Laboratorul 11. TIPURI DE DATE DEFINITE DE UTILIZATOR

Introducere

Reprezentarea informația prelucrată de un program, reprezentată doar folosind tipurile predefinite şi cele derivate, poate fi destul de dificilă atunci când datele sunt complexe şi trebuie structurate după diferite criterii. Programatorul poate să-şi definească propriile tipuri de date care să răspundă cât mai eficient cerințelor aplicației.

Limbajele C/C++, oferă următoarele posibilități de definire de noi tipuri de date:

- **declarația typedef** – permite redenumirea tipurilor de date; prin aceasta creşte lizibilitatea programului;
- **enumerarea** – permite denumirea unui tip de date asociat cu tipul int simultan cu definirea unei liste de constante;
- **structura** – defineşte colecții de date de diferite tipuri referite printr-un singur nume;
- **câmpuri de biți** – reprezintă membri ai unei structuri care nu sunt alocați la nivel de octet, ci alocați la nivel de bit;
- **uniunea** – defineşte colecții de date de tipuri diferite care folosesc în comun o zonă de memorie.

Limbajul C++ oferă în plus față de C posibilitatea definirii de clase, acestea fiind tipuri de date care respectă principiile programării orientate pe obiecte.

11.1. Enumerarea

Tipul enumerare constă dintr-un ansamblu de constante întregi, fiecare asociată unui identificator. Sintaxa declarației este:

enum <id_tip_enum> {id_elem<=const>,…} <lista_id_var>;

unde: **- id_tip_enum** = nume tip enumerare ;

- id_elem = nume element;

- const = constantă de inițializare a elementului.

id_tip_enum reprezintă un nume de tip de date, putându-se folosi similar cu orice tip de date predefinit, putându-se declara variabile, pointeri, tablouri, etc. din acel tip.

Dacă declarația nu specifică constante pentru id_elem, valorile implice sunt 0 pentru primul element, iar pentru celelalte valoarea elementului precedent incrementat cu o unitate.

Identificatorii elementelor trebuie să fie unici în domeniul lor (diferiți de numele oricărei variabile, funcție sau tip declarat cu typedef).

```
enum boolean {false, true};          // valoarea identificatorului false este 0, iar a lui true este 1
```

sau

```
typedef  enum {false, true} boolean;          // declarația este echivalentă declarației  anterioare
```

De exemplu, se poate scrie secvența:

```
# include <stdio.h>

enum boolean {false, true};                    // valoarea identificatorului false este 0, iar a lui
                                               // true este 1
void main()
{
   boolean op1=false, op2;
   op2=true;
   printf("\n op1=%d\nop2=%d", op1, op2);
}
```

Tipul enumerare facilitează operarea cu variabile care pot lua un număr mic de valori întregi, asociindu-se nume sugestive pentru fiecare valoare. Programul devine mai clar şi mai uşor de urmărit.

11.2. Structuri de date

Structura este o colecție de date referite cu un nume comun. O declarație de structură precizează identificatorii şi tipurile elementelor componente şi constituie o definiție a unui nou tip de date.

Sintaxa declarației unui tip structură este:

```
struct id_tip_struct
{
        tip_elem1 id_elem1;
        tip_elem2 id_elem2;
        …
        tip_elemN id_elemN;
} lista_id_var_struct;
```

unde: - **struct** = cuvânt cheie pentru declararea tipurilor structură;

- **id_tip_struct** = numele tipului structură declarat;
- **tip_elemK** = tipul elementului K, unde K=1…N;
- **id_elemK** = numele elementului K;
- **lista_id_var_struct** = lista cu numele variabilelor de tipul declarat, id_ti_struct.

Pot să lipsească, fie numele structurii, fie lista variabilelor declarate, dar nu amândouă. De regulă se specifică numele tipului structură definit, aceasta permițând declarații ulterioare de obiecte din acest tip.

Elementele structurii se numesc generic membrii (câmpurile) structurii. Pentru fiecare membru se precizează tipul datei şi numele. Tipurile membrilor pot fi oarecare, mai puțin tipul structură care se defineşte, dar pot fi de tip pointer la structura respectivă.

De exemplu, se poate defini următorul tip de date:

```
struct ex_stru
        {       int m1;
                char m2;
                float m3;
        };
```

O variabilă de tip ex_stru se declară :

```
ex_stru var1;
```

Variabila var1 se alocă în memorie ca în fig.11.1.

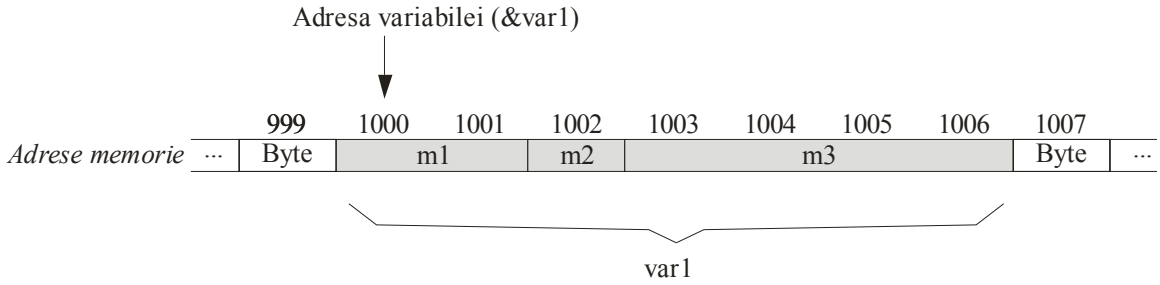

Fig.11.1. Alocarea în memorie unei variabile de tip ex_stru

Membrii unei structuri pot fi de orice tip, deci pot fi la rândul lor de tip structură, spre exemplu:

```
struct data              // se declară tipul de date data, ca structură cu 3 membri de tip unsigned int
       { unsigned int zi;
         unsigned int luna;
         unsigned int an;
       };

struct persoana                    // se declară tipul de date persoana ca structură cu 2
       { char nume[15];            // membri şir de caractere şi un membru de tip data
         char prenume[15];
         data data_n;
       }  pers1, pers2 ;                  // odată cu declarația tipului de date persoana, se
                                          // declară două  obiecte din acest tip de date
```

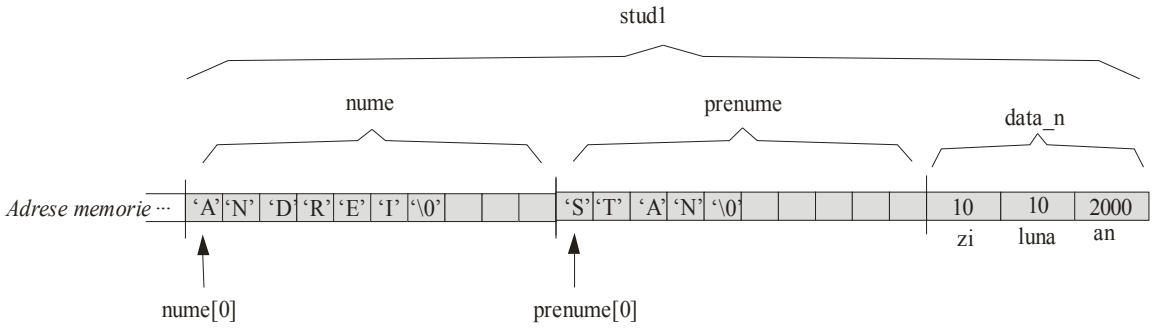

Fig.11.2. Alocarea în memorie a unei variabile de tip persoana

O variabilă de tip persoana se alocă în memorie ca în fig.11.2.

Se poate declara un tip structură folosind cuvântul cheie typedef, sintaxa fiind:

```
typedef struct {
                    tip_elem1 id_elem1;
                    tip_elem2 id_elem2;
                    …
                    tip_elemN id_elemN;
                } id_tip_struct;
```

unde semnificaţia denumirilor este identică cu cea desxcrisă anterior.

Se reia exemplul precedent utilizând typedef pentru declaraţia structurilor:

```
typedef struct
        {
          unsigned int zi;
          unsigned int luna;
          unsigned int an;
        } data;

typedef struct
        {
          char nume[15];
          char prenume[15];
          data data_n;
        } persoana;
```

În cazul folosirii declaraţiei cu typedef nu se pot face simultan şi declaraţii de variabile de tipul structurii. Acestea vor putea fi folosite ulterior.

Declaraţia unei variabile de tip structură se poate face cu iniţializare prin enumerarea valorilor membrilor în ordinea în care apar în declaraţie.

```
persoana pers= {“Ionescu”, ”Adrian”, 10, 10, 1975};        // declararea unui obiect persoana
                                                           // cu iniţializare
```

Dacă declaraţia nu se face cu iniţializare, membrii variabilelor iau valoare 0 dacă declaraţiile sunt globale sau statice, sau valori reziduale dacă declaraţiile sunt locale.

Referirea unui membru al unei variabile de tip structură se face folosind operatorul de selecţie (.) (punct), sub forma:

nume_variabilă . nume_câmp

De exemplu,

```
printf(”\nNumele: %s”, pers.nume);
printf(”\nPrenume: %s”, pers.prenume);
printf(”\nData nasterii: %d.%d.%d”, pers.data_n.zi, pers.data_n.luna, pers.data_n.an);
```

Se pot declara pointeri la structuri care pot prelua adresa oricărei variabile de tipul structurii respective.

```
persoana * p_pers;                  // se declară un obiect pointer la persoana
p_pers = &pers;                     // atribuire de valoare pointerului
```

Alocarea dinamică de memorie se poate face şi pentru obiecte de tip structură, folosind atât funcţiile specifice limbajului C, cât şi operatorii specifici din C++.

```
p_pers = (persoana*)malloc(sizeof(persoana));        // alocarea unui element de tip persoana
```

sau:

```
p_pers = (persoana*)malloc(dim*sizeof(persoana));    // alocarea unui tablou cu dim
                                                     // elemente de tip persoana
```

Eliberarea de memorie se face folosindu-se funcția free.

```
free(p_pers);
```

Alocarea de memorie folosind operatorii specifice se face astfel:

```
p_pers = new persoana[dim];          // alocarea unui tablou cu dim elemente de tip persoana
….
delete [ ] p_pers;                   // dealocarea memoriei
```

Referirea unui membru al unei structuri indicate de un pointer se face folosind sintaxa:

```
(* p_pers).nume;
```

sau folosind operatorul de selecție indirectă (->) (săgeată):

```
p_pers->nume;
```

Exemple de folosire a câmpurilor unei structuri:

```
puts("\nIntroduceti numele: ");
scanf("%s", &p_pers->nume);
puts("\nIntroduceti prenumele: ")
scanf("%s", &p_pers->prenume);
puts("\nIntroduceti data nasterii: ");
scanf("%d.%d.%d", &p_pers->data_n.zi, &p_pers->data_n.luna, &p_pers->data_n.an);
```

Operatorul de atribuire admite ca operanzi variabile structură de acelaşi tip, efectuând toate atribuirile membru cu membru.

```
persoana p1={"Popescu", "George", 5, 12, 1982}, p2;     // p1 se declară cu
                                                        // inițializare, p2 fără inițializare
p2=p1;                                                  // prin atribuire, p2 preia, membru
                                                        // cu membru datele din p1
printf("\nNumele:%s", p2.nume);
printf("\nPrenume:%s", p2.prenume);
printf("\nData nasterii: %d.%d.%d", p2.data_n.zi, p2.data_n.luna, p2.data_n.an);
```

Pot fi definite funcții care să realizeze transfer de informație prin variabile de tip structură. Transferul unui parametru de tip structură se poate face atât prin valoare, cât şi prin adresă sau variabile referință. Parametrii formali şi cei efectivi trebuie să fie de acelaşi tip.

Se consideră spre exemplu o structură definită pentru a reprezenta numere complexe prin valorile pentru partea reală şi cea imaginară:

```
struct complex { int re, im;} ;
```

Pentru acest tip de date se defini funcții care transferă obiecte în diferite moduri. Spre exemplu se poate defini o funcție care afişează informația conținută într-o astfel de dată:

```
void afisare(complex c)                 // funcție cu parametru de tip complex – se
{                                       // face transfer prin valoare
        printf("\n\t%d + i * %d\n", c.re, c.im);
}
```

Definirea unei funcții care citeşte de la tastatură valori pentru o variabilă complex se poate defini:

```
complex citire( )                    // funcție care returnează o dată de tip complex-
{       complex c;
        printf("\nre=");
        scanf("%d", &c.re);
        printf("\nim=");
        scanf("%d", &c.im);
        return c;
}
```

sau:

```
void citire(complex * c)             // funcție cu parametru pointer de complex-
{       printf("\nre=");             // transfer prin adresă
        scanf("%d", &c->re);
        printf("\nim=");
        scanf("%d", &c->im);
}
```

11.3. Câmpuri de biți

Câmpul de biți este un membru al unei structuri sau uniuni care are alocat un grup de biți în interiorul unui cuvânt de memorie.

Declarația unei structuri cu membrii câmpuri de biți se face sub forma:

```
struct id_tip_struct
        {
                tip_elem1 id_elem1 : lungime;
                tip_elem2 id_elem2 : lungime;
                …
                tip_elemN id_elemN : lungime;
        } lista_id_var_struct;
```

unde lungime reprezintă numărul de biți utilizați pentru reprezentarea în memorie a câmpului respectiv.

Restricțiile care se impun la declararea câmpurilor de biți sunt:

- tipul poate fi int signed sau unsigned;
- lungimea este o constantă întreagă cu valoarea în domeniul 0-15;
- nu se poate evalua adresa câmpului, deci nu se poate folosi operatorul & pentru acel câmp, deci nu se pot folosi pointeri pentru aceste câmpuri;
- nu se pot folosi tablouri de câmpuri de biți.

De exemplu, se consideră declarația:

```
struct ex_bit
{       unsigned m1 : 1;
        unsigned m2 : 4;
        unsigned m3 : 1;
        unsigned m4 : 10;
};
```

Dacă se declară o variabilă:

```
ex_bit variabila_mea;
```

ea, va fi alocată în memorie ca în fig. 11.3.

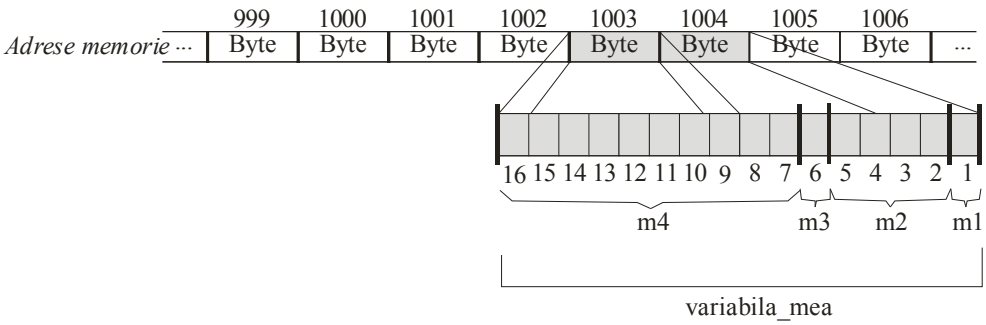

Fig.11.3. Alocarea în memorie a unei variabile de tip ex_bit

Aşa cum se ştie, unitatea de memorie cea mai mică direct adresabilă este octetul. Folosirea câmpurilor de biţi permite accesul la nivel de bit, ceea ce reprezintă o facilitate oferită de C/C++.

Un alt avantaj oferit de folosirea câmpurilor de biţi este economia de memorie care se poate face. Dacă unele date iau valori în domenii de valori restrânse, reprezentarea lor se poate face pe un număr de biţi care permite reprezentarea acelor valori.

Dacă se folosesc n biţi, se pot reprezenta 2^n valori.

De exemplu, dacă se doreşte folosirea datelor calendaristice, valoarea zilei este cuprinsă în intervalul 1...31, deci sunt necesari 5 biţi ($2^5 = 32$)., valoarea lunii se poate reprezenta pe 4 biţi, iar pentru an am putea folosi 12 biţi. O astfel de reprezentare va folosi 3 sau 4 octeţi, depinde de compilatorul folosit, deci mai puţin decât dacă declaraţiile ar fi făcute fără folosirea câmpurilor de biţi. Este posibil ca o parte din biţi să nu fie folosiţi. Acest lucru nu generează erori.

Referirea membrilor se face ca pentru orice structură, folosind operatorii de selecţie directă sau indirectă (punct sau săgeată).

O problemă apare, de exemplu, la citirea de valori pentru astfel de date, pentru că, pentru aceşti membri nu se poate folosi operatorul adresă (&).

```
#include <stdio.h>

struct DATA
{    unsigned int zi    :5;      // se declară membrul zi cu reprezentare pe 5 biţi
   unsigned int luna   :4;      // se declară membrul luna cu reprezentare pe 4 biţi
   unsigned int an     :12;     // se declară membrul an cu reprezentare pe 12 biţi
};

void main()
{   DATA data_n;                                    // se declară un obiect de tip DATE
  unsigned int aux1, aux2, aux3;

  puts(”Introduceti data :”)
  scanf(”%2d.%2d.%4d”, &aux1, &aux2, &aux3); // este necesară utilizarea unor
                                    // variabile auxiliare pentru citirea valorilor, deoarece
                                    // pentru câmpurile de biţi nu se poate face referire la adresă
   data_n.zi=aux1;
  data_n.luna=aux2;
  data_n.an=aux3;
  .... }
```

Dacă se doreşte determinarea dimensiunii în octeți a memoriei folosită pentru reprezentarea acestor date, se poate folosi operatorul sizeof().

```
printf("data_n ocupa %d octeti", sizeof(data_n));
```

11.4. Uniuni

Uniunea permite utilizarea în comun a unei zone de memorie de către mai multe obiecte de tipuri diferite. Sintaxa de declarare a unei uniuni este similară declarației unei structuri:

```
union id_tip_uniune
{       tip_elem1 id_elem1;
        tip_elem2 id_elem2;
        …
        tip_elemN id_elemN;
} lista_id_var_uniune;
```

sau

```
typedef union
{       tip_elem1 id_elem1;
        tip_elem2 id_elem2;
        …
        tip_elemN id_elemN;
} id_tip_uniune;
```

Spațiul alocat în memorie corespunde tipului de dimensiune maximă, membrii uniunii utilizând în comun zona de memorie.

Se consideră declarația:

```
union exemplu
{
        char c;
        int i;
        float f;
};
```

O variabilă de tipul exemplu declarată, var1, se reprezintă în memorie ca în fig.11.4. Reprezentarea se face pe 4 octeți, adică numărul de octeți necesar câmpului cu reprezentare în domeniul cel mai larg. Câmpul cel mai mare este câmpul f.

```
exemplu var1;
```

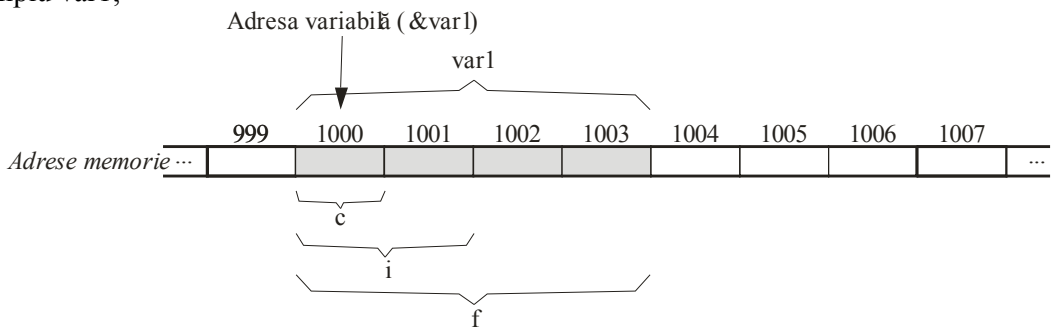

Fig.11.4. Reprezentarea în memorie a unei uniuni

La un moment se foloseşte un membru al uniunii. Modificarea unui membru va produce modificări şi asupra celorlalţi.

Se consideră următoarea aplicație:

```
#include <stdio.h>

struct octet
{       unsigned int b0:1;      // se declară o structură care ocupă un octet de memorie,
        unsigned int b1:1;      // cu acces separat, prin membrii săi, la fiecare bit în parte
        unsigned int b2:1;
        unsigned int b3:1;
        unsigned int b4:1;
        unsigned int b5:1;
        unsigned int b6:1;
        unsigned int b7:1;
  };

union intreg
{       char val;               // se declară o uniune ce ocupă un octet de memorie care poate
        octet bit;              // fi accesat ca şi char prin membrul val, sau ca octet prin
};                              // membrul bit

void main ()
{   intreg i;
    i.val=22;
// se afişează fiecare bit al uniunii separat folosindu-se câmpul i.bit:
    printf("\n0x%x se reprezinta in binar: %d%d%d%d%d%d%d%d", i.val, i.bit.b7,
            i.bit.b6, i.bit.b5, i.bit.b4, i.bit.b3, i.bit.b2, i.bit.b1, i.bit.b0);
}
```

Programul realizează afişarea în format binar a unei valori reprezentată pe un octet, de tip char.

Exemple

/**

Exemplu1 11.1. Se scrie un program în care se declară o structură pentru reprezentarea punctelor în plan prin coordonatele sale. Se exemplifică folosirea variabilelor de tip structura, tablouri si a pointerilor de structuri.

**/

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#define N 4

struct punct { int x, y;};    // structură pentru reprezentarea punctelor în plan prin coordonatele sale

void main()
{  punct p1, p2;              // se declară variabile de tip punct
```

```
  punct pct[N];          // se declară un tablou cu elemente de tip punct
  punct *pp;             // se declară un pointer la tipul punct
   int i;
  double dist;

  printf("Introduceti coordonate pentru p1:");
  printf("\np1.x=");
  scanf("%d", &p1.x);
  printf("p1.y=");
  scanf("%d", &p1.y);
  printf("Introduceti coordonate pentru p2:");
  printf("\np2.x=");
  scanf("%d", &p2.x);
  printf("p2.y=");
  scanf("%d", &p2.y);
// se calculeaza distanța dintre două puncte
  dist=sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
  printf("\nDistanta dintre p1 si p2 este: %.2f\n", dist);
  getch();

  printf("Introduceti coordonate pentru cele 10 puncte:\n");
  for(i=0 ; i<N ; i++)
  {printf("\npct[%d]:", i);
    printf("\nx=");
    scanf("%d", &pct[i].x);
    printf("y=");
    scanf("%d", &pct[i].y);
  }
  printf("Coordonatele sunt:\n");
  for(i=0 ; i<N ; i++)
      printf("punctul %d - <%d , %d> \n", i, pct[i].x, pct[i].y);
// se face referire la elementele tabloului cu elemente de tip punct in mod indirect, prin pointerul pp
  pp=pct;
  printf("Coordonatele sunt:\n");
  for(i=0;i<N;i++)
      printf("punctul %d - <%d , %d>\n", i, (pp+i)->x, (pp+i)->y);
  getch();
}
```

```
/*************************************************************************
```

Exemplu1 11.2. Se scrie un program în care se declară o structură pentru reprezentarea numerelor complexe. Se definesc următoarele funcții: o funcție care citeşte de la tastatură valori pentru o dată de tip complex, o funcție care afişează o dată de tip complex, o funcție care calculează şi returnează suma a două numere complexe. Valoarea returnată de funcție este tot de tip complex.

```
*************************************************************************/
#include <stdio.h>
struct complex { int re, im;} ;                    // declarația tipului de dată complex

void afisare(complex c)          // funcție cu parametru de tip complex – se face transfer prin valoare
{
          printf("\n\t%d + i * %d\n", c.re, c.im);
}
```

```
void citire(complex * c)        // funcție cu parametru pointer de complex- transfer prin adresă
{       printf("\nre=");
        scanf("%d", &c->re);
        printf("\nim=");
        scanf("%d", &c->im);
}

complex suma(complex & a, complex & b)  // funcție care preia prin referință date de tip complex
{       complex c;                      // şi returnează o dată de tip complex
        c.re=a.re+b.re;
        c.im=a.im+b.im;
        return c;
}

void main()
{  complex c1, c2, c3;                          // declarație de variabile complex

   printf("\nSe citesc valori pentru c1:");
   citire(&c1);                                 // funcția preia adresa unei variabile de tip complex
   printf("\nSe citesc valori pentru c2:");
   citire(&c2);
   printf("\nNumerele complexe sunt:");        // funcția de afişare preia ca parametri
   afisare(c1);                                 // variabile de tip complex, transferul se
   afisare(c2);                                 // face prin valoare
   c3=suma(c1, c2);             // funcția preia parametrii prin referință şi întoarce o valoare
                                // care e atribuită unei variabile
   printf("\nSuma numerelor complexe este:");
   afisare(c3);
}
```

/**

Exemplu1 11.3. Se scrie un program prin care se citeşte o valoare întreagă de la tastatură. Se afişează în binar valoarea citită. Se vor folosi câmpuri de biți şi uniuni. (Se consideră tipul int cu reprezentare pe 4 octeți).

**/

```
#include <stdio.h>

struct dword    {       unsigned b0:1;          // structura permite accesul la fiecare bit
                        unsigned b1:1;          // care intră în alcătuirea unei reprezentări
                        unsigned b2:1;          // pe 4 octeți (32 biți).
                        unsigned b3:1;
                        unsigned b4:1;
                        unsigned b5:1;
                        unsigned b6:1;
                        unsigned b7:1;
                        unsigned b8:1;
                        unsigned b9:1;
                        unsigned b10:1;
                        unsigned b11:1;
                        unsigned b12:1;
                        unsigned b13:1;
                        unsigned b14:1;
```

```
                    unsigned b15:1;
                    unsigned b16:1;
                    unsigned b17:1;
                    unsigned b18:1;
                    unsigned b19:1;
                    unsigned b20:1;
                    unsigned b21:1;
                    unsigned b22:1;
                    unsigned b23:1;
                    unsigned b24:1;
                    unsigned b25:1;
                    unsigned b26:1;
                    unsigned b27:1;
                    unsigned b28:1;
                    unsigned b29:1;
                    unsigned b30:1;
                    unsigned b31:1;
                };

union intreg    {       int a;          // prin cei doi membrii ai uniunii, aceeaşi zonă de
                        dword d;        // memorie poate fi interpretată fie ca int, fie la
                                        // nivel de biţi prin membrul dword
                };

void print_dword( dword b)   // afişarea binară a obiectului
{       printf("%d%d%d%d%d%d%d%d %d%d%d%d%d%d%d%d
                 %d%d%d%d%d%d%d%d %d%d%d%d%d%d%d%d",
                b.b31, b.b30, b.b29, b.b28, b.b27, b.b26, b.b25, b.b24,
                b.b23, b.b22, b.b21, b.b20, b.b19, b.b18, b.b17, b.b16,
                b.b15, b.b14, b.b13, b.b12, b.b11, b.b10, b.b9,  b.b8,
                b.b7,  b.b6,  b.b5,  b.b4,  b.b3,  b.b2,  b.b1,  b.b0);
}

void main()
{       intreg var1;
        printf("Introdu un intreg = ");
        scanf("%d", &var1.a);
        printf("\n %d se reprezinta in binar:", var1.d);
        print_dword(var1.d);
}
```

Întrebări. Exerciţii. Probleme.

1. Să se rescrie exemplul nr. 11.1 în care se defineşte o funcţie pentru citirea de la tastatură a unei date de tip punct, una pentru afişare şi o a treia care calculează distanţa dintre două puncte preluate ca parametri. Apelurile lor se vor folosi în funcţia main.

2. Să se scrie un program în care se defineşte o structură pentru reprezentarea unui triunghi prin lungimile laturilor sale. Se defineşte o funcţie pentru citirea de la tastatură a unei date de tip triunghi, una pentru afişare şi o a treia care calculează aria triunghiului. Apelurile lor se vor folosi în funcţia main.