UNIVERSITATEA TEHNICĂ A MOLDOVEI

PROGRAMAREA CALCULATOARELOR
TIPURI DE DATE DEFINITE DE UTILIZATOR
Prelegere

Kulev Mihail, dr., conf. univ.

Stimați studenți și stimată audiență!

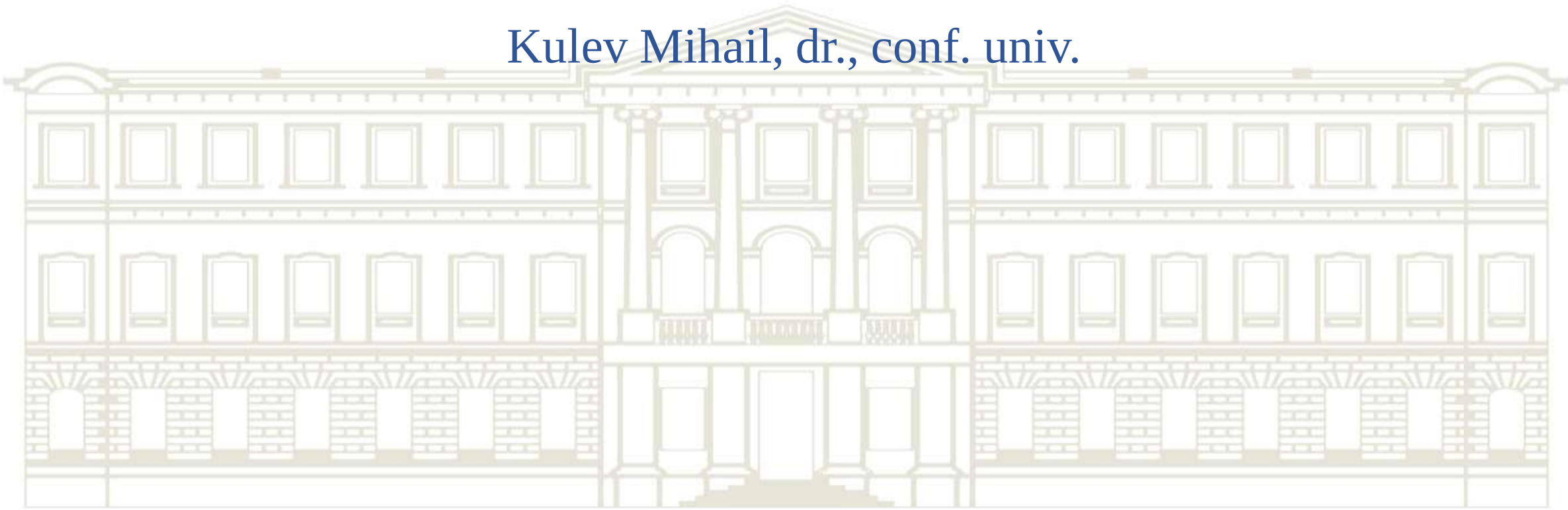
Mă numesc Kulev Mihail sunt doctor, conferențiar universitar la Universitatea Tehnică a Moldovei. Din cadrul cursului PROGRAMAREA CALCULATOARELOR Vă propun spre atenția Dumneavoastră prelegerea cu tema:

”TIPURI DE DATE DEFINITE DE UTILIZATOR”

UNIVERSITATEA TEHNICĂ A MOLDOVEI

PROGRAMAREA CALCULATOARELOR
TIPURI DE DATE DEFINITE DE UTILIZATOR
Prelegere

Kulev Mihail, dr., conf. univ.



UNIVERSITATEA TEHNICĂ A MOLDOVEI

TIPURI DE DATE DEFINITE DE UTILIZATOR

Conținutul prelegerii

1. Noțiuni de tipuri de date definite de utilizator.

Vom afla care sunt tipurile de date în limbajul C numite tipuri de date definite de utilizator.

2. Structuri și operații cu structuri.

Vom determina ce reprezintă tipul de date structură și vom demonstra exemple și modalități de declarare, inițializare, utilizare și prelucrare a variabilelor de tip structură și pointeri la structură.

3. Câmpuri de biți.

Vom afla ce reprezintă tipul de date câmpuri de biți bazat pe tipul structură și modalități de declarare și utilizare a datelor de acest tip.

4. Uniuni.

Vom afla ce reprezintă tipul de date uniune și vom demonstra exemple și modalități de declarare, utilizare și prelucrare a datelor de tip uniune.

5. Enumerări.

Vom afla ce reprezintă tipul de date enumerare și vom demonstra exemple și modalități de

declarare, utilizare și prelucrare a datelor de acest tip.

1. Noțiuni de tipuri de date definite de utilizator

Limbajul de programare C oferă utilizatorului facilităţi de a prelucra atât datele scalare (singulare, isolate), cât şi pe cele grupate (agregate, structurate). Un exemplu de grupare a datelor de acelaşi tip reprezintă tablourile. Datele de tipuri predefinite şi derivate (tablouri și pointeri), utilizate în limbajul C și prezentate la lecțiile anterioare, nu sunt suficiente. Informaţia prelucrată în programe poate fi organizată și în date de alte tipuri diferite. Pentru a putea descrie aceste date limbajul C oferă programatorului posibilităţi de definire a unor tipuri de date proprii numite **tipuri de date definite de utilizator** cu ajutorul:

structurilor - permit gruparea unor obiecte (date) de tipuri diferite, referite printr-un nume comun;

câmpurilor de biţi - membrii ai unei structuri pentru care se alocă un grup de biţi în interiorul unui octet (sau cuvânt) de memorie;

uniunilor - permit utilizarea în comun a unei zone de memorie de către mai multe obiecte de diferite tipuri;

enumerărilor - sunt liste de identificatori (nume simbolice) cu valori constante, întregi.

2. Structuri

De multe ori datele folosite într-un program caracterizează diferite obiecte/fenomene (entități), iar aceste entități au mai multe caracteristici (sau atribute) decât un simplu număr sau un șir de caractere. De exemplu, un punct în plan este definit de coordonatele sale (x, y), o persoană poate fi definită prin nume, dată de naștere și e-mail, o carte prin titlu, autor, editură, domeniu. În toate aceste situații, procesarea unei entități poate implica operații asupra tuturor componentelor sale, posibil, de diferite tipuri. Ar fi mult mai simplu dacă am putea grupa toate aceste informații specifice unei entități într-o singură variabilă. Acest lucru este posibil în C, folosind cuvântul cheie **struct**, prin care putem defini o structură

(sau inregistrare).

Structuri

Structurile sunt tipuri de date în care putem grupa mai multe date (variabile) de tipuri diferite (spre deosebire de tablouri, care conţin numai date de același tip). Un tip de structură se poate defini astfel:

```
struct nume_structura {

declaratii_de_variabile;
};
```

Exemplu:

```
struct student {
    char nume[40];
    int an;
    float medie;
};
```

Variabilele **nume, an, medie,** declarate în interiorul structurii se numesc “câmpurile” (sau membrii) structurii.

Structuri

Cuvântul cheie **struct** definește o nouă structură (un nou tip) de date având numele dat și conținând între acolade **{ }** toate definițiile componentelor sale. În interiorul unei structuri se pot defini doar variabile, nu și funcții. La sfârșitul structurii se pune punct-virgulă(**;**). Structurile pot fi definite oriunde într-un program, inclusiv în interiorul funcțiilor, dar, în general, se definesc în exteriorul lor, pentru a fi valide și accesibile oriunde în program. La fel ca în cazul variabilelor și a funcțiilor, structurile trebuie mai întâi definite și apoi folosite. Componentele unei structuri pot fi diferite după complexitate:

variabile simple, vectori, matrice, pointeri, alte structuri.

Structuri

Definiția unei structuri poate fi urmată imediat de declararea variabilelor de tip structură, cu forma generală :

```
struct [nume_structura] {
declaratie_de_variabile;
declaratie_de_variabile;
declaratie_de_variabile;
. . .
}[una sau mai multe variabile de structură];
```

Exemplu:

```
struct student {
 char nume[40];
 int an;
 float medie;
 } s1,s2,sv[10],*ps;
```

Unde între **[ ]** se află elementele opționale.

Structuri

După definirii tipului structură declararea şi iniţializarea unei variabile de acest tip se poate face astfel:

```
struct student s1 = {"Popescu Ion", 3, 9.25};
```

Pentru simplificarea declaraţiilor de variabile de tip structură, putem asocia unei structuri un singur nume de tip de date utilizînd instrucțiunea de redenumire de tip **typedef**:

```
typedef struct student Student;
Student s1 = {"Popescu Ion", 3, 9.25};
```

Structuri

Definirea structurii poate fi scrisă înainte de (sau inclusă (scrisă) în) instrucțiunea de redenumire de tip **typedef** (asociere de sinonim):

```
struct student {
  char nume[40];
  int an;
  float medie;
};
typedef struct student Student;
//sau
typedef struct student {
  char nume[40];
  int an;
  float medie;
} Student;
```

```
int main() {
// Ambele declaratii de mai
// jos sunt valide
struct student s1,s2,sv[10],*ps;
// sau Student s1,s2,sv[10],*ps;
}
```

Structuri. Observații

- Ordinea enumerării câmpurilor unei structuri nu este importantă, deoarece ne referim la câmpuri prin numele lor. Se poate folosi o singură declaraţie de tip pentru mai multe câmpuri (lista câmpurilor de același tip).
- În structuri diferite pot exista câmpuri cu acelaşi nume, dar în aceeaşi structură numele de câmpuri trebuie să fie diferite.
- Nu există constante de tip structură, dar este posibilă iniţializarea la declarare a unor variabile structură.
- Un câmp al unei structuri poate fi de tip structură, dar nu aceeaşi cu cea definită - însă în cadrul structurii poate fi declarat un câmp pointer la structura data (aspect care este utilizat la implementarea listelor înlănțuite):

```
typedef struct persoana { char nume[20];
struct data{int zi,an,luna;} data_nasterii;  //camp de tip structurâ
} Persoana;
typedef struct nod { int info;
struct nod * urm; //camp pointer la structura definita
} Nod;
```

Accesul la câmpurile (membrii) unei variabile de tip structură

se face utilizând operatorul de selecție sau operatorul punct (.)

```
typedef struct student
{char nume[40]; int an; float medie;} Student;
Student s1, s2, s3;
s1.an = 3;  s2.medie = 9.5; strcpy(s3.nume, "Popescu Ion");
```

Deoarece structurile se prelucrează frecvent prin intermediul pointerilor, a fost introdus operatorul săgeată (->), care combină operatorul de indirectare (*) cu cel de selectie (.).

```
Student *ps = (Student *)malloc(sizeof(Student));
(*ps).medie = 9.31; // operatorii (*) și (.)
// altă modalitate mai simplă şi mai des folosită:
ps->medie = 9.31; // operatorul (->)
```

Atribuirile de structuri se pot face astfel:

```
typedef struct complex
{
float re;
float im;
} Complex;
Complex c1, c2;
. . .
c2 = c1;
```

! Prin această atribuire se realizează o copiere bit cu bit

a cîmpurilor lui c1 în c2.

Diferenţa dintre copierea structurilor şi copierea pointerilor

! Dacă declaraţi pointeri la structuri, nu uitaţi să alocaţi memorie pentru aceştia înainte de a accesa câmpurile structurii. Nu uitaţi să alocaţi şi câmpurile structurii, care sunt pointeri înainte de utilizare, dacă este cazul. De asemenea, fiţi atenţi şi la modul de accesare al câmpurilor.

Pentru exemplificarea diferenţei dintre copierea structurilor şi copierea pointerilor să studiem următorul exemplu:

```
typedef struct person {char* name; int age;} Person;

Person s1, s2; char *num = "Popescu Ion";

s1.name = (char*)malloc((strlen(num)+1)*sizeof(char));

strcpy(s1.name, num); s1.age = 35;

s2 = s1; // ! copiere bit cu bit: - s2.name == s1.name și s2.age == s1.age

sau s2.name=(char*)malloc(sizeof(char)*(strlen(num)+1));

strcpy(s2.name, num); s2.age = 35;

// s2.name != s1.name și s2.age==s1.age 'copiere' corectă
```

Utilizarea tipurilor structură

Un tip structură poate fi folosit în:

- declararea de variabile structuri sau pointeri la structuri
- declararea unor parametri formali de funcţii (structuri sau pointeri la structuri)
- declararea unor funcţii cu rezultat (valoarea returnabilă) de un tip structură sau pointer la structură

O variabilă structură nu poate fi citită sau scrisă direct, ci doar prin intermediul câmpurilor!

Operaţiile posibile cu variabile de tip structură sunt:

- aplicarea operatorilor: **&** - de referențiere și **`sizeof`** – de dimensiune
- atribuirea între variabile de acelaşi tip structură:

  - se folosește operatorul de atribuire **=** , singurul operator care admite operanzi de tip structură
- nu se pot folosi alţi operatori ai limbajului
- trebuie definite funcţii pentru operaţii cu structuri: comparaţii, operaţii aritmetice, operaţii de citire-scriere etc.
- transmiterea ca argument efectiv la apelarea unei funcţii
- transmiterea ca rezultat al unei funcţii, într-o instrucţiune return.

Principalele avantaje ale utilizării unor tipuri structură sunt:

- Programele devin mai explicite dacă se folosesc structuri în locul unor variabile separate.
- Se pot defini tipuri de date specifice aplicaţiei, iar programul reflectă mai bine mediul aplicaţiei.
- Se poate reduce numărul de parametri al unor funcţii prin gruparea lor în parametri de tipuri structură şi, deci se simplifică utilizarea acelor funcţii.
- Se pot utiliza structuri de date extensibile formate din variabile structură alocate dinamic şi legate între ele prin pointeri (liste înlănţuite, arbori ş.a).

Funcţii cu parametri şi/sau rezultat structură

O funcţie care produce un rezultat de tip structură poate fi scrisă în două moduri, care implică şi utilizări diferite ale funcţiei:

- funcţia are rezultat (valoarea returnabilă) de tip structură:

```
typedef struct complex { float re; float im;} Complex;
// citire numar complex (varianta 1)
Complex readx (void) {Complex c; scanf (“%f%f”,&c.re, &c.im); return c;}
Complex a[100]; for (i=0;i<n;i++) {a[i]=readx();} //utilizare
```

- funcţia este de tip void şi depune rezultatul la adresa primită ca parametru (pointer la tip structură):

```
// citire numar complex (varianta 2)
void readx(Complex * px) {// px pointer la o structură Complex
scanf (“%f%f”, &px->re, &px->im); }
Complex a[100]; for (i=0;i<n;i++) {readx (&a[i]);} // utilizare
```

Structuri. Gruparea variabilelor

Uneori, mai multe variabile descriu împreună un anumit obiect şi trebuie transmise la funcţiile ce lucrează cu obiecte de tipul respectiv. Gruparea acestor variabile într-o structură va reduce numărul de parametri şi va simplifica apelarea funcţiilor. Exemple de obiecte definite prin mai multe variabile: obiecte geometrice (puncte, poligoane ş.a.), date calendaristice şi momente de timp,

structuri de date - vectori, matrice, liste etc.

Exemplu de grupare într-o structură a unui vector şi a dimensiunii lui

```
typedef struct vector{int vec[100]; int dim;} Vector; // sau int* vec;
// afişare vector
void scrvec(Vector v){int i; for(i=0;i<v.dim;i++) printf("%d\t",v.vec[i]);
printf ("\n");} // sau Vector *v
// elementele comune din 2 vectori
Vector elcomun(Vector *a, Vector *b){; int i,j,k=0;
for(i=0;i<a->dim;i++) {for (j=0;j<b->dim;j++) Vector c
{if (a->vec[i]==b->vec[j]) {c.vec[k++]=a->vec[i];}}} c.dim=k; return c;}
```

! Pentru structurile care ocupă un număr mare de octeţi este mai eficient să se transmită ca parametru la funcţii adresa structurii (un pointer) în loc să se copieze conţinutul structurii la fiecare apel de funcţie şi să se ocupe loc în memorie, chiar dacă funcţia nu face nici o modificare

în structură a cărei adresă o primeşte.

Exemplu: Să se definească o structură Point pentru un punct geometric 2D şi o structură Rectangle pentru un dreptunghi definit prin colţul stânga sus şi colţul dreapta jos. Să se iniţializeze şi să se afişeze o variabilă de tip Rectangle. Tipul Punct este îmbrăcat în tipul Rectangle.

```
#include <stdio.h>
typedef struct point {int x,y;} Point;
typedef struct rectangle {Point
topLeft;Point bottomRight;} Rectangle;
int main()
{Point p1,p2; p1.x=0; // p1 la (0, 3)
p1.y = 3;  p2.x = 4;  // p2 la (4, 0)
p2.y = 0;
printf("p1 la ( %d, %d)\n", p1.x,
p1.y);
printf("p2 la ( %d, %d)\n", p2.x,
p2.y);
Rectangle rect;  rect.topLeft = p1;
rect.bottomRight = p2;
printf("Stanga sus la(%d,%d)\n",
rect.topLeft.x, rect.topLeft.y);
printf("Dreapta jos la (%d,%d)\n",
rect.bottomRight.x,rect.bottomRight.y
);
return 0;
}
```

3. Câmpuri de biți

Limbajul C oferă posibilitatea de prelucrare a datelor la nivel de bit. De multe ori, se utilizează date care pot avea doar 2 valori (0 sau 1), cum ar fi datele pentru controlul unor dispozitive periferice sau datele de valori mici.

Limbajul C, prin tipul de date câmpuri de biţi, permite accesul la unul sau un grup de biţi, dintr-un octet sau cuvânt.

Pentru a avea acces la biţi, limbajul C foloseşte o metodă bazată pe tipul structură. De fapt, un câmp de biţi este un tip special de membru al unei structuri căruia i se specifică tipul si numărul efectiv de biţi. Forma generală de definire a unui câmp de biţi este:

```
struct nume_structura { tip1 nume1:lungime1; tip2 nume2:lungime2;
. . .
tipn numen:lungimen;
} lista_variabile;
```

unde:

- **`nume_structura`** – este numele structurii;
- **`tip`** – este tipul câmpului de biţi şi care trebuie să fie de tip int, short, unsigned sau signed. Câmpul de lungime unu trebuie să fie declarat ca fiind unsigned deoarece un singur bit nu poate avea semn.
- **`lungime`** – reprezintă numărul de biţi ai câmpului.

Câmpuri de biți

Dacă un câmp de biţi este specificat ca int sau signed, bitul cel mai semnificativ va fi bitul de semn. Numele câmpului de biţi şi lungimea lui sunt separate prin două puncte (:).

Câmpurile de biţi sunt utile în următoarele situaţii:

-dorim să acumulăm mai multe informaţii, de obicei de tip boolean (adevărat/fals – 1/0) într-un spaţiu cât mai mic;

-pentru a transmite informaţii codificate unor echipamente;

-pentru a cripta informaţii într-un octet sau cuvânt.

De exemplu:

```
struct camin {   // Toate aceste informaţii sunt stocate într-un singur octet (4 + 1 + 1 + 2 = 8 biţi). În mod normal, dacă nu
unsigned camera:4 ; // id camerei //  se folosea câmpul de biţi, ar fi fost necesar cel puțini 4 octeţi.
unsigned stare:1 ; // ocupate 1, libere 0
unsigned plata:1 ; // 1 plătit, 0 restanţă
unsigned perioada:2;} // perioada in luni închiriată
```

Câmpuri de biți

Un membru al unui câmp de biţi poate fi accesat ca orice element al unei structuri.

De exemplu:

```
struct camin nr1;

nr1.camera=14;
```

atribuie valoarea 14 câmpului camere a variabilei **nr1** de tip câmpuri de biți

Sau instrucţiunea:

```
if(nr1.stare) printf("camera este ocupatã"); else printf("camera este liberã");
```

Nu este necesar ca biţii dintr-un octet sau cuvânt să fie toţi ocupaţi .

```
struct alfa {
  int x: 10;
  int y: 2;
};
```

Câmpuri de biți

Compilatorul stochează structura definită câmp de biţi pe cea mai mică unitate de memorie octet sau cuvânt.

În aceeaşi structură pot coexista atât câmpuri de biţi cât şi variabile normale, membri ai structurii.

```
struct blocuri {
  char bloc[10]; // nume bloc
  float plata; // cheltuieli lunare
  unsigned achitat:1; // 1 – da, 0 – nu
  unsigned intarziere:4; // număr de luni
} bl;
```

Dacă accesul la structură se face printr-un pointer, se va folosi operatorul săgeată (->)

```
struct alfa{
  unsigned x: 3;
  unsigned y: 2;
} beta, *p; p=&beta;
```

p->x=5; // atribuie valoarea 5 câmpului de biți x a variabilei beta

Câmpuri de biți

Nu este necesar de a determina nume tuturor biţilor dintr-un câmp de biţi:

```
struct stare { unsigned x: 1;
               unsigned: 4;
               unsigned y ; };
```

Utilizarea câmpurilor de biţi impune următoarele restricţii:
Tipul membrilor poate fi int, short, signed sau unsigened int.
Lungime câmpului este o constantă întreagă din intervalul [0, lungime octet sau cuvănt];
Nu se poate obţine adresa unui membru al unui câmp de biţi, adică nu se poate folosi operatorul de adresă &.

Nu se pot organiza tablouri de câmpuri de biţi.
Datorită restricţiilor pe care le impune folosirea câmpurilor de biţi, cât şi datorită faptului că aplicaţiile care folosesc astfel de structuri de date au o portabilitate extrem de redusă (organizarea memoriei depinde de sistemul de calcul), se recomandă folosirea câmpurilor de biţi cu atenție, doar în situaţiile în care se face o economie substanţială de memorie.

4. Uniuni

O altă facilitate a limbajului C referitoare la tipurile structurate constă în posibilitatea de a crea mai multe variabile, care partajează aceeaşi zonă de memorie. Facilitatea este utilă în situaţii în care se doreşte accesarea aceleiaşi zone de memorie ca tipuri diferite de date. O uniune (union) se declară utilizând o sintaxă similară structurilor, dar între uniuni şi structuri există o diferenţă majoră: 1. - în cazul unei structuri, fiecare câmp are propria sa zonă de memorie alocată în cadrul structurii; dimensiunea în octeţi a unei structuri este egală cu suma dimensiunilor în octeţi a tuturor câmpurilor; 2. - în cazul unei uniuni, toate câmpurile partajează aceeași zonă de memorie; dimensiunea în octeţi a unei

uniuni este egală cu dimensiunea în octeţi a celui mai mare câmp.

Uniuni

- Defineşte un grup de variabile care nu se memorează simultan ci alternativ.
- Se pot memora diverse tipuri de date la aceeaşi adresă de memorie.
- Alocarea de memorie se face în funcţie de variabila ce necesită maxim de memorie.
- O uniune face parte de obicei dintr-o structură, care mai conţine şi un câmp discriminant, care specifică tipul datelor memorate (alternativa selectată la un moment dat).

Declararea uniunilor se face folosind cuvântul cheie **union**.

Cuvântul cheie **union** se foloseşte la fel ca cuvântul cheie **struct**.

Uniuni

Sintaxa pentru declararea unei uniuni este similară cu declararea unei structuri:

```
union [nume_uniune] {
declaratie_de_variabile;
declaratie_de_variabile;
. . .
}[una sau mai multe variabile de uniune];
```

Unde între [] se află elementele opționale.

Exemplu:

```
union intreg {
    int val;
    char octeti[2];
} u1,u2,uv[10],*pv;
```

Câmpurile se accesează direct cu operatorul . sau -> în cazul accesării indirecte prin intermediul unui pointer.

Uniuni

Uniunile sunt utilizate pentru a economisi memoria (se refoloseşte aceeaşi zonă de memorie pentru a stoca mai multe variabile).

Atunci când scriem ceva într-o uniune (de exemplu când facem o atribuire de genul u1.val = 7), ceea ce citim apoi trebuie să fie de acelaşi tip, altfel vom obţine rezultate eronate (adică trebuie să utilizam u1.val, nu u1.octet[0] sau u1.octet[1]). Programatorul trebuie să ţină cont de tipul variabilei, care este memorată în uniune în momentul curent, pentru a evita astfel de greşeli. Operaţiile care se pot face cu structuri se pot face şi cu uniuni;

o structura poate conţine uniuni şi o uniune poate conţine structuri.

Exemplul următor arată cum se poate lucra cu numere de diferite tipuri şi lungimi, reunite într-un tip uniune:

```
typedef struct numar {

char tipn; //tip numar (caracter: i-int, l-long, f-float, d-double)
union num{int ival;long lval;float fval;double dval;} val;//valoare }
Numar;    // definire tip de date structură Numar

void write (Numar n) { // in dependența de tip afiseaza valoare numar

switch (n.tipn) {

case 'i': printf ("%d \n", n.val.ival); break;

case 'l': printf ("%ld \n", n.val.lval); break;

case 'f': printf ("%f \n", n.val.fval); break;

case 'd': printf ("%.15lf\n", n.val.dval); }

return;

}
```

Observaţie: În locul tipului uniune se poate folosi o variabilă de tip void* care va conţine adresa unui număr, indiferent de tipul lui. Memoria pentru număr se va aloca dinamic:

```
typedef struct numar {
char tipn; //tip numar (caracter: i-int, l-long, f-float, d-double)
void *pv; }Numar
void write(Numar n) { // in dependența de tip afiseaza valoarea
switch (n.tipn) {
case 'i': printf ("%d \n", *(int*)n.pv); break;
case 'l': printf ("%ld \n", *(long*)n.pv); break;
case 'f': printf ("%f \n", *(float*)n.pv); break;
case 'd': printf ("%.15lf\n", *(double*)n.pv);
}
}
```

Exemplu: Dacă într-un program se doreşte accesarea valorii octeţilor care compun un întreg, se poate proceda în felul următor:

```
#include <stdio.h>
typedef union intreg { int val;
                       char octeti [2];
                     } Intreg;
int main ()
{   Intreg n;
    printf ("Dati valoarea intregului: "); scanf ("%d", &n.val);
printf ("Octetii sunt: oct1 = %d oct2 = %d\n",n.octeti[0],n.octeti [1]);
return 0;
}
```

Exemplu: Reprezentarea internă a unui caracter

```
#include <stdio.h>
struct biti
 {unsigned b0:1;
  unsigned b1:1;
  unsigned b2:1;
  unsigned b3:1;
  unsigned b4:1;
  unsigned b5:1;
  unsigned b6:1;
  unsigned b7:1;
 };
typedef union caracter
  {biti b;
   char c;
  } Caracter;
```

```
int main() {
Caracter octet;
printf("dati un caracter:”);
scanf(“%c”, &octet.c);
printf("reprezentarea interna:\n“);
printf(“%d”,octet.b.b7); printf(“%d”,octet.b.b6);
printf(“%d”,octet.b.b5); printf(“%d”,octet.b.b4);
printf(“%d”,octet.b.b3); printf(“%d”,octet.b.b2);
printf(“%d”,octet.b.b1); printf(“%d”,octet.b.b0);
return 0;
}
```

5. Enumerări

Tipul de date enumerare permite de a asocia niște nume (constante simbolice) unor date care iau valori constante întregi. Utilizarea acestui tip de date face programul scris mai clar și lizibil.

Tipul enumerare declară constante simbolice, cărora li se asociază valori numerice de tip întreg. Compilatorul asociază constantelor enumerate câte o valoare întreagă din succesiune începând cu 0. Implicit, şirul valorilor e crescător cu pasul 1. Un nume de constantă nu poate fi folosit în mai multe enumerări.

Enumerări

Sintaxa definiției tipului enumerare este următoarea:

```
enum nume_enumerare {listă_constante } listă_variabile;
```

unde:
nume_enumerare – este numele enumerării;
listă_constante – lista identificatorilor ce formează enumerarea;
lista_variabile – lista de variabile de tip enumerare.

Putem folosi instrucțiunea **typedef** pentru a introduce sinonimul al unui tip enumerare:
Exemplu:

```
enum culori {alb, albastru, galben, verde, rosu, negru};
enum culori cul = verde;
Sau
typedef enum culori
{alb, albastru, galben, verde, rosu, negru} Culori;
Culori cul = verde;
// implicit alb este asociat cu 0, albastru cu 1,
// . . . , negru cu 5
```

Exemple de utilizare

```
cul = albastru;
if (cul = = alb) printf ("culoarea este alb \n");
```

Instrucţiunea:

```
printf ("%d %d %d \n",alb,galben,negru);
```

afişează 0 2 5

La declararea tipului enumerare se poate specifica explicit valoarea unuia sau mai multor identificatori din listă folosind o iniţializare:

```
typedef enum culori
{alb, albastru=5, galben, verde=10, rosu, negru}
Culori;
```

Exemplu:

```
#include<stdio.h>
typedef enum operatii
{adunare=1, scadere , inmultire} Operatii;
int main ( ) {
int a, b , rez;
Operatii op;
printf("Dati 2 valori si codul operatiei: \n");
scanf("%d%d%d", &a, &b, &op);
switch (op) {
case adunare: rez = a + b; break;
case scadere: rez = a - b; break;
case inmultire: rez = a * b; break;}
printf ("%d\n", rez);
return 0;}
```

Probleme propuse spre rezolvare folosind tipuri de date definite de utilizator

1. Să se defineasca o structura “time" care grupează 3 întregi ce reprezintă ora, minutul şi secunda pentru un moment de timp. Să se scrie funcţii pentru: verificare corectitudine ora, citire moment de timp, scriere moment de timp, comparare de structuri “time”. Să se scrie un program pentru citirea şi ordonarea cronologică a unor momente de timp şi afişarea listei ordonate, folosind funcţiile anterioare.

2. Să se scrie un program care declară o strucură pentru un număr generic, folosind o uniune pentru valoarea numărului şi un câmp discriminant de tipul char ce vă spune ce fel de număr este (întreg – i, întreg lung – l, real – f, real dubla precizie – d). Să se scrie o funcţie ce citeşte o valoare numărului de tipul respectiv. Să se scrie o funcţie ce afişează valoarea unui număr de tip respectiv.

3. Să se refacă programul din p. 2 utilizînd un câmp discriminant de tipul enumerare cu numele simbolice I, L, F. D pentru tipul de număr generic.

4. Să se scrie o aplicaţie în C pentru evidenţa studenţilor de la facultate. Despre un student se cunosc următoarele informaţii: numele, anul și nota medie Programul trebuie să permită, printr-un meniu interactiv, următoarele operaţii (prin funcţii cu transfer de parametrii – parametrii obligatorii ai funcţiilor sunt adresa tabloului de studenţi şi numărul

curent de studenţi): introducerea informații despre studenți de la tastatură, afişarea informației despre

studenţi pe ecran, căutarea unui student după nume, sortarea studenţilor după nume.

Tutoriale online pentru tema prelegerii:

1. https://ocw.cs.pub.ro/courses/programare/laboratoare/lab11
2. https://igotopia.ro/cum-grupezi-variabilele-in-structuri-c/
3. https://docs.google.com/document/d/1Z5GKvAlQC1KqR4F6uCocBUceOkn11mqCoXdPT4pY2wo/edit
4. https://www.pbinfo.ro/articole/7652/structuri-de-date-neomogene-in-cpp
5. http://andrei.clubcisco.ro/cursuri/1pc/curs/1/Curs%209%20Doc.pdf
6. http://info.tm.edu.ro:8080/~junea/cls%2010/structuri/
7. https://info64.ro/Structuri_de_date-Uniuni/
8. http://www.aut.upt.ro/~rraul/PC/2009-2010/PC_lab9.pdf
9. https://muhaz.org/tipuri-de-date-definite-de-utilizator.html
10. http://www.cs.ucv.ro/staff/gmarian/Programare/cap8_Structuri.pdf

VĂ MULȚUMESC PENTRU ATENȚIE!

MULTĂ SĂNĂTATE ȘI SUCCESE!