CAPITOLUL 9

9. Tipuri de date definite de utilizator

S-a discutat până acum, despre tablouri de date mono sau multidimensionale care însă conțineau numai date de acelaşi tip. Sunt multe cazuri în care este nevoie ca date de tipuri diferite să fie grupate sub un nume comun.

Această problemă s-a rezolvat prin introducerea noțiunii de structură de date, care permite utilizatorului să-şi creeze noi tipuri de date cât mai apropiate de obiectele din lumea reală. Acest lucru se mai numeşte abstractizarea datelor.

La fel cum funcțiile permit descompunerea complexității unui program în termeni de operațiuni pe date, structurile sunt folosite pentru abstracțiuni.

9.1. Declararea unei structuri

O structură este un tip de dată generalizat definit de utilizator, care poate fi folosit pentru gruparea diferitelor tipuri de date într-un nou concept. De exemplu un tip de date ce se referă la unele din caracteristicile unui student este prezentat mai jos.

```
struct Student
    {
    char name[20]; // numele studentului
    unsigned short int age; // virsta
    unsigned char n_lab; // nota la laborator
    unsigned char n_ex; // nota la examen
    }
```

Există o convenție comună legată de numele unei structuri şi anume acesta să înceapă cu literă mare. Un exemplu de declarații cu acest tip de date este dat mai jos.

```
    struct Student tmp, an99[20];
```

9.2. Operatorul typedef

Comanda typedef care este folosită la redenumirea unor variabile
enumerate, poate fi folosită şi împreună cu cuvântul cheie struct. Pentru o structură trebuie să apară, după ce este definită, şi înainte de orice declarație de variabile. Uzual noul tip de date se redenumeşte (i se ataşează o etichetă echivalentă) ca mai jos:

```
typedef struct Student
{
char name[20]; // numele studentului
unsigned short int age; // virsta
unsigned char n_lab; // nota la laborator
unsigned char n_ex; // nota la examen
} stud;
```

Deci noul tip de date a fost redenumit ca stud, în aceste condiții declarații relative la el devin:
Un alt exemplu de definire Ş
Stud tmp,an99[20];

- Definesc noul tip de date

```
struct my_data
{
char name[19];
secol[11];
short month,day,year;
}
```

- Folosesc noua definiție

```
struct my_data test,tab[100],*ptab;
```

Deoarece my_data nu este un tip predefinit de date ,ci este un conglomerat definit de programator trebuie specificat şi cuvântul cheie struct.

Creearea unui nou tip de date se face astfel:

```
typedef  struct
        {
        char name[19];
        secol[11];
        short month,day,year;
        }pers;
```

Începând de acum pers se va folosi direct fără alte specificații în definirea unor elemente de acest tip.

```
pers test,tab[100],*ptab;
```

Inițializarea unei structuri
Ca şi în cazul tablourilor sau altor tipuri de date, poate fi realizată astfel:
→ La declararea unui element
Ex: pers eu={“MIKE”, “354263”,3,5,1978};
→ Oriunde in cadrul programului
Obs: Inițializarea unei structuri nu poate fi realizată la definirea noului tip de dată (cu typedef).Evident, există o diferență clară intre definirea unui nou tip de dată şi definirea unei variabile de un anumit tip.

9.3. Accesarea membrilor unei structuri

Deoarece acest tip de date este creat de programator compilatorul sau funcțiile existente nu ştiu să lucreze decât cu tipurile predefinite de date. De aceea este necesar ca

prelucrarea să se facă tot la nivel de membri (câmpuri). Pentru aceasta trebuiesc construite funcții specializate în operațiunile cu noul tip de date.

În cazul declarației unui element de tipul respectiv operatorul de acces la oricare din membri este '.'. Accesul se face prin referire la numele global urmat de numele membrului ca în exemplul de mai jos unde este realizată citirea de la tastatură a unui tablou de elemente de tip student.

```
…
for(i=0;i<10;i++)
    {
    clrscr();
    printf("Nume");
    scanf("%s", an99[i].nume);
    printf("\n Nota la laborator:");
    scanf("%d",an99[i].n_lab);
    printf("\n Nota la examen:");
    scanf("%d",an99[i].n_ex);     }
```

9.4. O structură în interiorul altei structuri (nested structures)

Există cazuri când este necesar ca informația să fie cât mai ierarhizată obținându-se astfel un cod cât mai clar. În cazul din exemplului anterior, dacă doresc să realizez o aplicație pentru un secretariat, am nevoie şi de numărul grupei. Există două metode de a rezolva problema. Una ar fi introducerea a încă unui câmp care să rețină numărul grupei, dar mod de rezolvarea poate duce pe măsură ce tot mai adaug caracteristici noi în structură la un cod greu de urmărit. Altă cale ar fi crearea unei noi structuri după cum este prezentat mai jos.

```
typedef struct Grupa
        {
        int nume; // numarul grupei
        int an; // an de studii
        stud t[35]; grupa contine maximum 35 de studenti
        }gr;
gr temp;
```

Accesul la membri în acest caz se face prin compunere ca mai jos

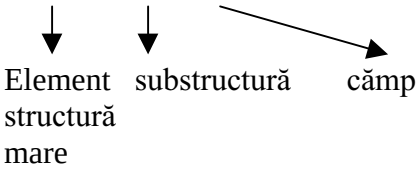
```
temp.nume //umele grupei
temp.t[3].name      // numele celui de-al treilea student din grupă.
```

Mai jos prezentăm încă un exemplu

```
typedef  struct
        {char name[19],snum[11];
        struct
                {short day;
                short month;
```

```
        short year;
        }birth_date
}pers;
```

O referire la day se va face pentru persoana eu: eu.birth_date.day



Obs: numele cămpurilor unei structuri nu sunt văzute direct în afară şi deci o referire:
struct x {int x;) x; este legală la accesare x.x=5.
Atenție ! Începătorii să evite acest lucru deoarece crează o ambiguitate nerecomandată în cod.

9.5 Structuri autoreferite

O structură NU poate conține instanțieri ale ale ei, dar poate conține pointeri către instanțieri ale ei.

```
struct p
    {int a,b;
        float c;
        struct p next;
    }
```

la analiza lexicală se intră în ciclu infinit

```
struct p
    {int a,b;
        float c;
        struct p* next;
    }
```

OK!!Acest lucru este posibil deoarece pointerul este o variabilă care reține adresa altei variabile

9.6 Transferul structurilor ca argumente de funții

→ transfer chiar structura (prin valoare)
→ transfer un pointer către ea (prin referință)

Ex: pers om;
func1 (om)
func2 (&om)

Transferul prin valoare se recomandă in următoarele cazuri:

- structura este mică
- garantarea faptului că funția apelantă nu modifică structura apelată

La apelul prin valoare compilatorul generează o copie a respectivei structuri (funcția apelantă poate deci modifica doar această copie).
Dacă am dori să specificăm transferul prin referință in clar (vezi func2(&om)) este suficient ca să declarăm pers* om şi atunci func2(om) este corect.
Atunci în acest caz func1(om) nu ami este corect pentru că om este un pointer şi nu o valoare.

9.7 Returnarea unei structuri

K&R nu permite returnarea unei structuri dar ANSI permite acest lucru.

Deci:

pos by value

```
pers f(….)
{
        pers x;
        …
        return x;
}
```

pos by reference

```
pers* f1(…..)
{
        static pers* p;
        …
        return &p;
}
```

9.8. Câmpuri de biți

Membrii unei structuri de tip int sau unsigned pot fi alocați în câmpuri de 1 până la 16 biți. Biții sunt alocați începând cu cel mai puțin semnificativ bit al fiecărui cuvânt de 16 biți. Sintaxa este:

tip [identificator] ":" lărgime ";"

Lărgimea câmpului trebuie să fie o expresie întreagă constantă cuprinsă intre 0 si 16 (0 înseamnă aliniere la următoarea limită de cuvânt). Dacă nu se specifică un nume pentru câmpul de biți, câmpul este alocat dar nu este accesibil (se foloseşte pentru alinieri).

```
struct special_word
    {
    int      i :2 ; // va fi reprezentat pe biții b0,b1
    unsigned  j :5 ; // va fi reprezentat pe biții b2 până la b6
    int        :4 ; // va fi reprezentat pe biții b7 până la b10
    int      k :1 ; // va fi reprezentat pe bitul b11
    unsigned  m :4 ; // va fi reprezentat pe biții b12 până la b15
    } ;
```

O restricție importantă asupra operațiilor care se pot face cu câmpurile de biți este aceea că nu li se poate afla adresa cu ajutorul operatorului & (adresa unui bit nu are sens).

9.9. Uniuni

Uniunile corespund cu înregistrările cu variante din alte limbaje de programare (Pascal, Modula-2, etc.); o uniune este un tip “hibrid”, valoarea variabilei uniune fiind valoarea membrului activ al uniunii, ceilalți membri fiind nedefiniți. Din punct de vedere sintactic definiția unui tip uniune seamănă cu definiția unui tip structură cu diferența că indicatorul struct se înlocuieşte cu union (sunt permise şi câmpuri de biți).
Ca implementare sunt similare cu structurile, cu diferența că membrii sunt suprapuşi unul după altul în memorie. Spațiul de memorie este folosit la maximum în raport cu

structurile. De fapt, permit ca aceeaşi zonă de memorie să fie interpretată diferit fără a folosi **cast**.

Dimensiunea spațiului de memorie alocat uniunii este egală cu dimensiunea spațiului alocat celui mai mare membru. Accesul la membrii uniunii se face sub forma unor componente selectate, folosind aceiaşi operatori şi aceleaşi reguli ca în cazul structurilor. Numele de uniuni respectă aceleaşi reguli ca şi numele de uniuni (spațiu de nume, utilizare etc.).

Înregistrări variabile

În cazurile în care se poate ca într-o bază de date, două sau mai multe cămpuri să se autoelimine reciproc (funcție de datele introduse în ea) se poate obține optimizarea spațiului grupat în memorie de un element al structurii prin gruparea acelor cămpuri într-o uniune ce aparține structurii.

Ex.:

```
typedef union
{
        struct
        {
        char c1,c2;
        }s
long j;
float x;
} u
```

1000	1001	1002	1003
C1	C2		
J			
X			

Compilatorul alocă suficientă memorie pentru a putea reține cel mai mare membru şi toți membrii încep la aceeaşi adresă. Datele stocate într-o uniune depind de care membru al uniunii se va utiliza.

pas.s.c1='a'

pas.s.c2='b'

1000	1001	1002	1003
a	b	...	Unused

pas.j=5 suprascrie cele 2 caractere şi depune 5 pe toți cei 4 octeți (tip long int)

1000	1001	1002	1003
0	0	0	5

Suportă aceleaşi reguli sintactice ca structurile.

Există 2 aplicații de bază pentru uniuni:

1. creează structuri flexibile (varianta records în care pot reține diferite tipuri de date)
2. interpretează aceeaşi memorie în moduri diferite

Exemplu pentru cazul 1:

```
Union myunion
   {
   int i ;
   double d ;
   char c ;
   } mu ;

sizeof ( myunion ) --> 8
sizeof ( mu ) --> 8
sizeof ( mu .i ) --> 2
sizeof ( mu .c ) --> 1
sizeof ( mu .d ) --> 8
```

Respectând conversiile necesare, putem spune că un pointer către o uniune este un pointer către oricare dintre membrii ei:

*(double *) & mu == & mu .d*

O uniune nu poate fi inițializată decât prin intermediul primului său membru declarat. Vom prezenta mai jos un program care citeşte cuvintul special care reține componentele hardware de bază instalate într-un sistem de calcul. Acesta este inițializat în urma autodetecției făcute în secvența de pornire ce are loc la punerea sub tensiune a sistemului.

```
#include<stdio.h>
#include<conio.h>
#include<dos.h>

union
    {
    unsigned w;
    struct bit_field {
     unsigned fd :1;
     unsigned dimR :2;
     unsigned modV :2;
     unsigned nrF :2;
     unsigned DMA :1;
     unsigned nrRS :3;
     unsigned game :1;
     unsigned serP :1;
     unsigned nrLpt :2;
```

```
            } b;
    } eqp;

void main(void)
    {
    unsigned far *ptr;

     clrscr();
     ptr=MK_FP(0,0410);
     eqp.w=*ptr;
    // urmează afişarea fiecărui element al uniunii conform
    // cu înțelesul dat
    …
    getch();
    }
```

Cazul 2 Interpretarea diferită a datelor

Ca un exemplu să considerăm problema comunicației seriale, unde datele vin octet cu octet.

Uniunile permit o modalitate de grupare, a octeților, împreună astfel încăt să poată fi reconstruite în forma lor originală. De exemplu, să presupunem că get_byte( ) este o funcție care întoarce un octet citit dintr-un device de comunicație.

O valoare double (pe 8 octeți) poate fi extrasă din device prin 8 apeluri recursive ale funcției după cum se vede:

```
union double
{
        char c[];
        double val;
}

double get_double( )
{
        extern char get_byte( );  //optimal – depinde de modul de structurare a
                                  //fişierului_____
        int j;
        union double d;
        for (j=0; j<8 ; j++)
                d.c[j]=get_byte( );
        return d.val;
}
```

Atenție!!! Este o diferență majoră între acest mod de interpretare şi conversiile prin **cast**.

Inițializarea uniunii (specifică ANSI):

```
union ex
        {int I;
        float f;
```

```
        };
    union ex test={1};
```
sau în cazul uniuni ce reține şi o structură
```
    union n
        {
        struct {int i; float f;}s;
        char ch[6];
        }
    union n test1 = {1, 1.10}
```

9.10. Enumerări

Tipurile de date enumerate sunt folosite în principal pentru a da nume mnemonice unor constante întregi. În exemplul de mai jos

enum zile { luni,marti,miercuri,joi,vineri,simbata,duminica } oricezi ;

se defineşte un nou tip întreg, numit enum zile, un număr de constante de acest tip (luni, marți, ...) cu valori de acest tip, şi se declară o variabilă de tip enum zile, numită oricezi.

În prezenta opțiunii -b (sau a comenzii corespunzătoare din mediul integrat) compilatorul are libertatea sa aloce 1 sau 2 octeți pentru o variabila de tip enumerare (in funcție de valorile constantelor definite); valorile constantelor definite sunt de tip unsigned char sau int. În C, unei variabile de tip enumerare i se poate atribui orice valoare întreagă;

```
enum days this_day ;        // legal si in C si in C++
days another_day ;          // legal numai in C++
```

Constantele de enumerare au implicit valori începând cu 0 care cresc din 1 in 1 (luni = 0, marți = 1, ..., duminica = 6). La definiția tipului, orice constantă de enumerare poate fi eventual urmată de un inițializator, sub forma unei expresii întregi constante (pozitive sau negative), care stabileşte valoarea constantei. Constantele de enumerare următoare care nu au o valoare explicită cresc din 1 in 1. Sunt permise valori duplicate. Numele constantelor de enumerare se află în acelaşi spațiu de nume ca şi numele de variabile, funcții şi tipuri. Valori de tip enumerare pot apare oriunde sunt permise valori de tip întreg.

```
enum coins
    {
    penny = 1,              // 1
    tuppence,               // 2
    nickel = penny + 4,     // 5
    dime = 10,              // 10
    quarter = nickel * nickel    // 25
    } ;
```

Tipuri enumerative

Se folosesc atunci cand se doreste creerea unui set mic de valori care pot fi asociate cu o variabila .Compilatorul va genera o eroare daca se incearca asigurarea unei valori care nu face parte din multimea valorilor specificate in declaratii.

De ex.:

enum { red,blue,green,yellow } my_colors
enum { bright,medium,dark } my_intensity

color = yellow *OK*
color = bright *ERROR*

Aceste valori numerice pot fi si direct specificat ,caz in care compilatorul aplica sumarea cu 1 poziție numai la cele nespecificate.
De ex.:

enum { apple,arong=10,lemon,grope=-5,melon }

va avea ca rezultat urmatoare atribuire de catre compilator:

apple=0;
orange=10;
lemon=11;
grape=-5;
melon=-4;