LUCRAREA 10

Scopul lucrării *constă în introducerea în lucru a unor noi tipuri de bază native C, anume enumerările şi uniunile, clasificare acestora în omogene şi neomogene, folosirea operatorilor specifici fiecărui tip, precum şi evidenţierea diferenţelor dintre uniuni şi structuri.*

I. OBSERVAŢII TEORETICE

1.1. Enumerări. Definiţie şi sintaxă

Prin enumerare se înţelege o ***mulţime*** de elemente de aceeaşi natură, fiecărui element fiindu-i asociată, prin construcţie, o valoare numerică de tip întreg (pozitivă sau negativă). Numărătoarea începe implicit de la `0`, iar fiecare element următor preia o valoare cu o unitate mai mare (deci consecutivă) faţă de cea precedentă, în funcţie de poziţia în care se află.

Sintaxa este:

```
enum numeEnumerare {camp1, camp2 [, camp3,…]};
```

`camp1, camp2` se numesc *elementele* enumerării. Numărul elementelor este practic nelimitat, putând fi tot atâtea câte valori pozitive ale tipului `long int`, cu semn, adică:

$[-2^{31}, 2^{31}-1]$

Astfel, într-o declaraţie ca:

```
enum ziSaptamana {Luni, Marti, Miercuri, Joi, Vineri,
                                                Sambata, Duminica};
```

s-a declarat enumerarea `ziSaptamana`, cu mulţimea de valori asociată formată din elementele: `Luni, Marti,…, Duminica`. În această construcţie elementul `Luni` capătă valoarea întreagă `0`, iar toate celelalte elemente vor primi un număr cu 1 mai mare decât valoarea atribuită elementului anterior lui. Astfel:

`Marti = 1, Miercuri = 2, …, Duminica = 6.`

Numărarea se poate însă întrerupe de către utilizator, sau se poate stabili o altă valoare de start. Aceasta se face prin impunerea forţată a unei valori pentru un câmp, alta decât cea care îi era atribuită în mod normal. De aici înainte, celelalte elemente ale enumerării îşi preiau valorile după aceeaşi regulă enunţată anterior (deci `secvenţial`), crescând cu o unitate faţă de valoarea vecinului precedent.

De exemplu, prin declaraţia:

```
enum ziSaptamana1 {Luni=1, Marti, Miercuri};
```

se va atribui lui `Marţi` valoarea `2`, iar lui `Miercuri` valoarea `3`.

O renunţare la comentariul de pe `linia 34`, concomitentă cu comentarea `liniei 33` generează o depăşire a gamei de reprezentare întreg lung fără semn (`unsigned long int`) care reprezintă tipul de dată maxim admisibil pentru elementele unei enumerări. Mesajul de eroare este:

lab10_1.cpp:33: overflow in enumeration values at 'Miercuri'

Este deci clară rezolvarea: reducerea valorii de 2Giga (2^{31} = `2 147 483 648`) la valoarea de pe `linia 33`, drept ultimă valoare valabilă pentru tipul de dată întreg lung fără semn. Aceasta are însă ca efect depăşirea în situaţia constantei simbolice *Miercuri*, care nu va putea fi folosită în contextul numerelor întregi, ci doar în tandem cu tipurile reale de dată.

Prin renunţarea la comentariile de pe `liniile 43…52` se încearcă iniţializarea valorii unei constante simbolice - prin definirea prealabilă a unei constante şi atribuirea valorii acestei elementului dorit a fi modificat. Se generează următoarea eroare:

lab10_1.cpp:46: enumerator value for 'Marti' not integer constant

1.2. Lucrul cu elementele unei enumerări. Accesul la elemente

Odată declarată/definită o enumerare, denumirile elementelor acesteia devin `constante simbolice (constante întregi)`, adică entităţi cu care se poate lucra ca atare, prin intermediul numelor lor. Sunt constante deoarece singurul loc unde ele primesc valori este momentul definiţiei enumerării. Adică prezintă un comportament analog constantelor, studiate în **`Laboratorul 2`**.

După declaraţia/definiţia:

```
enum Traseu {statie1, statie2, capat};
```

se pot folosi elementele enumerării (deci ale mulţimii `Traseu`, conform definiţiei) în felul următor:

```
   int intermediar = statie2;
   if(intermediar >statie1 && intermediar < capat) printf("\n Sunt pe
traseu!");
   else    if(intermediar == capat) printf("\n Am incheiat traseul… O
mica pauza!");
          else printf("\n Abia am inceput!");
```

Se observă folosirea elementelor ca şi cum ar fi variabile obişnuite (prin numele acestora). Să nu uităm însă că ele au comportamentul constantelor (au `Lvalue` constantă) şi că valorile posibile sunt întregi (cu sau fără semn).

Prin modificarea valorii variabilei `intermediar` se pot genera toate mesajele prevăzute în `if-else`.

1.3. Operatori folosiți în situația enumerărilor

1.3.1. Operatorul de referențiere (&)

Pentru că elementele unei enumerări sunt constante simbolice (după cum s-a văzut şi din mesajele de eroare anterioare) nu se poate prelua adresa unui element al unei enumerări. Această încercare generează mesajul:

lab10_3.cpp:17: non-lvalue in unary `&'

Obțineți acest mesaj dacă renunțați la comentariul de pe `linia 17` a programului ***lab10_3.cpp***.

Concluzia este clară acum: unei constante simbolice nu i se permite modificarea valorii, deci aceasta nu posedă `Lvalue`, în adevăratul sens al noțiunii (adresa unei variabile în memoria program folosită (şi) în operațiile de atribuire pentru modificarea valorii acelei variabile).

Trebuie făcută o distincție între un pointer la o valoare constantă (de orice tip) şi un pointer constant către o valoare. În primul caz, valoarea variabilei la care punctează pointerul este constantă, iar în a doua situație, se punctează o locație şi acest pointer nu poate fi schimbat (pointerul este deci *constant*).

Dacă se renunță la comentariul de pe `linia 21` se întâlneşte mesajul următor:

lab10_3.cpp:21: assignment of read-only location

Dacă se încearcă schimbarea adresei unui pointer constant (renunțați la comentariul de pe `linia 28`) mesajul de eroare obținut este explicit:

lab10_3.cpp:28: assignment of read-only variable `pSimbolic'

Observați diferența de nuanță între cele două enunțuri de eroare. Primul se referă la o conținutul unei zone de memorie, în sensul unei valori numerice constante, iar cel de-al doilea la o constantă cu rol de *pointer*.

1.3.2. Operatorul `typedef`

După cum s-a văzut în cele două programe precedente se poate lucra cu enumerări locale sau globale. Din moment ce acest lucru este posibil putem folosi `typedef` respectiv pentru tipuri locale sau generice.

Sintaxa se aseamănă cu cea din cazul structurilor:

```
typedef enum denumire{element1, element2, [element3,…] };
…
denumire enum1 [, enum2, … ];
```

Deşi sintaxa este perfect valabilă, utilitatea nu se justifică. Din moment ce constantele simbolice sunt valabile din momentul în care sunt definite, a declara mai multe variabile de acelaşi tip nu are sens, pentru că elementele acestora având deja

valoare (conform comportamentului tipic al enumerării) se generează de fapt cópii identice ale aceleiaşi enumerări. Cum nu există nici un operator prin care să avem acces la elementele unei enumerări, rezultă că de fapt lucrăm cu unele şi aceleaşi constante simbolice, cele ale primei declaraţii a enumerării, acolo unde apare operatorul `typedef`, şi deci unde se anunţă întâia dată componenţa enumerării.

În sursa *lab10_4.cpp*, se observă că, deşi apare declaraţia enumerării `traseu1` pe `linia 16`, aceasta are efect asupra liniilor următoare, câmpurile sale fiind perfect valabile din acest punct înainte. Aceasta pe lângă câmpurile enumerării globale, care sunt valabile peste tot. Chiar dacă ar fi fost tot local, tipul `enum traseu` şi-ar fi păstrat valabilitatea în cadrul funcţiei `main()` din momentul declarării şi până în momentul încheierii lui `main()`. Pentru a observa aceasta decomentaţi `linia 7` şi comentaţi `linia 4`. Observaţi aceleaşi afişări la ecranul utilizator ca şi programul iniţial (cel fără erori). Acestea justifică echivalenţa celor două situaţii.

Dacă vreun tip enumerare este prins în cadrul unui `corp de instrucţiuni`, atunci valabilitatea sa este doar pe parcursul acelui corp de instrucţiuni. Decomentaţi `liniile 47…50` salvaţi şi recompilaţi programul. Veţi întâlni următorul mesaj de eroare:

lab10_4.cpp:48:

(Each undeclared identifier is reported only once for each function it appears in.)

Aceasta pentru că, prin ignorarea acestor comentarii, denumirile `statie_a`, `statie_b` şi `statie_c` nu mai există, cele valabile fiind, respectiv: `statieA`, `statieB` şi `statieC`, generate de enumerarea globală, singura vizibilă în acel punct al programului.

1.4. Uniuni

Un alt tip neomogen de dată, alături de structură, este `uniunea`. Am studiat deja în **`Lucrarea 9`** tipul `structură`. Aţi văzut acolo că o structură este încadrată în clasa tipurilor `neomogene` din cauză că poate conţine tipuri diferite de date sub `acelaşi` nume. Aceasta prin contrast cu vectorii, pentru care tipul de dată este unic, şi stabilit in momentul definiţiei variabilei vector.

Uniunea este similară structurii. Asemănarea provine din următoarele:

- poate conţine tipuri diferite de date sub acelaşi nume;
- accesul la câmpuri se face cu ajutorul aceloraşi operatori de acces:
    - `operatorul punct` în cazul lucrului direct, folosindu-ne de numele variabilei uniune respective;
    - `operatorul săgeată` - pentru situaţia folosirii indirecte a variabilei uniune, prin intermediul unui pointer la aceasta;
- se poate defini un tip uniune atât global cât şi local;
- poate conţine o structură sau o altă uniune;
- spaţiul de memorie al unei uniuni se poate aloca în ansamblu, folosindu-ne tipul de dată cel mai cuprinzător (v. discuţia următoare);

- poate fi atât argument de funcție cât și valoare returnată de o funcție.

Deosebirea majoră față de structuri se referă la spațiul efectiv rezervat câmpurilor uniunii. Acesta este dat de relația:

spatiuUniune = $\max_i$ ***(sizeof(numeCamp_i))***

adică este rezervat în memorie doar spațiul celui mai 'întins' (în octeți) dintre câmpurile uniunii. Prin `numeCamp_i` am notat simbolic numele variabilelor care pot apărea în cadrul uniunii.

De exemplu, pentru uniunea:

```
union Numar
{
   // în notaţia simbolică anterioară, acest câmp este numeCamp_1
   int nrIntreg;

   // în notaţia simbolică anterioară, acest câmp este numeCamp_2
   float nrReal;
}
```

se rezervă **efectiv** în memorie `4B`, adică `sizeof(float)`. Tipul `float` cere cel mai mare număr de octeți dintre cele două câmpuri ale structurii.

1.4.1. Modul de lucru al uniunii

Accesul la unul sau altul dintre câmpurile unei uniuni se poate face atât la un moment dat t_1, cât și la un alt moment, t_2, sau general, la un moment de timp t_j de pe parcursul rulării funcției sau programului în care apare uniunea (am presupus $t_j > \ldots t_2 > t_1$). La fiecare moment de timp, `ACELAŞI` spațiu de memorie va fi interpretat diferit, în funcție de tipul de dată dorit a se accesa la acel moment de timp.

Aceasta ne impune o **observație interesantă**: dacă uniunea conține câmpuri de tipuri de date diferite, accesul la un moment dat influențează simultan **toate** câmpurile, pentru toate momentele de timp ulterioare celui in care s-a făcut accesul la uniune.

În exemplul din programul `lab10_5.cpp` se poate vedea cum semnul unui număr real poate fi schimbat prin intermediul unui câmp de tip `întreg lung fără semn (unsigned long)`. Am folosit câmpuri de tipuri diferite, dar de aceeași lungime în octeți, pentru a putea pe deplin influența valoarea reală, conform standardului de reprezentare reală (`IEEE 754`). Adică se influențează, implicit, așezarea în memorie a entităților care alcătuiesc un număr real (`mantisă`, `exponent` și `semn`).

`nrReal = semn mantisă*baza`$^{\texttt{exponent}}$, unde `semn` poate fi ±

Semnul este codificat pe 1 bit (`MSb`) și este asimilat mantisei, în reprezentarea de mai sus.

Tot în programul `lab10_5.cpp` se arată şi funcţionarea `operatorului săgeată`, pentru lucrul cu *pointeri* asupra uniunii.

1.4.2. Iniţializarea câmpurilor uniunii

Iniţializarea unei uniuni se poate face direct în momentul declarării/definirii, dar NUMAI pentru primul câmp al acesteia, cu o valoare tipică lui:

```
union cursaRATB
{
 unsigned int urban_Rural;          // variabila logică
 char numarLinie[4];                // sir numeric
}
…
cursaRATB c1 = {1}; // doar primul camp se poate iniţializa !
```

O linie de transport în comun poate la un moment dat să-şi schimbe denumirea şi să devină o cursă de tip rural (preorăşenesc). Această situaţie practică se poate modela cu ajutorul unei structuri de tipul celei prezentate anterior. Primul câmp reprezintă o variabilă logică, cu semnificaţia dacă linia respectivă este urbană (`1`) sau rurală (`0`). Programatorul poate modifica această convenţie, dar trebuie să anunţe potenţialul utilizator de convenţia adoptată. Al doilea câmp reprezintă numărul cursei, dat sub formă de şir numeric, adică un şir care va conţine numai cifre. Bineînţeles, se poate concepe o rutină de verificare a faptului că şirul de intrare conţine într-adevăr numai cifre. Nu este cazul acum.

Singurul câmp ce poate fi iniţializat, conform standardului ANSI C este primul, şi în linia declaraţiei variabilei `c1` se vede acest lucru. Câmpul logic este şi el afectat de această iniţializare, dar în mod indirect.

Programul **`lab10_6.cpp`** arată ce valori au fiecare dintre câmpuri în urma acestei iniţializări.

1.5. Detalii de programare

Pentru înţelegerea programelor utilizate în lucrare studiaţi *ANEXA* lucrării.

1.6. Comenzile compilatorului

Înainte de compilare codul sursă trebuie salvat (cu combinaţia 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`. Acţiunile echivalente din meniu: meniul `Execute` comanda `Compile`, respectiv `Compile + Run`.

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

II. DESFĂŞURAREA LUCRĂRII

2.1. O introducere în lucrul cu enumerări este prezentată în programul `lab10_1.cpp`. Se studiază comportamentul prin definiţie al enumerărilor.

Pentru a observa limita maxim admisibilă pentru valorile constantelor simbolice se de-comentează `linia 34`, simultan cu comentarea `liniei 33`. Salvaţi programul, recompilaţi-l şi notaţi efectul generat. Comparaţi mesajul de eroare cu cel din `paragraful 1.1`.

Încercaţi apoi să iniţializaţi o constantă simbolică prin intermediul unei constante căreia i s-a stabilit în prealabil o valoare. Renunţaţi la comentariile de pe `liniile 43…52` şi observaţi mesajul de eroare; comparaţi-l cu cel din paragraful teoretic. Reţineţi aspectul invocat.

2.2. Lucrul cu constantele simbolice se aprofundează în `lab10_2.cpp`. Aici vedeţi cum sunt folosite elementele enumerărilor şi ce comportament tipic le acordă compilatorul.

Modificaţi valoarea variabilei de lucru `intermediar` astfel încât să generaţi, pe rând, toate mesajele din ramurile `if-else`. Studiaţi şi `Laboratorul 4`, pentru detalii asupra `if-else`.

2.3. Operatorul referinţă şi lucrul cu structurile şi pointerii sunt analizate în programul `lab10_3.cpp`.

Renunţând la comentariul de pe `linia 17` este generat un mesaj de eroare explicit, care marchează concluzia asupra lucrului cu pointeri în contextul enumerărilor.

Tot aici se observă deosebirea între un pointer la o 'valoare constantă' şi un 'pointer constant'. Reţineţi mesajul de eroare generat în urma decomentării succesive a `liniilor 21 şi 28`. Notaţi mesajul şi reţineţi cazurile exceptate întâlnite. Reluaţi şi `Laboratoarele 7 şi 8` pentru detalii aspra pointerilor.

2.4. Operatorul `typedef` în situaţia enumerărilor este analizat în programul `lab10_4.cpp`. Sunt studiate atât tipurile globale cât şi locale nou create. Echivalenţa celor două situaţii (local şi global) **`în situaţia particulară a enumerărilor`** se observă prin comentarea linie 4 şi

renunțarea la comentariul de pe `linia 7`. Rețineți acest comportament particular.

Lucrul cu corpurile de instrucțiuni generează efectul cunoscut. Renunțați la comentariile de pe `liniile 47…50` salvați şi recompilați. Observați efectele. Comparați erorile generate de compilator cu cele din `paragraful 1.3.2`. Notați-le şi rețineți efectul unui corp de instrucțiuni (invariabil în orice program C, indiferent de tipurile de date utilizate). Reluați `Laboratorul 3` pentru detalii asupra corpurilor de instrucțiuni.

III. ÎNTREBĂRI

3.1. Ce echivalent matematic are o enumerare? Dați două exemple sugestive.
3.2. Elementele unei enumerări pot fi modificate? Cu ce scop şi în ce variantă corectă sintactic?
3.3. Poate lipsi *tag*-ul unei enumerări?
3.4. Ce erori se generează în situația utilizării pointerilor? Cum se comportă elementele enumerării în prezența operatorului referință?
3.5. Explicați mecanismul care interzice modificarea valorii unui element al enumerării, altfel decât prin valorile de inițializare.
3.6. Există tipuri enumerare locale? Dar globale?
3.7. Odată declarată o enumerare, care este durata de viață a elementelor sale? Cum poate fi influențată această durată de viață?
3.8. Poate fi o enumerare câmp al unei structuri? Cum se poate tipări o constantă simbolică în acest caz? Este utilă o astfel de construcție?

IV. ANEXĂ - sursele complete ele programelor

```
// lab10_1.cpp
// Enumerări: introducere.
// Atentie! Se lucreaza numai cu enumerari locale.

#include<stdio.h>
#include<math.h>

typedef const unsigned long int CULI;

int main(void)
{
// enumerare clasica
 enum ziSaptamana {Luni, Marti, Miercuri, Joi, Vineri, Sambata, Duminica};
 enum {unu, doi};// lipseste tag-ul si numele variabilei enumerare.

 printf("\n Prima zi din saptamana (Luni) are aici valoarea \
                         constanta: %u, %i, %ul", Luni, Luni, Luni);
 printf("\n Ultima zi din saptamana are valoarea constanta: %u, %i, \
                                %ul", Duminica, Duminica, Duminica);
 printf("\n Elementul >unu< are valoarea constanta: %u, %i, %ul",
                                                    unu, unu, unu);

// Schimbarea ordinii din numerotare. folosirea corpului de
// instructiuni
 {
 enum ziSaptamana1 {Luni=1, Marti, Miercuri};
 printf("\n\n Prima zi din saptamana are acum valoarea constanta: \
                                     %u, %i, %ul", Luni, Luni, Luni);
 printf("\n ...iar Miercuri are valoarea constanta: %u, %i, %ul",
                                       Miercuri, Miercuri, Miercuri);
 }

// fortarea valorii maxime a unei constante simbolice: 2^31-1
 {
  enum ziSaptamana2 {Luni, Marti = 2147483646, Miercuri};
//  enum ziSaptamana3 {Luni, Marti = 2147483647, Miercuri};\
  printf("\n\n Prima zi din saptamana are din nou valoarea
                         constanta: %u, %i, %ul", Luni, Luni, Luni);
  printf("\n ...iar Miercuri are acum valoarea constanta: %u, %i, \
                                %ul", Miercuri, Miercuri, Miercuri);
 }

//  O alta eroare: initializarea printr-o valoare calculata a unei
// constante simbolice
/*
 {
  CULI valoare = (CULI)pow(2,20)-8;
  enum ziSaptamana4 {Luni, Marti = valoare, Miercuri};
  printf("\n\n Prima zi din saptamana are din nou valoarea \
                         constanta: %u, %i, %ul", Luni, Luni, Luni);
  printf("\n ...iar Miercuri are acum valoarea constanta: %u, %i, \
```

```
                                %ul", Miercuri, Miercuri, Miercuri);
 }
*/
// incheiere
 int i;
 scanf("%i", &i); return 0;
}


// lab10_2.cpp
//  Enumerări_2: tratarea constantelor simbolice drept constante
// în expresii.
#include<stdio.h>

enum Traseu {statie1, statie2, capat}; // enumerare globala

int main()
{
 int intermediar = statie2;

 if(intermediar >statie1 && intermediar < capat)
   printf("\n Sunt pe traseu!");
 else     if(intermediar == capat)
                printf("\n Am incheiat traseul... Pauza!");
          else
                printf("\n Abia am inceput!");

// Incercarea de preluare a adresei unei constante simbolice
/*
 unsigned int *pSimbolic = &statie1;
 printf("\n Adresa primului elment al enumerarii: %x, %u, %p",
                                    pSimbolic, pSimbolic, pSimbolic);
*/
// incheiere
 int i;
 scanf("%i", &i); return 0;
}


// lab10_3.cpp
//  Operatorul de referinţă (&) NU se poate folosi in cazul
// enumerărilor.
#include<stdio.h>

enum Traseu {statie1, statie2, capat}; // enumerare globala

int main(){
 int intermediar2 = statie2;

 if(intermediar2 > statie1 && intermediar2 < capat)
   printf("\n Sunt pe traseu!");
 else     if(intermediar2 == capat)
                printf("\n Am incheiat traseul... Pauza!");
```

```
          else printf("\n Abia am inceput!");

 int intermediar1 = statie1;

// incercarea de preluare a adresei unei constante simbolice
// linia urmatoare genereaza eroare daca se renunta la comentariu
// int *pSimbol = & statie2; // incercare eronata

// 'pointer constant' la un intreg
 int * const pSimbolic = &intermediar2;

// pointer la un 'intreg constant'
 int const *pSimbolic1 = &intermediar1;
 printf("\n Adresa primului element al enumerarii: %x, %u, %p",
                                  pSimbolic, pSimbolic, pSimbolic);
 printf("\n Urmatoarea valoare din memorie: %u, %x, %p",
                                                  *(pSimbolic+1));

//  Urmatoarea afisare arata cum se poate modifica o valoare din
// memorie, dar nu se poate modifica niciodata (!) valoarea unei
// constante simbolice
 printf("\n Modificarea valorii din memorie: %u, %x, %p",
                                                    *pSimbolic+=1);
 printf("\n Valoarea constantei simbolice: %i", statie2);

//  Incercare eronata de schimbare a locatiei catre care arata un
// 'pointer constant'
//  Diferenta intre un 'pointer constant' si un 'pointer la o
// constanta'
// pSimbolic = &intermediar1;

// incheiere
 int i;
 scanf("%i", &i); return 0;
}


// lab10_4.cpp
// Tratarea constantelor simbolice drept constante in expresii.
#include<stdio.h>

// tip enumerare, global
typedef enum traseu{statieA, statieB, statieC} T;

int main()
{
// tip enumerare, local
// typedef enum traseu{statieA, statieB, statieC};
 enum traseu traseu1;
 int intermediar = statieB;

 if(intermediar > statieA && intermediar < statieC)
             printf("\n Sunt pe traseu!");
 else     if(intermediar == statieC)
```

```
                printf("\n Am incheiat traseul... Pauza!");
          else printf("\n Abia am inceput!");

 printf("\n Valoarea variabilei intermediar: %i", intermediar);

//  Corp de instructiuni cu variabila locala. Efectul acestei
// variabile este limitat pe parcursul acestui corp de
// instructiuni. Vedeti acest efect prin decomentarea liniilor
// 47...50
 {
  // tip enumerare, local
  typedef enum traseu{statie_a, statie_b, statie_c};
  enum traseu traseu1;
  int intermediar = statie_b;

  if(intermediar > statie_a && intermediar < statie_c)
             printf("\n Sunt pe traseu!");
  else    if(intermediar == statie_c)
                printf("\n Am incheiat traseul... Pauza!");
          else printf("\n Abia am inceput!");

 printf("\n Valoarea variabilei intermediar: %i", intermediar);
 }

//
/*
 // tip enumerare, local
 typedef enum traseu1{statie1=1, statie2, capat};
 int intermediar1 = capat;

 enum traseu1 traseu_1;
 if(intermediar1 >statie1 && intermediar1 < capat)
             printf("\n Sunt pe traseu!");
 else     if(intermediar1 == capat)
                printf("\n\n Am incheiat traseul... Pauza!");
          else printf("\n\n Abia am inceput!");
 printf("\n Valoarea variabilei intermediar1: %i", intermediar1);
*/
 intermediar = statieC;
 printf("\n\n Valoarea variabilei intermediar: %i", intermediar);
/*
 intermediar = statie_c;
 printf("\n\n Valoarea variabilei intermediar: %i", intermediar);
*/

// incheiere
 int i;
 scanf("%i", &i); return 0;
}
```

```
// lab10_5.cpp
// Uniuni_1 - spaţiu de memorie, accesul la câmpuri şi pointeri la
// uniuni.
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<math.h>

// tipul de data uniune denumit NUMAR, si tipul pointer la uniunea NUMAR
typedef union
{
 unsigned int nrNatural;
 float nrReal;
} NUMAR, *pNumar;

int main()
{
 NUMAR n1;
 unsigned int nr;
 pNumar pN1 = &n1, pN2;
// Este obligatorie initializarea poiterului;
// el nu capata automat adresa variabilei n1, doar pentru ca
// puncteaza la un tip uniune identic cu al lui n1.

 printf("\n Spatiul rezervat uniunii (octeti): %u", sizeof(n1));

// Folosirea combinata a operatorilor punct si săgeata
 n1.nrReal = -0.22*pow(10,2);
 printf("\n Campurile primei uniuni: %u, %f", n1.nrNatural,
                                                    n1.nrReal);
 n1.nrNatural = 16;
 printf("\n Campurile primei uniuni: %u, %f", n1.nrNatural,
                                                  (*pN1).nrReal);

// Alocarea de spatiu pentru o noua variabila de tip uniunie
 pN2 = (pNumar) calloc(1, sizeof(NUMAR));
 if(pN2 == NULL)
      { printf("\n main: Eroare alocare spatiu uniune... Stop!");
        exit(1);}

 printf("\n Campurile uniunii (imediat dupa alocare dinamica): %u, \
                                %f", pN2->nrNatural, (*pN2).nrReal);

// utilizarea operatorului sageata
 pN2->nrReal = -12.012;

// semn: minus, mantisa = .12012, exponent +2 (in baza 10)
 printf("\n Campurile uniunii (dupa alocare dinamica): %u, %f",
                                    pN2->nrNatural, (*pN2).nrReal);
// modificarea unui camp existent (efectele sunt prezentate prin
// intermediul unei variabile externe, anume nr)
 nr = (int) pN2->nrReal;
 printf("\n Valoarea numarului nr (inainte): %u, %x, %f",
                                              nr, nr, (float)nr);
```

```
pN2->nrNatural = 0xCFFFFFFF;        // numar fara semn
 printf("\n Campurile uniunii alocate (modificarea a doua): %u, %x, \
             %f", pN2->nrNatural, (*pN2).nrNatural, (*pN2).nrReal);
//
 nr &= pN2->nrNatural;
 printf("\n Valoarea numarului (dupa): %u, %x, %f", nr, nr,
                                                    (float)nr);

// urmatorul comentariu genereaza eroare de compilare pentru ca nu
// corespunde tipului de data al campului ce se doreste modificat,
// prezent in definitia tipului structura pe care il folosim
// pN2->nrNatural = -12012;  // eroare de compilare

// eliberarea zonei de memorie alocate anterior
 free(pN2);

// incheiere
 int i;
 scanf("%i", &i); return 0;
}


// lab10_6.cpp
// Uniuni_2 - iniţializare şi efecte.
#include<stdio.h>
#include<string.h>
#include<conio.h>

union cursaRATB
{
 unsigned int urban_Rural;    // variabila logica
 char numarLinie[4];          // sir numeric
};

int main()
{
 cursaRATB c1 = {1};   // doar primul camp se poate initializa

 printf("\n Spatiul rezervat uniunii: %u", sizeof(c1));

// Efectul initializarii asupra sirului
 printf("\n Campurile uniunii: %u, %x, %s",
                    c1.urban_Rural, c1.urban_Rural, c1.numarLinie);

// Stabilirea unei valori pentru sirul numeric. Efectul asupra
// campului logic
 strcpy(c1.numarLinie, "336");
 printf("\n Campurile uniunii: %u, %x, %s",
                    c1.urban_Rural, c1.urban_Rural, c1.numarLinie);

// incheiere
 int i;
 scanf("%i", &i); return 0;
}
```