Curs SDA (PC2)

Curs 3
Pointeri (continuare)
Structuri de date

Iulian Năstac

Pointeri (Recapitulare)

- **Un pointer este o variabilă care conţine o adresă din memorie, unde se află valoarea altei variabile.**

2. Declararea pointerilor (Recapitulare)

Declararea variabilei de tip pointer se face respectând următorul format:

tip *nume_pointer;

Deoarece declaraţia uzuală a unei variabile este:

tip nume;

putem spune că *tip** dintr-o declaraţie de pointer reprezintă *tip* dintr-o declaraţie obişnuită. Prin urmare *tip** va fi asociat unui tip nou, *tipul pointer*.

3. Operatori pentru pointeri (recapitulare)

Asupra pointerilor pot fi executate un număr variat de operaţii, însă există doi operatori speciali pentru pointeri întâlniţi în expresii (unare) de tipul:

operator *operand*

Aceşti doi operatori sunt:

- ***operatorul de indirectare*** (*) → expresia care-l urmează este un pointer iar rezultatul este o *lvaloare.*

- ***operatorul de obţinere a unui pointer*** (&) → operandul, asupra căruia acţionează acest operator, este o lvaloare iar rezultatul este un pointer.

4. Expresii cu pointeri (recapitulare)

4.1. Instrucţiuni de atribuire pentru pointeri

Unui pointer i se poate atribui:

- o adresă;
- un pointer.

Ca observaţie, specificatorul de format folosit în funcţii de ieşire (cum ar fi *printf*) pentru afişarea valorii unui pointer (adrese) este %p.

4.2. Aritmetica pointerilor
(recapitulare)

- Operaţiile aritmetice ce se pot efectua cu ajutorul pointerilor sunt adunarea şi scăderea, la care se adaugă cele asociate de incrementare si respectiv decrementare.

- Trebuie precizat că aritmetica pointerilor este relativă la tipul lor de bază.
- Ceilalţi operatori aritmetici (exceptând +, ++, –, şi - -) sunt interzişi în operaţii ce vizează pointeri (adrese de memorie).
- Nu se pot aduna sau scădea variabile sau constante de tip *float* sau *double* din pointeri (adică numere cu virgulă).

4.3. Utilizarea unui pointer cu mai multe tipuri de date (recapitulare)

- Există cazuri în care acelaşi pointer se poate utiliza pentru mai multe tipuri de date, în acelaşi program, însă nu simultan. Acest lucru se poate realiza declarând iniţial pointerul ca fiind de tip *void*:

  *void *nume;*

Conversia de tip *cast* respectă următorul format:

- *(tip) operand*
- *Exemplu:* *(int *)p=10;*
- *Regula conversiilor implicite pentru variabile.*

Regula conversiilor implicite

Acţionează când un operator binar se aplică la 2 operanzi.

Paşii regulii:

- Mai întâi se convertesc operanzii de tip char şi enum la tipul int;
- Dacă operatorul curent se aplică la operanzi de acelaşi tip atunci rezultatul va fi de acelaşi tip. Dacă rezultatul reprezintă o valoare în afara limitelor tipului respectiv atunci rezultatul este eronat (are loc o depăşire).
- Dacă operatorul binar se aplică la operanzi diferiţi atunci se face o conversie înainte de execuţia operatorului conform paşilor următori:

- Dacă un operand este long double rezultă că şi celălalt se converteşte spre long double şi rezultatul este de tip long double.
- Altfel, dacă unul dintre operanzi este de tip double şi celălalt se converteşte spre double şi rezultatul este de tip double .
- Altfel, dacă unul dintre operanzi este de tip float rezultă că celălalt este tot float, iar rezultatul este float.
- Altfel, dacă unul dinre operanzi este unsigned long rezultă că şi celălalt se converteşte la unsigned long, iar rezultatul este de tip unsigned long.
- Altfel, dacă unul dintre operanzi este de tip long rezultă că şi celălalt se converteşte la tipul long, iar rezultatul operaţiei este tot de tip long.
- Altfel, dacă unul dintre operanzi este de tip unsigned şi celălalt int, rezultă automat că ambii devin unsigned, iar rezultatul este unsigned.

4.4. Compararea pointerilor (recapitulare)

- Se pot compara doi pointeri printr-o expresie relaţională. Această comparaţie are totuşi justificare numai dacă cei doi pointeri pointează către elementele aceluiaşi tablou (matrice).
- Altfel, nu are relevanţă efectul comparaţiei atât timp cât compilatorul plasează variabilele distincte la adrese diferite de memorie funcţie de disponibilităţile de moment ale maşinii de calcul.
- Exemplu arătat în cursul trecut: stiva (cu vector)

Exemplu:

- Generăm o listă condusă pe principiul „ultimul venit primul plecat” (sau LIFO – Last Input First Output). Aceasta este o stivă care va stoca şi furniza valori întregi funcţie de numerele introduse. Astfel, dacă se introduce:
  - O valoare diferită de 0 sau -1 → aceasta se plasează în vârful stivei;
  - 0 → se scoate o valoare din stivă;
  - -1 → se opreşte programul.

```
#include<stdio.h>
#include<stdlib.h>
#define MARIME 50
void pune(int i);
int scoate(void);
int *b, *p, stiva[MARIME];
int main()
{       int valoare;
        b=stiva;  /*b indică baza stivei*/
        p=stiva;   /*iniţializează p*/
        do
          {printf("\n Introduceti valoarea:");
           scanf("%d", & valoare);
           if(valoare!=0)    pune(valoare) ;
                    else printf("\n Valoarea din varf este %d\n",
                                                scoate());
         }while(valoare !=-1) ;
}
```

```
void pune(int i)
{
        p++;
        if(p==(b+MARIME)) /* sau >=  */
          {printf("\nStiva
            supraincarcata");
           exit (1);
          }
        *p = i;
}
```

```
int scoate(void)
{ if(p==b) /* sau <=  */
    {printf("\n Stiva vida" );
     exit(1) ;
     }
   p - - ;
   return *(p+1) ;
}
```

5. Pointeri şi matrici

- Legătura dintre pointeri şi matrici a fost discutată pe larg în capitolul anterior (Utilizarea matricelor). Limbajul C furnizează două metode de acces la elementele unei matrice:
  - ***aritmetica pointerilor*** (mai rapidă);
  - ***indicii matricilor*** (mai lentă, dar apropiată de formalismul matematic).

Reamintire curs precedent

De exemplu fiind dată matricea:

int mat[lim_1][lim_2][lim_3]...[lim_n];

atunci elementul ***mat[i_1][i_2]...[i_n]*** este echivalent cu:

****(mat + i_1·lim_2·lim_3·...·lim_n +***
i_2·lim_3·lim_4·...·lim_n + ... + i_(n-1)·lim_n +
i_n)

Exemple:

1) În cadrul unui program ce conţine codul:

```
…
char sir[100], *p;
p=sir;
…
```

sunt similare (echivalente) expresiile de tipul ***sir[4]*** şi ****(p+4)*** .

Deoarece viteza este un criteriu în programare, programatorii experimentaţi în C/C++ utilizează de obicei pointerii pentru a avea acces la elementele matricei.

2) Se scriu două variante ale unei funcţii ce afişează un şir de caractere.

```
void scrie_sir(char s[ ])
{     register int t;
      for(t=0; s[t]; t++)       putchar(s[t] ) ;
}
```

sau:

```
void scrie_sir(char *s)
{     while(*s)     putchar(*s++);
}
```

5.1. Matrici de pointeri

Pointerii pot fi organizaţi în matrice ca orice alt tip de date.

Exemple:
1) Într-un program ce conţine matricea

int *x[10];

prin declaraţia :

x[2]=&var;

se atribuie adresa variabilei *var* elementului al treilea al matricei de pointeri.
Cu ajutorul

****x[2]***

se obţine valoarea lui *var*.

2) Matricile de pointeri se pot folosi ca argument de funcţie:

```
void afis_matrice(int *q[ ])
{     int t;
      for(t=0; t<10; t++)
      printf(„%d”, *q[t]) ;
}
```

În acest exemplu, *q* nu este un pointer către o variabilă de tip întreg ci un pointer către o matrice de pointeri către întregi. De aceea s-a folosit **q[ ]* în argumentul funcţiei, pentru a evita confuziile.

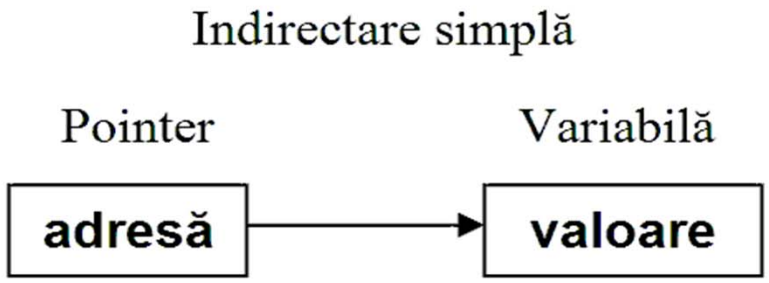
3) Matricile de pointeri sunt adeseori utilizate pentru a păstra pointerii către şiruri, ca în exemplul:

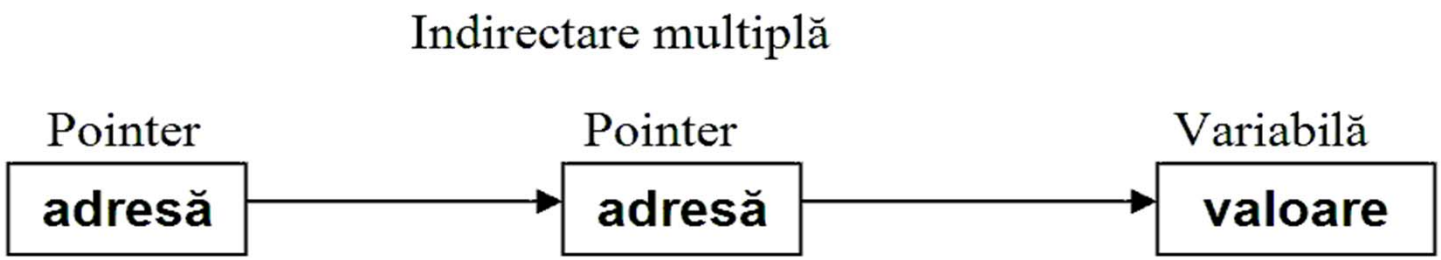
```
void eroare_afisare(int num)
{       static char *err[ ]={   “Nu se poate
                                        deschide fişierul”,
                                “ Eroare de citire”,
                                “ Eroare de
                                        scriere\n” };
        printf(„%s”, err[num]) ;
}
```

Matricea *err* păstrează pointeri către fiecare mesaj de eroare.

6. Indirectare multiplă

- Un pointer poate conţine adresa altui pointer care la rândul său indică valoarea ţintă. Această construcţie se numeşte ***indirectarea multiplă*** sau *pointer către pointer*.





Indirectarea multiplă poate fi continuată oricât de mult se doreşte (dar mai mult de un pointer către un alt pointer este rareori necesar).

Observaţii:

1) Indirectarea multiplă se cere folosită cu atenţie deoarece indirectarea excesivă este derutantă şi sursă de erori conceptuale.
2) Nu trebuie confundată indirectarea multiplă cu structurile de date de nivel înalt cum ar fi listele înlănţuite, care conţin pointeri. Cele două concepte se deosebesc fundamental.

O variabilă care este pointer către pointer se declară :

tip **nume ;

Exemplu:

```
# include<stdio.h>
  int main()
     {  int x, *p, **q;
        ...
        x=10;
        p=&x;
        q=&p;
        printf("%d", **q) ; /*afişează valoarea lui x*/
        ...
      }
```

7. Iniţializarea pointerilor

- Odată declarat un pointer, dacă acestuia nu i se atribuie o valoare, el poate să conţină o valoare necunoscută.

- O convenţie (respectată de multe compilatoare în C/C++) precizează că unui pointer care nu indică efectiv o locaţie de memorie validă i se dă valoarea null (zero). Totuşi faptul că un pointer are valoarea null nu îl face sigur.

- Majoritatea programatorilor în C/C++ iniţializează de obicei pointerii chiar la declarare pentru a evita ulterioare neplăceri.

Exemplu:

Un program care afişează un şir de caractere în ambele sensuri.

```
#include<stdio.h>
#include<string.h>
char *p="Limbajul de programare C";
int main()
    {   register int t;
        printf("%s", p); /*afişează şirul în mod direct*/

        for(t=strlen(p)-1; t>=0; t--) /* se afişează invers şirul*/
                printf("%c", p[t]); /* echivalent cu
                                        putchar(p[t])*/
        ...
    }
```

Atenție:

- Pentru a înțelege funcțiile de tratarea a șirurilor de caractere, recomandăm a se vedea notele de curs de la Programarea Calculatoarelor (din semestrul anterior):

http://www.euroqual.pub.ro/cursuri/programarea-calculatoarelor/

8. Pointeri către funcţii

- Folosirea pointerilor către funcţii este o caracteristică importantă a limbajului C, dar utilizată incorect este generatoare de confuzii.

- Pe măsură ce este compilată o funcţie, codul sursă este transformat în cod obiect şi se stabileşte un punct (adresă) de intrare în funcţie.

- La rularea programului, atunci când este apelată funcţia, acest punct de intrare este utilizat de limbajul maşină. De aceea, dacă un pointer conţine adresa punctului de intrare, el poate fi folosit pentru a apela acea funcţie.

- Adresa unei funcţii se obţine utilizând numele funcţiei fără nici o paranteză sau argumente.

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

void cauta(char *a, char *b, int (*comp)(const char *, const char * ));

int main()
    {   char s1[80], s2[80];
        int (*p)( const char *, const char *);
        p=strcmp;
        .....
        gets(s1);
        gets(s2);
        cauta(s1,s2,p);
       ....
     }

void cauta(char *a, char *b, int (*comp)(const char *, const char * ))
    {   printf("\n testeaza egalitatea \n" );
        if (!(*comp)(a,b)) printf("\n egal ");
        else printf("\n diferit");
     }
```

Observație:

- Există mai multe funcţii utilizate pentru compararea şirurilor de caractere. Una dintre cele mai utilizate este funcţia *strcmp* care are formatul:

int strcmp(const char *s1, const char *s2);

Observație: avem șirurile **s1** și **s2** (reamintire Semestrul I)

- **s1 = s2** – dacă au lungimi similare și s1[i] = s2[i] ∀ i.

- **s1 < s2** – dacă ∃ ***i*** astfel încât *s1[i]<s2[i]* și *s1[j]=s2[j]* ∀ j= 0, 1, …, i-1.

- **s1 > s2** – dacă ∃ ***i*** astfel încât *s1[i]>s2[i]* și *s1[j]=s2[j]* ∀ j= 0, 1, …, i-1.

Putem face o asociere mentală cu ordonarea cuvintelor într-un dicționar ...

Funcția strcmp
(reamintire Semestrul I)

int strcmp(const char *s1, const char *s2);

- Această funcție returnează:
  - valoare negativă - dacă s1 < s2
  - 0 - dacă s1 = s2
  - valoare pozitivă - daca s1 > s2

A se vedea Curs 11 de la:

http://www.euroqual.pub.ro/cursuri/programarea-calculatoarelor/

Observaţii:

- În exemplu anterior, argumentele funcţiei *cauta()* sunt doi pointeri de tip caracter şi unul către o funcţiei.
- Se observă că pointerul către funcţie introduce un grad semnificativ de confuzie fără a creşte semnificativ eficienţa.
- Tehnica este totuşi avantajoasă când se transmit mai multe funcţii ca parametrii sau se creează o matrice de funcţii.
- De exemplu în locul unei mari instrucţiuni *swich* cu o mulţime de funcţii listate în ea, poate fi creată o matrice de pointeri pentru funcţii (fiind mai uşor de selectat funcţia adecvată).

9. Apelarea prin referinţă a unei funcţii

O funcţie poate fi apelată prin :

- ***valoare*** (variabila folosită ca argument la apelare nu se modifica deoarece valoarea ei este copiată şi se operează cu această copie);

- ***referinţă*** (se lucrează cu un pointer deci cu o adresă, iar conţinutul locaţiei de memorie dat de acel pointer poate fi modificat de funcţie).

10. Funcţii pentru alocarea dinamică a memoriei

- Alocarea dinamică este caracteristica prin care un program poate obţine memorie în timpul rulării.

- În limbajul C, variabilelor globale li se alocă memorie în timpul compilării, iar cele locale folosesc memoria stivă. Nici variabilele locale nici cele globale nu pot fi adăugate în timpul execuţiei programului.

- Există însă cazuri în care memoria necesară unui program nu poate fi cunoscută dinainte (de exemplu un procesor de texte sau o bază de date care pot necesita spaţii de memorie diferite pentru rulări diferite ale aceluiaşi program). Pentru acest tip de programe se utilizează alocarea dinamică a memoriei funcţie de numărul şi dimensiunea variabilelor noi create pe parcursul execuţiei.

Observaţii:

- Zona de memorie folosită pentru alocarea dinamică este obţinută din *heap* (care este o zonă de memorie liberă aflată între zona de memorie permanentă a programului care se desfăşoară şi cea stivă).
- Deşi mărimea zonei heap este necunoscută, ea conţine, în general, o cantitate destul de mare de memorie liberă.
- Principalele funcţii pentru alocarea dinamică în C sunt *malloc* şi *free* aflate în fişierele header *stdlib.h* şi *alloc.h (sau malloc.h - depinde de compilator)*.

Funcţia malloc alocă o zonă din memoria rămasă liberă din heap

Prototipul funcţiei este:

*void *malloc(unsigned nr_de_octeti);*

- Funcţia returnează un pointer de tip void, care în fapt se poate atribui oricărui pointer.
- După o apelare *reuşită* se returnează un pointer spre primul octet al regiunii de memorie astfel alocată.
- Dacă memoria liberă din heap este insuficientă, apare o blocare de alocare şi se returnează valoarea NULL.

Exemple:

1) Pointerului către un şir de caractere:

*char *p;*

i se pot aloca 1000 de octeţi în memorie astfel:

- în C → *p=malloc(1000);*
- în C++ → *p=(char *)malloc(1000);*

Ca observaţie, în C++ trebuie folosit un modelator de tip (cast).

2) Se alocă spaţiu pentru 50 întregi.

```
int *p;
p=malloc(50*sizeof(int));
```

3) Un mod corect de utilizare a lui *malloc* necesită şi prevederea eventualei lipse de memorie (când *malloc* returnează null):

```
if(!(p=malloc(100)))
{printf("Depasire de memorie\n");
  exit(1);
}
```

Funcţia free returnează în sistem memoria alocată anterior (cu *malloc*).

Zona eliberată poate fi refolosită de o apelare ulterioară a lui malloc. Prototipul funcţiei este:

*void free(void *p);*

unde p este un pointer spre memoria alocată anterior.

Ca observaţie, nu trebuie apelată funcţia *free* cu un argument impropriu deoarece poate distruge lista de memorie liberă.

11. Utilizarea incorectă a pointerilor

- De multe ori erorile datorate pointerilor sunt foarte dificil de depistat.
- La folosirea greşită a unui pointer se citeşte sau se scrie într-o zonă necunoscută de memorie.
- Inconvenientul major este că efectele acestor greşeli sunt vizibile doar la execuţia programului, nu şi la compilare.
- Găsirea greşelilor este de multe ori o sarcină anevoioasă pentru programatori, mai ales că la unele rulări ale programelor ele nu sunt direct vizibile.

Exemple:

1) Utilizarea unei locaţii necunoscute de memorie.

```
int main()
{ int x, *p;
  x=10;
  *p=x;    /*se atribuie valoarea 10 unei locaţii
                    necunoscute*/
  ...
}
```

Această eroare poate să nu fie vizibilă pentru programe de mică dimensiune.

2) Neînţelegerea modului de folosire a unui pointer.

```
#include<stdio.h>
int main()
{       int x, *p;
        x=10;
        p=x; /*atribuire greşită (ar fi trebuit p=&x)*/
        printf("%d", *p);  /*Nu se va afişa
                                    valoarea 10*/
  …
}
```

3) Presupunerea incorectă asupra amplasării variabilelor în memorie.

```
…
char s[80], y[80];
char *p1, *p2;
p1=s;
p2=y;
if(p1 < p2) … /* operatie nerelevanta */
…
```

4) Presupunerea eronată că două matrice sunt alăturate şi pot fi indexate ca una singură prin simpla incrementare a unui pointer.

```
…
int prima[10], adoua[10];
int *p, t;
p=prima ;
for(t=0;t<20;t++)      *p++ = t;

…
```

5) Omiterea reiniţializării unui pointer care este lăsat să traverseze o zonă mare de memorie. Programul afişează valorile ASCII asociate caracterelor unui şir ***S***.

```
#include<stdio.h>
#include<string.h>
int main()
{       char *p;
        char s[80];
        p=s;
        do{     gets(s); /*citeşte şir*/
        /*apoi afişează echivalentul ASCII al fiecărui caracter*/
                while(*p)       printf(" %d ", *p++);
          } while(strcmp(s,"gata"));
 ...
}
```

- Problema este că pointerului ***p*** îi este atribuită adresa lui ***s*** o singură dată și prin incrementare, ajunge peste alte date.

Varianta corectă este:

```
#include<stdio.h>
#include<string.h>
void main(void)
{       char *p;
        char s[80];
        do{    p=s;   /*p este reiniţializat la fiecare intrare*/
               gets(s);
               while (*p)       printf(„%d ”, *p++);
           }while(strcmp(s,”gata”));
}
```

În concluzie, trebuie acordată atenţie codului programelor la folosirea pointerilor.
Este bine a se anticipa efectele utilizării acestora.

Cap. Structuri

1. Definirea conceptului de structură

- Limbajul de programare C poate prelucra atât date şi variabile singulare, cât şi variabile grupate, care permit o prelucrare globală.
- Un exemplu elocvent, din cea de-a doua categorie, îl reprezintă matricile care sunt mulţimi ordonate de date de acelaşi tip, relaţia de ordine dintre elemente fiind dată cu ajutorul indicilor. Numărul indicilor indică dimensiunea unei matrice (tablou). Tipul comun al elementelor tabloului este şi tipul tabloului.
- De multe ori însă, este util să grupăm datele în alt mod decât cel utilizat pentru matrici. Este cazul datelor care nu sunt neapărat de acelaşi tip şi care necesită o prelucrare globală. Această formă de grupare poartă denumirea de ***structură***.

Referirea la elementele grupei nu se face cu indici ci cu construcţii de felul numelui. Componentele unei grupe pot fi ele însele grupe. Mai mult, se poate defini o **ierarhie** a grupelor (datelor).

Astfel:

- grupa care nu este componentă a altei grupe este de *nivel cel mai înalt*;
- datele care nu mai sunt grupe de alte date sunt *date elementare*.

Ca o definiţie foarte generală, se poate spune că datele grupate conform unei ierarhii se numesc ***structuri***.

Observaţii:

- Datele elementare ale unei structuri pot fi izolate (singulare) sau tablouri;
- Fiecare structură reprezintă un tip nou de date, definit de utilizator.

2. Declaraţia de structură

Formatul general pentru declararea unei structuri este:

struct identificare
{
declaratii de variabile;
} identificare_1,identificare_2,...,identificare_n;

unde *identificare, identificare_1, identificare_2, ..., identificare_n* sunt nume care pot lipsi dar nu toate deodată.

Astfel:

- dacă *identificare_1, identificare_2, ... , identificare_n* sunt absenţi, atunci *identificare* trebuie să fie prezent.
- dacă *identificare* este absent, cel puţin *identificare_1* trebuie să fie prezent.

Este de precizat că:

- *identificare* defineşte un tip nou introdus prin declaraţia de structură dată.
- *identificare_1, identificare_2,... ,identificare_n* sunt structuri de tip *identificare.*

Observaţii:

- O structură de tip *identificare* poate fi declarată şi ulterior

  struct identificare numele_noii_structuri;

- O declaraţie oarecare de structură ***identificare_t*** (unde *t* = *1…n*) poate fi înlocuit de un tablou *k*-dimensional de elemente de tip *identificare*:

  identificare_t[lim1][lim2]...[limk]

Exemple:

1) Următoarele trei exemple de cod au acelaşi rezultat:

```
struct data_calendaristica
{int zi;
char luna[11];
int an;
} data_nasterii,data_angajarii;
```

sau

```
struct
{int zi;
char luna[11];
int an;
} data_nasterii,data_angajarii;
```

sau

```
struct data_calendaristica
{int zi;
char luna[11];
int an;
};
...
  struct data_calendaristica data_nasterii,data_angajarii;
```

2) Structură care conţine datele personale:

```
struct date_personale
  { char nume_prenume[100];
    char adresa[1000];
    struct data_calendaristica data_nasterii,data_angajarii;
    char sex;
  };
    ..............
 struct date_personale director, angajati[1000];
```

Variabila *director* este o structură de tip *date_personale* iar *angajati[1000]* este un tablou de structuri.

3) Definirea unor numere complexe A, B şi C.

```
struct COMPLEX
        {double real;
         double imag;
        }A, B, C;
```

4) Poziţia unui punct pe ecran este dată de două coordonate:

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

3. Accesul la elementele unei structuri

Accesul la elementele unei structuri se poate face sub una din cele două forme:

- *nume.nume_dată*
- *pointer -> nume_dată*

unde: *nume* este numele structurii,
nume_dată este numele componentei,
iar *pointer* este un pointer spre structură.

Exemple:

1)
```
struct data_calendaristica
        { int zi;
        char luna[10];
        int an;
        } dc,d[10];
    ...
 dc.zi=1;
dc.an=1995;
strcpy(dc.luna,”septembrie”);
...
d[3].zi=dc.zi;
d[3].an=dc.an;
strcpy(d[3].luna,dc.luna);
...
```

2) Funcţie ce calculează şi returnează modulul numărului complex z.

```
double modul(COMPLEX *z)
        {
        return sqrt(z->x * z->x + z->y * z->y);
        }
```

Trebuie precizat că structurile şi componentele acestora pot fi *iniţializate*.

4. Declaraţii de tip

Prin declararea unei structuri se introduce un tip nou de dată.

În general, se poate atribui nume unui tip, indiferent dacă el este un tip predefinit sau unul utilizator, utilizând o construcţie de forma:

typedef tip nume_tip;

unde

- *tip* este un tip predefinit sau un tip utilizator;
- *nume_tip* este numele care se atribuie tipului definit de *tip*.

Exemple:

1)Prin declaraţia

typedef double REAL;

datele

REAL x,y;

sunt de tip double.

2) Declararea tipului COMPLEX.

```
typedef struct
        { double real;
          double imag;
        } COMPLEX;
...
```

Putem declara numere complexe prin declaraţii de forma:

COMPLEX z,tz[10];

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<conio.h>
typedef struct {
                double x;
                double y;
               } COMPLEX;
void sum_c(COMPLEX *a, COMPLEX *b, COMPLEX *c);

int main()
{
 COMPLEX a,b,c;
 printf("\n\nIntroduceti partea reala si partea imaginara ");
 printf("\n ale numarului complex a :\n");
 if(scanf("%lf %lf",&a.x,&a.y)!=2)
        {
          printf("\nEroare");
          exit(1);
         }
 ….
```

```
…
printf("a = %g + i*(%g)\n",a.x,a.y);
 printf("\nIntroduceti partea reala si partea imaginara ");
 printf("\n ale numarului complex b :\n");
 if(scanf("%lf %lf",&b.x,&b.y)!=2)
        {
          printf("\nEroare");
          exit(1);
         }
 printf("b = %g + i*(%g)\n",b.x,b.y);
 sum_c(&a,&b,&c);
 printf("\na+b = %g + i*(%g)",c.x, c.y);

 getch();
}


void sum_c(COMPLEX *a, COMPLEX *b, COMPLEX *c)
{
  c->x = a->x + b->x;
  c->y = a->y + b->y;
}
```

5. Reuniuni

În C unei date i se alocă o zonă de memorie potrivit tipului datei respective şi în zona alocată acesteia se pot păstra numai date de tipul menționat.

Exemplu:

double x;

Pentru x se alocă 8 octeţi (64 de biţi) şi în zona respectivă se păstrează numere reale reprezentate prin complement faţă de 2 (a se vedea capitolul aferent sistemelor de operare).

Observaţie:

Reprezentarea cu semn (în binar) se face standard în trei moduri:

- *în mărime şi semn (MS)*;
- *în complement faţă de 1 (C1)*;
- *în complement faţă de 2 (C2)*.

Exemplu:

	MS	C1	C2
+5	0101	0101	0101
-5	1101	1010	1011

Se pot ivi situaţii când în aceeaşi zonă de memorie am dori să păstrăm date de tipuri diferite. De exemplu dacă după un anumit timp nu mai este nevoie de variabila ***x***, zona ar putea fi utilizată pentru a păstra date de alt tip (*char, int, float,* etc.). Aceste reutilizări ale zonelor de memorie duc automat la economisirea de memorie.

Pentru a realiza aceast deziderat se grupează împreună datele ce se doresc a fi alocate în aceeaşi zonă de memorie şi se foloseşte o construcţie similară ca cea de la structuri pe care o vom denumi ***reuniune*** (şi foloseşte cuvântul cheie ***union*** la declarare).

Exemple:

1)
```
union a
        {int x;         /* 2 octeţi pentru x */
        long y;         /* 4 octeţi pentru y */
        double r;       /*8 octeţi pentru r*/
        char c;         /* 1 octet pentru c */
        } var;
```

În declaraţia de mai sus ***var*** este o reuniune de tipul **a**. Accesarea variabilelor se poate face: *var.x;* sau *var.y;* sau *var.r;* sau *var.c*; dar în locaţii diferite ale programului

Pentru ***var*** se alocă zonă de memorie suficientă pentru a păstra numărul maxim de octeţi (8 octeţi în acest exemplu). Dacă s-ar fi înlocuit ***union*** cu ***struct*** ar fi fost necesari 15 octeţi (2+4+8+1=15).

2)

```
struct data
        { int timp;
          union { int i;
                  float f;
                  double d;
                } zc;
        } util;
```

Putem accesa:

```
util.zc.i=123;
```

Ca observaţie, spre deosebire de structuri, reuniunile nu pot fi iniţializate.

6. Câmpuri

- În limbajul C se pot defini şi prelucra date pe biţi. Această utilizare permite economisirea de memorie. De exemplu, presupunem că avem nevoie de o dată care ia numai valorile 0 sau 1. O astfel de dată s-ar putea păstra pe un singur bit, şi nu pe doi biţi cum ar fi o variabilă de tip întreg.

- **Prin definiţie, un şir de biţi formează un *câmp*.**

- Un câmp se păstrează într-un *cuvânt calculator* (care poate conţine mai multe câmpuri). Un cuvânt are 16 biţi (2 octeţi).

Practic, mai multe câmpuri se pot grupa formând o structură:

```
struct identificare
      { camp_1;
        camp_2;
       ...
        camp_n;
      } nume_1, nume_2, ..., nume_n;
```

Declaraţia de câmp este:

tip nume_camp: lungime_în_biţi;

sau:

tip: lungime_în_biţi;

unde ***tip*** poate fi: *unsigned*, *int*, sau *char*.

Observaţii:

- Dacă un câmp nu se poate aloca în cuvântul curent el se va aloca în cuvântul următor;
- Un câmp fără nume nu se poate referi. El defineşte o zonă neutilizată dintr-un cuvânt;
- Dacă lungimea în biţi a câmpului este 0, automat data următoare se alocă în cuvântul următor.