("%d", &n.v.ival); break;
   case 'l': scanf ("%ld", &n.v.lval); break;
   case 'f': scanf ("%f", &n.v.fval); break;
   case 'd': scanf ("%lf", &n.v.dval);
   }
  return n;
}

int main () {
      număr a, b, c, d;
      a = read('i'); b=read('l'); c = read('f'); d=read('d');
      write(a); write(b); write(c); write(d);
     return 0;
}
```

Rezolvare 2

```
#include<stdio.h>
enum tnum {I,L,F,D} ; // definire tip “tnum”
typedef struct{
  tnum tipn;          // tip număr (un caracter)
  union {
   int ival;  long lval;  float fval;  double dval;
  } v; // valoarea numărului
} număr;

void write (număr n) {
  switch (n.tipn) {
   case I : printf ("%d ",n.v.ival); break;                    // int
   case L: printf ("%ld ",n.v.lval);break;                     // long
   case F: printf ("%f ",n.v.fval);break;                      // float
   case D: printf ("%.15lf ",n.v.dval);                        // double
  }
}
```

Rezolvare 2

```
număr read (tnum tip) {
  număr n;
  n.tipn=tip;
  switch (tip) {
       case I: scanf ("%d", &n.v.ival); break;
          . . .
  }
  return n;
}

int main () {
      număr a,b,c,d;
      a = read(I);
      write(a);
       …
}
```