LUCRAREA 9

Scopul lucrării îl constituie prezentarea tipurilor de date neomogene *(structurile), utilizarea operatorului* `typedef` *în contextul structurilor, pointerii la structuri şi construcția unui tip abstract de dată, lista.*

I. OBSERVAȚII TEORETICE

1.1. Structuri

1.1.1. Utilitate

Dacă vectorii au utilitate esențială în cazul şirurilor de caractere (v. **`Laborator 8`**), unde mulțimea omogenă este esențială, structurile sunt foarte potrivite în situațiile în care trebuie memorate sub acelaşi nume caracteristici diferite ale unui anumit obiect. De exemplu `Numele`, `Prenumele`, `Adresa` şi `DataNaştere` sunt patru caracteristici de tipuri diferite de date ale aceluiaşi obiect, `Persoana`.

O înlănțuire de astfel de obiecte formează o clasă de obiecte, iar modelarea sa se poate face în C cu ajutorul structurilor de date dinamice de tip **`listă`**. Acest subiect este tratat în detaliu în semestrul 2 al anului întâi.

Fiecare element al unei structuri poartă denumirea de **`câmp`**.

Declarația unei structuri se poate vedea din următorul exemplu.

Exemplu:

```
...
struct Persoana
{
   char Nume[24];
   char Prenume[32];
   char Adresa[64];
   int ZiNastere;
   char LunaNastere[12];
   int AnNastere;
};
```

În acest prim exemplu, folosind cuvântul-cheie `struct`, compilatorul va recunoaşte corect acest tip de dată. După cum vedeți, într-un acelaşi spațiu de memorie, identificat cu un nume (aici `Persoana`) sunt stocate (salvate) şase câmpuri, de *tipuri diferite.*

Pentru a declara variabile de tip structură, sintaxa este următoarea:

```
struct Persoana pers1, pers2;
```

Aici au fost declarate două variabile, `pers1` şi `pers2`, de tipul structurii `Persoana`.

Observaţie:

Dacă se omite identificatorul structurii (denumit *tag* în termenii limbajului C), aici `Persoana`, atunci declaraţia anterioară nu are sens, şi este marcată cu eroare din partea compilatorului.

1.1.2. Spaţiul rezervat

Pentru a putea face alocări dinamice în care sunt implicate structurile, trebuie să cunoaştem spaţiul pe care compilatorul îl rezervă acestora.

Ca `regulă generală`, trebuie folosit operatorul **`sizeof()`** asupra tipului structură dorit. După cum vă reamintiţi, `sizeof` întoarce numărul de octeţi (`bytes`) ai tipului de dată asupra căruia se aplică.

Atenţie!

Este incorect să sumăm octeţii fiecărui câmp în parte şi astfel să stabilim numărul total de octeţi. Aceasta pentru că nu se garantează că aceste câmpuri se află în memorie succesiv. Astfel, pot exista octeţi liberi între diferitele câmpuri, şi aceasta din cauza eventualelor "`alinieri în memorie`", sau a optimizărilor venite din partea compilatorului (rezultate în urma unor set-ări specifice de configuraţie, pe care le putem face după instalarea mediului de lucru).

1.2. Structurile şi definiţiile de tipuri

1.2.1. Tipul structură

Dacă dorim o utilizare facilă a acestui tip special de dată, este de preferat varianta în care definim, cu ajutorul `typedef`, **`tipul de dată`** structură.

Definiţia se face respectând **`regula typedef`**: înaintea noului nume de tip pe care dorim să-l introducem, se specifică tipul de dată căruia i se va asocia acea nouă denumire (cunoscută şi drept *alias*).

Exemplu:

```
typedef struct Persoana PERSOANA;
```

Noul tip de dată este `PERSOANA`. `PERSOANA` este un *alias*. El este un echivalent perfect al notaţiei:

```
struct Persoana
```

pentru orice context sintactic corect, în care această combinație de cuvinte-cheie ar putea apărea. Un scurt exemplu în care putem folosi noul tip de dată construit este următorul:

```
PERSOANA p1;
strcpy(p1.LunaNastere, "Decembrie");
```

1.2.2. Tipul pointer la structură

Pe lângă tipul de dată structură, în operațiile cu structuri de date abstracte este folosit şi tipul '*pointer la structură*'.

În situația în care structura este **referită indirect**, prin intermediul unui pointer, operatorul specific de lucru cu câmpurile structurii este operatorul săgeată, compus din semnele `minus` şi '`mai mare`': **->**

Exemplu:

```
struct Persoana
   {
   char Nume[24];
   char Prenume[32];
   char Adresa[64];
   int ZiNastere;
   char LunaNastere[12];
   int AnNastere;
   };
struct Persoana p1;
struct Persoana *pPersoana;  // pointerul la structură

pPersoana = &p1;
pPersoana->ZiNastere = 25;
pPersoana->AnNastere = 1950;
```

După cum vedeți, s-a declarat o variabilă de tipul acestei structuri, anume `p1`, şi un pointer la aceasta, `pPersoana`. După inițializarea obligatorie a pointerului, aici cu adresa variabilei *p1*, stabilirea valorilor câmpurilor prin intermediul pointerului (deci *indirect*) cere utilizarea operatorului săgeată.

Pentru a defini acum '*pointerul la structură*' drept un tip de dată generic, se foloseşte sintaxa:

```
typedef
   struct Persoana
   {
   char Nume[24];
   char Prenume[32];
   char Adresa[64];
   int ZiNastere;
   char LunaNastere[12];
   int AnNastere;
   }
*pPERSOANA;
```

```
struct Persoana pers1;
pPERSOANA point1;
point1 = &pers1;
```

1.2.3. Utilitatea pointerilor la structuri

Structurile abstracte de date reţin informaţia în colecţiile de informaţii ce o compun, denumite *noduri* şi modelate în C cu ajutorul structurilor. În majoritatea covârşitoare a cazurilor, *nodurile* conţin cel puţin un câmp referinţă (pointer) la tipul nodului (adică o referinţă către tipul structurii cu care a fost modelat nodul respectiv). Prezenţa acestuia asigură caracteristica de *înlănţuire*, adică oferă posibilitatea `legării` între ele a mai multor noduri. Ceea ce se obţine în urma înlănţuirii poartă una dintre denumirile: `stivă`, *listă*, `coadă`, `arbore`. Acestea sunt structurile de date abstracte studiate în semestrul 2 al anului 1.

Într-o variantă simplificată, cele spuse îşi găsesc o variantă de implementare în exemplul următor.

Exemplu:

```
struct Persoana
{
 char Nume[24];
 char Prenume[32];
 Persoana *pPersoana;// asigură legătura cu o structură Persoana
}

int main(void)
{
   struct Persoana pers1, pers2, *point;
   strcpy(pers1.Nume, "Codrescu");
   strcpy(pers1.Prenume, "Adrian");

   strcpy(pers2.Nume, "Stefan");
   strcpy(pers2.Prenume, "Valentin");

   pers1.pPersoana = &pers2;
   pers2.pPersoana = NULL;

   point = &pers1;
   while(point != NULL)
     {
          printf("\n %s, %s \n", point->Nume, point->Prenume);
          point = point->pPersoana;
     }
}
```

1.3. Detalii de programare

Pentru înțelegerea programelor utilizate în lucrare studiați ANEXA lucrării.

1.4. Comenzile compilatorului

Înainte de compilare codul sursă trebuie salvat (cu combinația de taste CTRL+S), sau, echivalent, din meniul File, cu comanda Save as….

Compilarea se va face cu comanda *CTRL+F9*, iar rularea cu comanda CTRL+F10. Compilarea şi rularea se pot executa, succesiv, printr-o singură tastă, anume F9. Acțiunile echivalente din meniu: meniul Execute comanda Compile, respectiv Compile + Run.

Observații:

1. Orice nouă modificare a codului-sursă trebuie urmată obligatoriu de salvarea acestuia (CTRL+S). Abia după aceea pot urma celelalte acțiuni dorite.
2. Dacă nu se realizează aducerea la zi a programului, compilarea şi rularea ce urmează se fac tot pe varianta veche, adică ce de dinaintea ultimelor modificări.

II. DESFĂŞURAREA LUCRĂRII

2.1. Declarația/definiția unei structuri este tratată în programul lab9_1.cpp. Se foloseşte operatorul punct pentru accesul direct (prin numele structurii şi nu printr-un pointer la structură) la câmpurile acesteia. Structura este declarată global.

Renunțați la comentariul de pe linia *17* şi recompilați programul. Notați-vă eroarea generată şi explicați-o. Corectați, in final programul.

2.2. Pentru a afla numărul de octeţi al structurii date ca exemplu în programul anterior, rulați lab9_2.cpp. Se observă că atât înainte cât şi după inițializarea câmpurilor structurii, spațiul său din memorie este neschimbat. Aceasta pentru că în compilatorul folosit (Developer C++) nu s-au făcut optimizări de aliniere în memorie, situație posibilă în anumite compilatoare (dintre care le menționăm pe cele din familia Borland Turbo C/C++).

2.3. Operatorul `typedef` asociat structurilor se studiază în programul `lab9_3.cpp`. Sunt definite două tipuri de structuri, acoperind cele două situații sintactice corecte: prezența sau absența identificatorului (`tag`) structurii. Apare funcția `strcpy()`, funcție de bibliotecă C, ca parte a fișierului antet `string.h`, a cărei utilizare este aceea de a copia un şir de caractere sursă într-unul destinație.

Completați programul şi declarați un nou `tip structură` cu efect local, cu ajutorul căruia să declarați o variabilă de acel tip. Stabiliți valori pentru câmpurile structurii astfel declarate şi afişați-le.

2.4. Tipul pointer la structură este studiat în programul `lab9_4.cpp`. Renunțați la comentariile de pe liniile `19…31` şi recompilați programul. Folosiți noul tip pointer în cadrul funcției `main()` pentru stabilirea valorilor structurii `pers1`. Tot pe baza acestui nou tip pointer combinați sintaxa `C` astfel încât să folosiți pointerul la structură pentru modificarea valorii uni câmp dar în tandem cu operatorul punct şi nu săgeată (care ar fi tipic `accesului indirect`, prin pointeri, la câmpurile unei structuri).

2.5. Simularea unui `tip abstract` de dată (denumit în literatura de specialitate `structură de dată`) este creat şi folosit în programul `lab9_5.cpp`. Structura generică de joacă rolul de nod în noul tip de dată conține un câmp pointer la o structură de acelaşi tip cea, adică la următorul nod. După ce sunt legate între ele două noduri, această `mini-listă` este parcursă în ciclul `while()` din finalul programului.

Completați programul şi declarați alte două structuri (viitoare noduri în lista anterioară) şi adăugați-le la mini-lista creată. Stabiliți valori pentru câmpurile informație ale celor două noi noduri şi parcurgeți lista modificată. Afişați valorile câmpurilor în cele două variante: direct (operatorul punct) şi indirect (operatorul săgeată).

III. ÎNTREBĂRI

3.1. De ce structura este catalogată drept tip de dată `ne-omogen`? Cu ce tip de dată nativ `C` face contrast?

3.2. Când poate lipsi şi când nu identificatorul structurii (denumit `tag`)?

3.3. Poate varia numărul de octeți ai unei structuri, în situația portării unui cod-sursă de pe un sistem bazat pe o anumită familie de procesoare pe un alt sistem având la bază o altă familie de procesoare, dacă:
- compilatorule este acelaşi,
- compilatorul este diferit.

3.4. Poate conține o structură o altă structură `de tip diferit`? Dar de acelaşi tip cu cea de bază?

3.5. Se poate folosi `typedef` la nivel local? Cum este limitat lucrul asupra unei variabile astfel definite, prin comparație cu o variabilă declarată pe baza unui tip global de structură?

3.6. Unde este corect să se folosească operatorul punct şi unde operatorul săgeată pentru accesul la câmpurile unei structuri?

3.7. Poate fi declarat un tip 'pointer la un pointer la o structură' (pointer dublu)? Creați un scurt program `C` care să confirme sau infirme această cerință.

3.8. Cum se poate asigura `legătura` între două structuri, în situația în care, de exemplu, dorim să avem o listă de abonați telefonici?

3.9. Folosind exemplele şi explicațiile din **`Laboratoarele 8 şi 9`**, `alocaţi dinamic` spațiu pentru o structură de tipul `Persoana` de genul celei din programul `lab9_5.cpp`. Numărați octeții pe care aceasta îi ocupă, înainte şi după stabilirea de valori pentru câmpurile acesteia. Afişați valorile stabilite, în variantele directă (operator punct) şi indirectă (operator săgeată).

IV. ANEXA – sursele complete ale programelor

```
// lab9_1.cpp
// Structuri: introducere - declarare, definire.
#include<stdio.h>
#include<string.h>

struct Persoana
{
   char Nume[24];
   char Prenume[32];
   char Adresa[64];
   int ZiNastere;
   char LunaNastere[12];
   int AnNastere;
   };

int main(void)
{
 struct Persoana pers1, pers2;
// struct pers11, pers22;    // eroare!!

 strcpy(pers1.Nume, "Constantinescu");
             // copierea unui sir de caractere in altul.
 pers1.ZiNastere = 24;       // initializarea unui intreg.

// stabilirea prin copiere a unui sir in alt sir
 strcpy(pers2.Adresa, "Str. Mircea Vulcanescu, nr.42, sector 1");
 pers2.AnNastere = 1960;

// afisarea valorilor campurilor initializate
 printf("\n Campuri ale lui 'pers1': \n Nume: %s, \n ZiNastere: \
         %d", pers1.Nume, pers1.ZiNastere);
 puts("");
 printf("\n Campuri ale lui 'pers2': \n Adresa: %s, \n AnNastere: \
         %d", pers2.Adresa, pers2.AnNastere);

// incheiere
 puts("\n Sfarsit..."); scanf("%d", &pers1.ZiNastere); return 0;
}



// lab9_2.cpp
// Structuri: numărul de octeţi.
#include<stdio.h>
#include<string.h>

struct Persoana
{
   char Nume[24];
   char Prenume[32];
   char Adresa[64];
```

```
   int ZiNastere;
   char LunaNastere[12];
   int AnNastere;
};

int main(void)
{
 struct Persoana pers1;

// afisarea numarului de octeti
 printf("\n Structura generica Persoana ocupa in memorie \
            %d octeti.", sizeof(Persoana));
 printf("\n Inainte de initializare: structura pers1 ocupa \
            %d octeti.", sizeof(pers1));

// copierea unui sir de caractere in altul.
 strcpy(pers1.Nume, "Constantinescu");
 pers1.ZiNastere = 24; // initializarea unui intreg.

printf("\n Dupa initializare: structura pers1 ocupa in memorie \
            %d octeti.", sizeof(pers1));

// incheiere
 puts("\n Sfarsit..."); scanf("%d", &pers1.ZiNastere); return 0;
}


// lab9_3.cpp
// Structuri: definiţii de tip.
#include<stdio.h>
#include<string.h>
typedef
   struct Persoana
   {
   char Nume[24];
   char Prenume[32];
   char Adresa[64];
   int ZiNastere;
   char LunaNastere[12];
   int AnNastere;
   }  PERSOANA1;
typedef
   struct
   {
   char Nume[24];
   char Prenume[32];
   char Adresa[64];
   int ZiNastere;
   char LunaNastere[12];
   int AnNastere;
   }
 PERSOANA2;
```

```
int main(void)
{
 PERSOANA1 p1;
 PERSOANA2 p2;

// Afişarea numarului de octeti, si a valorii unui camp
 printf("\n Structura generica Persoana1 ocupa in memorie \
            %d octeti.", sizeof(PERSOANA1));
 strcpy(p1.LunaNastere, "Decembrie");
 printf("\n Dupa initializare: LunaNasterii: %s.", p1.LunaNastere);

// pentru varianta a doua
 printf("\n Structura generica Persoana2 ocupa in memorie \
            %d octeti.", sizeof(PERSOANA2));
 strcpy(p2.Adresa, "Str. X, nr. Y, ap. Z");
 printf("\n Dupa initializare: LunaNasterii: %s.", p2.Adresa);

// incheiere
 puts("\n Sfarsit..."); scanf("%d", &pers1.ZiNastere); return 0;
}


// lab9_4.cpp
// Structuri: definiţii de tip.
#include<stdio.h>
#include<string.h>

typedef
   struct Persoana
   {
   char Nume[24];
   char Prenume[32];
   char Adresa[64];
   int ZiNastere;
   char LunaNastere[12];
   int AnNastere;
   }
 PERSOANA;            // tipul structura

typedef PERSOANA *pPERSOANA;

// variantă
/*
typedef
   struct Persoana
   {
   char Nume[24];
   char Prenume[32];
   char Adresa[64];
   int ZiNastere;
   char LunaNastere[12];
   int AnNastere;
   } *pPERSOANA1;      // tipul pointer la structura
 */
```

```
int main(void)
{
 PERSOANA pers1;
 pPERSOANA point1;

 point1 = &pers1;      // initializarea pointerului la structura

//  Afisarea numarului de octeti, si a valorii unui camp,
// prin pointer
 printf("\n Structura generica Persoana1 ocupa in memorie \
                                %d octeti.", sizeof(PERSOANA));
 strcpy(pers1.LunaNastere, "Decembrie");
 printf("\n Dupa initializare: LunaNasterii: %s.",
                                point1->LunaNastere);
// incheiere
 puts("\n Sfarsit..."); scanf("%d", &pers1.ZiNastere); return 0;
}
```

```
// lab9_5.cpp
// Mini-listă.
#include<stdio.h>
#include<string.h>
struct Persoana
{
 char Nume[24];
 char Prenume[32];
 Persoana *pPersoana; // asigura legatura cu o structura Persoana
};

int main(void)
{
   struct Persoana pers1, pers2, *point;
   strcpy(pers1.Nume, "Codrescu");
   strcpy(pers1.Prenume, "Adrian");

   strcpy(pers2.Nume, "Stefan");
   strcpy(pers2.Prenume, "Valentin");

   pers1.pPersoana = &pers2;
   pers2.pPersoana = NULL;

   point = &pers1;
   while(point != NULL)
    {
       printf("\n %s, %s \n", point->Nume, point->Prenume);
       point = point->pPersoana;
    }

   // incheiere
   puts("\n Sfarsit..."); scanf("%d", &pers1.ZiNastere); return 0;
}
```