Cursul 11

Tipuri de date definite de utilizator

1. Redenumiri de tipuri

Limbajul C/C++ dă posibilitatea programatorului să introducă, cu ajutorul cuvântului cheie ***typedef***, denumiri personalizate pentru tipurile predefinite ale limbajului. In exemplul următor

```
#include<iostream>
using namespace std;
typedef int Intreg;
int main(){
	Intreg a=-5;
	cout<<a<<endl;
	return 0;
}
```

identificatorul ***Intreg*** este un sinonim pentru ***int***.

Redenumirea unui anumit tip este simplă: scriem declaraţia fără iniţializare a unei date de tipul vizat şi apoi transformăm această declaraţie de dată într-o *declaraţie de tip* scriind cuvântul cheie ***typedef*** în faţa sa, în acest fel numele datei devine numele tipului redenumit. Astfel, instrucţiunea

```
	int Intreg;
```

declară variabila ***Intreg*** de tip ***int***, iar instrucţiunea

```
	typedef int Intreg;
```

declară tipul ***Intreg*** ca o redenumire a tipului ***int***.

Prin convenţie, denumirile de tipuri se scriu cu majusculă pe primul loc, sau numai cu majuscule, sau numai cu litere mici dar cu sufixul ***_t*** :

```
	typedef unsigned Natural, UINT, size_t;
```

Redenumirile sunt deosebit de utile în cazul tipurilor compuse:

```
#include<iostream>
using namespace std;
typedef int Tablou[5], Intreg;
typedef Tablou Matrice[2];
int main(){
	Tablou tab={1,2,3,4,5};
	for(Intreg i=0;i<5;i++) cout<<tab[i]<<endl;
	Matrice mat={{10,20,30,40,50}, {11,12,13,14,15}};
	for(Intreg i=0;i<2;i++){
		for(int j=0;j<5;j++) cout<<mat[i][j]<<' ';
		cout<<endl;
	}
	return 0;
}
```

O altă utilizare a redenumirilor constă în asigurarea unui anumit grad de indepedenţă faţă de impementarea curentă a tipurilor fundamentale. De exemplu, funcţiile API ale sistemului de operare Windows folosesc următoarele tipuri:

```
typedef unsigned int          UINT;
typedef char                  CHAR;
typedef char *                PCHAR;
typedef unsigned char         UCHAR;
typedef int                   INT;
typedef int *                 PINT;
typedef unsigned int          UINT;
typedef unsigned short        USHORT;
typedef unsigned long         DWORD;
typedef long                  LONG;
typedef long *                PLONG;
typedef unsigned long         ULONG;
```

Tipul ***INT*** este sinonim cu ***int*** care acum desemnează un întreg pe 32 de biţi, dar dacă în viitor ***int*** va desemna un întreg pe 64 de biţi, să spunem, atunci se va putea păstra semnificaţia lui ***INT*** actuală adaptând numai declaraţia sa ***typedef***, fără alte schimbări în definiţia funcţiilor şi datelor care folosesc tipul ***INT***.

Subliniem că o declaraţie ***typedef*** nu defineşte un tip nou de dată, ea introduce numai o nouă denumire pentru un tip deja existent. Programatorul are totuşi posibilitatea să definească şi tipuri noi de date, dar numai dacă acestea se încadrează în una din următoarele patru categorii: *enumerări*, *structuri*, *uniuni* sau *clase*. Aceste patru tipuri sunt numite *tipuri utilizator*, programatorul fiind considerat un utilizator al limbajului. Dintre acestea, noi vom studia în continuare numai primele trei, clasele necesitând un studiu separat, ele fiind esenţa extinderii la C++ a limbajului C.

2. Enumerări

O *enumerare* este un tip de dată definit de programator prin care un şir de constante întregi, bine precizate, capată un tip comun, introdus prin cuvântul cheie ***enum***. De exemplu, declaraţia

```
enum Logic {indecis, fals, adev};
```

defineşte un nou tip de dată, tipul ***Logic*****,** fiecare variabilă de acest tip având numai una dintre cele trei *valori*: `indecis`, `fals` sau `adev`. O variabilă de tip ***Logic*** se declară, la fel ca orice alt tip de variabilă:

```
Logic omega;
```

şi poate fi initializată astfel:

```
omega=fals;
```

Avem şi urmatoarea variantă de declaraţie:

```
enum Logic {indecis, fals, adev} omega;
```

şi chiar şi varianta cu iniţializare:

```
enum Logic {indecis, fals, adev} omega=fals;
```

Enumerările sunt utilizate pentru a da mai multă claritate programului. Ele sunt de fapt nişte codificari numerice. In exemplul precedent, cele trei valori, `indecis`, `fals` şi `adev`, sunt tratate de compilator drept identificatori de constante întregi cu valorile implicite

în ordine: `indecis=0`, `fals=1` şi `adev=2`. Programatorul poate impune alte valori, dar numai în momentul declaraţiei, în modul urmator:

```
enum Fuzzy{imposibil,neverosimil,incert=5,plauzibil,sigur=10};
```

Acum avem `imposibil=0`, `neverosimil=1`, `incert=5`, `plauzibil=6`, `sigur=10`. Valorile care nu sunt impuse de programator se obţin prin incrementarea celor precedente cu câte o unitate. Prima valoare este în mod implicit zero. Programatorul trebuie să se asigure că valorile obţinute sunt distincte, altfel utilitatea codificării este îndoielnică.

In expresii aritmetice orice dată de tip enumerare este tratată ca un număr întreg, conversia către ***int*** fiind implicită, dar conversia unui întreg către un tip enumerat trebuie cerută explicit.

Exemplul 1. Definiţii, declaraţii, conversii implicite, conversii explicite:

```
#include<iostream>
using namespace std;

enum Zile{luni,marti,miercuri,joi,vineri,sambata,duminica};
enum Calificativ{minuscul=1,mititel,micut,mic=10,mare,maret};

int main(void){
	Zile azi;
	Calificativ cumEste;
	int x;
	azi=luni;
	cout<<azi<<" "<<duminica<<endl;		// 0 6
	cout<<micut<<" "<<maret<<endl;		// 3 12

	/*Conversie int <- enum; OK, IMPLICIT: */
	x=cumEste=mare;
	cout<<x<<endl;				// 11
	x=duminica+maret;
	cout<<x<<endl;				// 18

	/*Conversie enum <- int; CAST EXPLICIT: */
	/*		specific C++		*/
	//azi=2;		//error: cannot convert
			//from 'const int' to 'enum Zile'
			//Conversion  requires an explicit cast
	azi=(Zile)2;
	if( azi==miercuri) cout<<"DA"<<endl;	// DA
	azi=(Zile)(33*x+1);
	cout<<azi<<endl;			// 595
	//luni=(Zile)4;	//error: left operand must be l-value (!)

	/*Conversie enum <- enum; CAST EXPLICIT: */

	//azi=minuscul;	//error: cannot convert from
			//'enum Calificativ' to 'enum Zile'
	azi=(Zile)minuscul;
	if (azi==marti) cout<<"DA"<<endl;	// DA
	if (miercuri==mititel) cout<<"DA"<<endl;	// DA (!)
	return 0;
}
```

Exemplul 2. Utilizare:

```
#include<iostream>
using namespace std;

enum Zile {luni,marti,miercuri,joi,vineri,sambata,duminica};
char denum[7][100]={"Luni","Marti","Miercuri","Joi",
"Vineri","Sambata","Duminica"};

int timpDeLucru(Zile zi){
	switch(zi)
	{
	case luni:
	case marti:
		return 6;
	case miercuri:
	case joi:
		return 12;
	case vineri:
		return 3;
	default:
		return 0;
	}
}
void scrieTexte(Zile zi){
	int t;
	t=timpDeLucru(zi);
	cout<<"Azi este "<<denum[zi]<<" si muncesc "<<t<<" ore, ";
	if (t>1) cout<<"prea mult!"<<endl;
	else cout<<"asa da!"<<endl;

	zi=(Zile)((zi+1)%7);
	t=timpDeLucru(zi);
	cout<<"Maine este "<<denum[zi]<<" si muncesc "<<t<<" ore,";
	if (t>1) cout<<"prea mult!"<<endl;
	else cout<<"asa da!"<<endl;

	return;
}

int main(void)
{

	scrieTexte(vineri);
	return 0;
}
/*
Azi este Vineri si muncesc  3 ore, prea mult!
Maine este Sambata si muncesc  0 ore, asa da!
Press any key to continue
*/
```

3. Structuri

In limbajul C, o *structură* este un ansamblu format din una sau mai multe variabile grupate împreună sub un singur nume. Datele de tip structură au pătruns în limbajele de programare în primul rând pentru facilitarea manipulării datelor de gestiune economică. Să analizăm un tabel de inventar în care pe fiecare linie sunt următoarele informaţii despre articolul inventariat: număr de inventar, denumire articol, firma producatoare, unitatea de masura, cantitate, pret unitar.

Observăm că pe fiecare coloană avem date anonime de acelaşi tip care pot fi indexate prin numerotare de sus în jos, astfel că ele ar putea fi memorate în mod natural într-un tablou C/C++. Memorarea unui tabel de inventar pe coloane, ca o colecţie de tablouri, este utilizabilă practic numai dacă tabelul are un număr mic de linii şi, mai ales, dacă numărul de linii este fix, operaţiile de adăugare, de inserare sau de ştergere de linii din tabel fiind foarte greoaie. Este clar că invetarierea presupune mai ales astfel de operaţii, corespunzătoare introducerii sau scoaterii din invetar a unor articole.

Pentru facilitarea operaţiilor cu liniile unui astfel de tabel i s-a permis programatorului să îşi definească tipul necesar de dată care să grupeze la un loc toate informaţiile unei singure linii. O astfel de dată este numită structură (*structure*) în C/C++ şi Basic, înregistrare (*record*) în Pascal sau, în general, *articol* în lucrul cu fişiere. Este clar că tiparul acestei date eterogene, cu câmpuri membre de diverse tipuri, este dat de linia capetelor de coloană. Având la dispoziţie variabile de acest tip, care permit de exemplu atribuiri de linii în totalitate, actualizarea tabelelor este mult mai simplă:

```
#include<iostream>
#include<iomanip>
using namespace std;
const int dim=100;

struct ArticolDeInventar{
      int nrInventar;
      char denumire[100];
      char firmaProd[100];
      char unitateDeMasura[100];
      int cantitate;
      double pretUnitar;
};

ArticolDeInventar tab[dim]={
      {112233,"Hartie de scris A4","Xerox","top",132,9.25},
      {126565,"Hartie copiator color", "Mondi", "top",25,14.50}};

int main(void){
      ArticolDeInventar ultimul={132435,"Pix cu gel",
                                 "Noki", "bucata", 43, 2.30};
      int n;
      for(n=0;n<dim;n++)
            if(tab[n].nrInventar==0) break;
      if(n==dim) cout<<"Trebuie inceput un nou tabel"<<endl;
      else tab[n]=ultimul;
      cout.setf(ios::left, ios::adjustfield);
```

```
	for(int i=0; i<=n; i++){
		ArticolDeInventar art=tab[i];
		cout<<setw(3)<<i+1;
		cout<<setw(7)<<art.nrInventar;
		cout<<setw(25)<<art.denumire;
		cout<<setw(8)<<art.firmaProd;
		cout<<setw(8)<<art.unitateDeMasura;
		cout<<setw(7)<<art.cantitate;
		cout<<setw(9)<<art.pretUnitar;
		cout<<endl;
	}
	return 0;
}
/*
1  112233 Hartie de scris A4       Xerox   top     132     9.25
2  126565 Hartie copiator color    Mondi   top     25      14.5
3  132435 Pix cu gel               Noki    bucata  43      2.3
Press any key to continue . . .
*/
```

Structurile sunt utilizate şi în alte scopuri, nu numai la tabelarea datelor. De exemplu, sistemul de operare Windows foloseşte pentru comunicarea între aplicaţii mesaje implementate ca structuri de forma:

```
typedef struct tagMSG {
  HWND   hwnd;
  UINT   message;
  WPARAM wParam;
  LPARAM lParam;
  DWORD  time;
  POINT  pt;
} MSG, *PMSG, *LPMSG;
```

unde câmpurile membre au următoarele semnificaţii: ***hwnd*** este *handler*-ul (identificatorul) aplicaţiei căreia îi este destinat mesajul, ***message*** este *id*-ul (identificatorul) mesajului, ***wParam*** şi ***lParam*** sunt parametri care conţin informaţii suplimentare în funcţie de *id*-ul mesajului, ***time*** reprezintă momentul postării mesajului şi ***pt*** poziţia cursorului în coordonatele ecranului la momentul postării. Toate tipurile implicate sunt numerice întregi, cu excepţia lui ***POINT*** care este o structură:

```
typedef struct tagPOINT {
  LONG x;
  LONG y;
} POINT, *PPOINT;
```

Pentru utilizarea judicioasă a memoriei structurile pot avea, pe lângă câmpurile obişnuite, şi aşa numitele *câmpuri pe biţi*: variabile de tip întreg reprezentate pe un număr specificat de biţi. În acest fel putem reprezenta mai eficient valorile mici, care nu necesită toţi biţii alocaţi unui tip întreg predefinit.

Un câmp pe biţi se declară ca un câmp obişnuit de tip întreg urmat de simbolul : şi apoi de numărul de biţi specificat printr-o constantă. Exemplu:

```
struct Date {  // bit fields
      unsigned short nWeekDay : 3;         // 0..7   (3 bits)
      unsigned short nMonthDay : 6;        // 0..31  (6 bits)
      unsigned short nMonth : 5;           // 0..12  (5 bits)
      unsigned short nYear : 8;            // 0..100 (8 bits)
};
```

Utilizarea strucurilor s-a extins considerabil odată cu dotarea lor, pe lângă membrii de tip variabilă (*câmpuri*), cu membri de tip funcţie (*metode*). Această extindere a făcut trecerea de la C la C++ şi, pentru a marca schimbarea majoră de viziune asupra structurilor, ele au fost redenumite *clase*. In C++ nu sunt diferenţe esenţiale între structuri şi clase. Aici vom prezenta structurile aşa cum au fost ele implementate iniţial în limbajul C.

Structurile se deosebesc de tablouri prin următoarele aspecte: (*i*) elementele membre ale unei structuri pot avea tipuri diferite, (*ii*) structurile se comportă la alocare exact ca variabilele simple: dacă sunt locale sunt alocate pe stivă, altfel sunt alocate în zona variabilelor globale, structurile pot fi atribuite, funcţiile pot returna structuri, (*iii*) elementele unui structuri nu sunt variabile anonime ci au câte un nume, (*iv*) referirea unui element al unei structuri se realizează cu operatorul de selecţie (.) şi nu cu operatorul de indexare ([ ]).

Structurile sunt definite cu ajutorul cuvantului cheie ***struct***, iar o declaraţie de structură introduce un nou tip de dată: cel tocmai precizat. Exemplu, declaraţia

```
struct Complex{
      double x;
      double y;
};
```

defineşte tipul ***Complex***, fiecare dată de tip ***Complex*** este un *obiect* de tip structură compus din două câmpuri membre, ***x*** şi ***y***, ambele tip ***double***. O variabilă de acest tip se declară, în C, astfel:

```
struct Complex w;
```

In C++ nu mai este obligatorie utilizarea cuvântului cheie ***struct*** la declararea unei variabile (dacă structura a fost deja definită), deci declaraţia de mai sus poate fi scrisă astfel:

```
Complex w;
```

Accesul la câmpurile membre este dat de operatorul de selecţie "punct":

```
cout<<w.x<<endl;
```

Iniţializarea unei variabile de tip structură poate fi facută odată cu declararea ei, respectând strict ordinea membrilor din definiţia structurii:

```
Complex w={1.5, 3.0};
```

De asemenea, variabilele pot fi declarate odată cu definiţia structurii, astfel:

```
struct Complex{
      double x;
      double y;
} z1,z2, z3={1.5, 3.3};
```

Pot fi declarate şi structuri anonime, caz în care toate variabilele de acest tip trebuie declarate de la bun început:

```
struct {
      double x;
      double y;
} z1,z2, z3={1.5, 3.3};
```

In sfârşit, se poate redenumi orice tip structură cu ***typedef***. In exemplul următor redenumim cu identificatorul ***Complex*** structura ***tagComplex***:

```
typedef struct tagComplex{
      double x;
      double y;
} Complex;
```

şi în acest caz orice compilator de C acceptă declaraţia de variabilă

```
Complex w;
```

fără să mai fie prefixată cu ***struct***, exact ca în C++. Din motive de sintaxă, în C numele unei structuri este format din cuvântul cheie ***struct*** urmat de *tag*-ul (eticheta) structurii – identificatorul scris în faţa acoladelor la declarare –, sau din cuvântul cheie ***struct*** urmat direct de descrierea structurii între acolade. In C++ *tag*-ul a devenit chiar numele tipului declarat.

Membrii unei structuri pot fi variabile de orice tip, pot fi chiar structuri deja definite, sau pot fi pointeri către tipul structură care tocmai se defineşte. Nu sunt admise definiţiile recursive. Aşa cum am mai precizat, în C membrii unei structuri nu pot fi de tip funcţie (nu există variabile de tip funcţie), acest lucru este însă admis în C++, unde structurile sunt cazuri particulare de clase.

Structurile pot fi declarate atât global, în afara corpului oricărei funcţii, cât şi local, in interiorul corpului unei funcţii sau al unui bloc de instrucţiuni, caz în care sfera de vizibilitate este restrânsă corespunzător locului declaraţiei.

Să analizăm următoarea implementare a numerelor complexe în stil C clasic:

```
#include<iostream>
#include<math.h>
using namespace std;
struct Complex{
            double x;
            double y;
};
Complex initAlg(double x, double y){
      Complex z={x,y};
      return z;
}
Complex initTrig(double ro, double fi){
      return initAlg(ro*cos(fi),ro*sin(fi));
}
void scrie(Complex z){
    if (z.y == 0) cout << z.x;
    else if (z.x == 0 )  cout << z.y << "i";
    else if ( z.y>0)  cout << z.x << "+" << z.y << "i";
    else cout << z.x << z.y << "i";
    return;
}

double modul2(Complex z){
      return z.x*z.x+z.y*z.y;
}

double modul(Complex z){
      return sqrt(modul2(z));
}
```

```
double arg(Complex z){
	return atan2(z.y,z.x);
}
Complex sum(Complex u, Complex v){
	return initAlg(u.x+v.x,u.y+v.y);
}
Complex prod(Complex u, Complex v){
	return initAlg(u.x*v.x-u.y*v.y, u.y*v.x+u.x*v.y);
}
Complex conj(Complex u){
	return initAlg(u.x,-u.y);
}
Complex realToComplex(double x){
	return initAlg(x,0);
}

Complex cat(Complex u, Complex v){
	Complex ro=realToComplex(1.0/modul2(v));
	return prod(ro, prod(u,conj(v)));
}

const double  PI = 3.141592653589;
int main(void){

	Complex z1,z2,z3;
	z1=initAlg(-1,1);
	cout<<"z1=";
	scrie(z1);
	cout<<endl;
	z2=initTrig(1,PI/3);
	cout<<"z2=";
	scrie(z2);
	cout<<endl;
	z3=prod(cat(z1,z2),z2);
	cout<<"z3=";
	scrie(z3);
	cout<<endl;
	return 0;
}

/*
z1=-1+1i
z2=0.5+0.866025i
z3=-1+1i
Press any key to continue . . .
*/
```

In funcţia ***main*** declaraţia

```
Complex z1,z2,z3;
```

provocă rezervarea pe stivă a trei locaţii de tip ***Complex***, mărimea unei astfel de locaţii depinde de compilator şi poate fi aflată cu ***sizeof***:

```
cout<<sizeof(Complex)<<endl;//16
```

Dacă nu a fost iniţializată la declarare, o structură poate fi iniţializată mai târziu sau pe componente sau printr-o atribuire de structuri:

```
Complex z1, z2, z3={1,2};
//z2={10,20};//syntax error : '{'
z2.x=10;
z2.y=20;
z1=z2;
```

Pentru a fi utilizată corect trebuie iniţializate toate componentele unei structuri:

```
Complex w;
w.x=10;
scrie(w);
//10-9.25596e+061i
```

Atribuirea între structuri este *de tip valoare*: se execută prin copierea valorilor câmpurilor membre, indiferent dacă acestea sunt de tip aritmetic sau pointer. Subliniem că şi în cazul în care membrul unei structuri este un tablou, la atribuire se copie toate elemente tablolului:

```
struct Tablouri{
	int tab[3];
	Complex ctab[3];
};

int main(void){
	Tablouri s, t={{10,20,30},{{1,2},{3,4},{5,6}}};
	s=t;
	t.tab[0]=1000;
	t.ctab[0].x=1111;
	cout<<s.tab[0]<<endl;//10
	cout<<s.ctab[0].x<<endl;//1
	return 0;
}
```

O altă posibilitate de atribuire între obiecte, întâlnită în alte limbaje, ar fi fost ca atribuirea să fie *de tip referinţă*, adresa obiectului din dreapta să fie cea atribuită celui din stânga, şi nu valoarea sa. Acest mod de atribuire între obiecte s-ar justifica din motive de economie de memorie şi de creştere a vitezei de execuţie. Nici în C şi nici în C++ nu există o astfel de atribuire; în aceste limbaje, pentru a economisi spaţiul pe stivă, obiectele mari se alocă dinamic în heap prin intermediul pointerilor, după cum vom vedea mai târziu. Atribuirile de pointeri din C/C++ au devenit atribuiri prin referinţă în unele limbaje care provin din C, cum ar fi Java şi C#.

Revenind la exemplul de implementare a numerelor complexe, să analizăm acum instrucţiunea

```
z1=initAlg(-1,1);
```

In timpul apelului `initAlg(-1,1)` este creată şi iniţializată pe stivă variabila temporală ***z*** a cărei valoare este returnată de funcţie la încetarea apelului. Valoarea returnată (formată din două valori de tip ***double***) este atribuită prin copiere lui ***z*1**.

Subliniem că, în conformitate cu declaraţia sa, funcţia ***initAlg*** returnează rezultatul prin valoare, nu returnează obiectul ***z***, ci numai valoarea sa, mai precis valorile celor două câmpuri membre, prin copiere. Strategia de transmitere este următoarea: la încheierea apelului este creat un obiect temporar care conţine rezultatul, în acest caz o copie a lui ***z***, obiect care este menţinut în viaţă la dispoziţia funcţiei apelante până la terminarea evaluării întregii expresii care conţine apelul. In mod normal acest obiect temporal este atribuit (prin copiere) unei variabile, lui ***z*1** în exemplul dat, după care el este lăsat să dispară. Atenţie: el poate fi şi alterat din greşeală:

```
	Complex z1,z2,z3;
	z2=initAlg(100,200);
	z1=(initAlg(-1,1)=z2);
	cout<<"z1=";
	scrie(z1);//z1=100+200i
```

In general, în timpul evaluării expresiilor care conţin obiecte de tip structură sau clasă sunt create obiecte temporare care conţin "rezultate intermediare" disponibile până la terminarea evaluării întregii expresii. Modificarea sau luarea adresei unui astfel de obiect constituie întotdeauna o eroare de programare.

Structurile se transmit în mod implicit între funcţii exact ca variabilele simple: prin valoare (altfel spus: prin copiere):

```
Complex test(Complex z){
	z.x=100;
	z.y=200;
	return z;
}
int main(void){
	Complex z1={1,2},z2;
	z2=test(z1);
	cout<<"z1=";
	scrie(z1);
	cout<<endl;
	cout<<"z2=";
	scrie(z2);
	cout<<endl;
	return 0;
}
/*
z1=1+2i
z2=100+200i
Press any key to continue . . .*/
```

La apelul `z2=test(z1)` pe stivă în domeniul lui test este creată variabila locală ***z*** în care este copiată valoarea lui ***z*1**, ***z*** fiind cea modificată şi nu ***z*1**, după cum se vede.

4. Uniuni

O dată de tip *uniune* este o structură "colapsată": toţi membrii ei sunt suprapuşi în acelaşi spaţiu de memorie. O uniune se declară exact la fel ca o structură, schimbând doar cuvântul cheie ***struct*** cu ***union*** dar, spre deosebire de cazul unei structuri, unei uniuni nu i se alocă decât spaţiul de memorie strict necesar pentru a cuprinde cel mai expansiv membru, toţi membrii uniunii urmând sa fie alocaţi, rând pe rând, în această zonă comună. In consecinţă, membrii unei uniuni nu pot fi folosiţi simultan ci numai succesiv: ultimul alocat este utilizat în mod valid până la alocarea altuia. Programul trebuie să ştie permanent care membru al uniunii are reprezentarea corectă în memorie în acel moment.

Uniunile sunt utilizate pentru economisirea memoriei alocate programului. Să studiem pe un exemplu:

```
#include<iostream>
using namespace std;

union Gramada{
      char ch;
      int in;
      double db;
};
int main(void){
      cout.precision(12);
      cout<<"sizeof char    = "<<sizeof(char)<<endl;
      cout<<"sizeof int     = "<<sizeof(int)<<endl;
      cout<<"sizeof double  = "<<sizeof(double)<<endl;
      cout<<"sizeof Gramada = "<<sizeof(Gramada)<<endl;

      Gramada alfa;

      cout<<"\n PE RAND: "<<endl;
      alfa.ch='A';
      cout<<"ch="<<alfa.ch<<endl;
      alfa.in=11111;
      cout<<"in="<<alfa.in<<endl;
      alfa.db=2.22222222;
      cout<<"db="<<alfa.db<<endl;

      cout<<"\n ULTIMUL INCARCAT: db "<<endl;
      cout<<"ch="<<alfa.ch<<endl;
      cout<<"in="<<alfa.in<<endl;
      cout<<"db="<<alfa.db<<endl;

      alfa.db=2.22222222;
      alfa.in=11111;
      alfa.ch='A';
      cout<<"\n ULTIMUL INCARCAT: ch "<<endl;
      cout<<"ch="<<alfa.ch<<endl;
      cout<<"in="<<alfa.in<<endl;
      cout<<"db="<<alfa.db<<endl;

      return 0;
}
/* REZULTAT:
sizeof char    = 1
sizeof int     = 4
sizeof double  = 8
sizeof Gramada = 8

 PE RAND:
ch=A
in=11111
db=2.22222222

 ULTIMUL INCARCAT: db
ch=Æ
in=1903870354
db=2.22222222

 ULTIMUL INCARCAT: ch
```

```
ch=A
in=11073
db=2.22222137452
Press any key to continue . . .*/
```

Se observă că de fiecare dată este citită în mod corect numai valoarea câmpului încărcat ultimul.

Să prezentăm şi un exemplu de utilizare din lumea reală: compilatorul C++ Builder al firmei Embarcadero Technologies utilizează pentru mesajele sistemului de operare structura ***TMessage*** definită astfel:

```
typedef unsigned int Cardinal;
typedef unsigned short Word;
struct TMessage {
	Cardinal Msg;
	union {
		struct {
			Word WParamLo;
			Word WParamHi;
			Word LParamLo;
			Word LParamHi;
			Word ResultLo;
			Word ResultHi;
		};
		struct {
			int WParam;
			int LParam;
			int Result;
		};
	};
};
```

Un obiect ***TMessage*** este format din *id*-ul mesajului, câmpul ***Msg***, şi două structuri suprapuse într-o uniune. Funcţia care citeşte mesajul decide din *id*-ul acestuia care dintre cele două structuri este cea validă, şi interpretează datele în consecinţă.