POO – C++ - Laborator 3

Cuprins

Namespace

O problemă care ar putea să apară în programele C, este legată de faptul că, pe măsură ce dimensiunea programului creşte, este din ce în ce mai greu să se evite duplicarea numelor pentru funcţii şi variabile. C++-ul standard oferă un mecanism pentru evitatea acestor coliziuni de nume, şi anume prin utilizarea namespace-urilor.

Se vor grupa declaraţii / definiţii la nivelul unor namespace-uri, şi în acest caz, numele unor variabile sau funcţii se poate repeta dar la nivelul unor spaţii de nume diferite. Datorită faptului că ele vor fi plasate în namespace-uri (spaţii de nume) diferite nu vor exista coliziuni.

Librăriile limbajului C++ standard sunt plasate în spaţiul de nume **std (standard).** De aceea în momentul în care vor fi folosite aceste librării C++ standard se va utiliza directiva **using**.

Exemplu1.cpp

```
#include <conio.h>
#include <iostream>
using namespace std;

namespace C1_1
{
	int x = 10;
	float y = .5;
}

namespace C1_2
{
	double x = 2.5;
	char y = 'x';
}

int main()
{
```

```
	cout << C1_1::x << "\t";
	cout << C1_2::x << endl;

	using C1_1::y; //using declaration
	cout << y << "\t";
	cout <<C1_2::y << endl;

	using namespace C1_1; //using directive
	cout << x << "\t";
	cout << y << endl;

	_getch();
	return 0;
}
```

Rulare:

```
10   2.5
0.5  x
10   0.5
```

Un namespace poate fi continuat în mai multe fişiere header, nu se consideră o redefinire a spaţiului de nume, ci doar o continuare a celui deja definit:

Header1.h

```
#ifndef _HEADER1_H
#define _HEADER1_H

namespace lib
{
	extern int x;
	void f();
	//......
}
#endif
```

Header2.h

```
#ifndef _HEADER2_H
#define _HEADER2_H

#include "Header1.h"

namespace lib
{
	extern float y;
	void g();
	//......
}
#endif
```

Exemplu2.cpp

```
#include <conio.h>
#include <iostream>
using namespace std;

#include "Header1.h"
#include "Header2.h"

using namespace lib;
```

```
int lib::x = 5;
float lib::y;

void lib::f()
{
	cout << "f():" << x << endl;
}

void lib::g()
{
	cout << "g():" << y << endl;
}
int main()
{
	int x = 10;
	cout << x << endl;
	f();
	g();

	_getch();
	return 0;
}
```

Rulare:

```
10
f():5
g():0
```

Spre deosebire de o **directivă using**, care tratează numele introduse ca fiind globale scopului, o **declaraţie using**, este o declaraţie care se face în interiorul scopului curent. Practic, prin utilizarea **declaraţiei using** se poate suprascrie un nume introdus prin intermediul unei **directive using**.

Header.h

```
#ifndef _HEADER_H
#define _HEADER_H

namespace Functii1
{
	void f();
	void g();
}

namespace Functii2
{
	void f(int x);
	void f();
	void g();
}

#endif
```

Functii.cpp

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

#include "header.h"
```

```
void Functii1::f()
{
	cout<<"Functii1: f()" << endl;
}


void Functii1::g()
{
	cout<<"Functii1: g()" << endl;
}


void Functii2::f(int x)
{
	cout<<"Functii2: f(" << x << ")" << endl;
}

void Functii2::f()
{
	cout<<"Functii2: f()" << endl;
}


void Functii2::g()
{
	cout<<"Functii2: g()" << endl;
}
```

Exemplu3.cpp

```
#include <conio.h>
#include "header.h"


void h()
{
	using namespace Functii1; //directiva using
	using Functii2::f; //declaratia using
	f(5);
	f();
	Functii1::f();
}
int main()
{
	h();
	_getch();
	return 0;
}
```

Rulare:

```
Functii2: f(5)
Functii2: f()
Functii1: f()
```

Notă: La declaraţia using: *using Functii2::f;* s-a folosit numai numele identificatorului (funcţia **f**) fără nici o informaţie despre tipul argumentelor funcţiei.
Dacă namespace-ul conţine un set de funcţii supraîncărcate cu acelaşi nume, prin declaraţia using se introduc toate funcţiile care se află în acest set.

Se va încerca să se evite cazurile de ambiguitate care pot să apară în aceste situaţii.

Mecanisme de transfer a parametrilor

Referinţa este un alias pentru o anume variabilă. Dacă în C, transmiterea parametrilor unei funcţii se face prin valoare (inclusiv şi pentru pointeri), în C++ se adaugă şi transmiterea parametrilor prin referinţă.
Dacă tipul pointer se introduce prin construcţia: **tip ***, tipul referinţă se introduce prin **tip &**.

O variabilă referinţă trebuie să fie iniţializată la definirea sau declararea ei cu numele unei alte varibile.

```
int i;
int &j = i;
```

Variabila **j** este un nume alternativ pentru **i,** cu această referinţă se poate accesa întregul păstrat în zona de memorie alocată lui **i.**

În limbajul C++, parametri pot fi transferaţi în două moduri: prin valoarea (valoare directă sau adresă) şi prin referinţă. În transferul prin valoare parametri actuali (specificaţi în momentul apelului) sunt copiaţi în zona de memorie rezervată pentru parametri formali (specificaţi în momentul definiţiei funcţiei). Orice modificare efectuată asupra parametrilor formali nu va implica şi modificarea parametrilor actuali (de apel).

Modificarea parametrilor actuali poate fi realizată dacă în momentul apelului se transmite adresa de memorie a acestora. Astfel secvenţele de instrucţiuni ale funcţie pot modifica conţinutul memoriei de la adresele transmise şi implicit valorile parametrilor actuali (de apel).

În cazul transferului prin referinţă, funcţiei i se transmit nu valorile parametrilor actuali ci un alias al acestora. În acest fel secvenţa de instrucţiuni a funcţiei poate modifica valorile parametrilor actuali.

Domeniul de valabilitate şi vizibilitate

Prin domeniu de valabilitate (vizibilitate) se înţelege zona de program în care este valabilă declararea unui identificator (variabilă, funcţie). Astfel, toţi identificatorii declaraţi într-un bloc (secvenţă de program delimitată de acolade) sau modul (fişier sursă) sunt cunoscuţi blocului/modulului respectiv şi se numesc variabile locale sau globale. Dacă în interiorul unui bloc se defineşte un alt bloc atunci variabilele locale ale blocului părinte devin variabile globale pentru blocul fiu iar variabilele locale blocului fiu nu au valabilitate în blocul părinte. Dacă în blocul fiu este declarat (redefinit) un identificator identic ca denumire cu unul din blocul părinte atunci se ia în considerare ultima declaraţie. Se poate face apel la declaraţia din blocul părinte prin utilizarea operatorului ::.

Funcţii cu parametri impliciţi

Este posibil să se apeleze o funcţie cu un număr de parametri actuali mai mic decât numărul parametrilor formali. Parametrii care pot lipsi se numesc impliciţi (cu valori implicite); ei trebuie să se regasească în extrema dreaptă a listei parametrilor formali din antetul funcţiei.

```
1. void f(int m=0, int n=1){}
2. f();      //f(0,1)
3. f(1);     //f(1,1)
4. f(1,2);

5. void g(float x, int m=0, int n=1){}
6. g(2.5);   //g(2.5, 0,1)
7. g(2.5, 1);       //g(2.5, 1,1)
8. g(2.5, 1,2);
```

Dacă se lucrează cu funcţii ce au parametri impliciţi, trebuie să se evite situaţiile de ambiguitate ce pot apare la supraîncărcarea funcţiilor.

```
void F1(void)
{
      cout << "Apel F1\n";
}
void F2(double re=0, double im=0)
{
      cout << "Apel F2\n";
}
```

Supraîncărcarea funcţiilor

În C++ pot exista mai multe funcţii cu acelaşi nume, dar cu liste diferite de argumente. Aceste funcţii sunt **supraîncărcate**. Supraîncărcarea se utilizează pentru a da acelaşi nume tuturor funcţiilor care fac acelaşi tip de operaţie.

Numele unei funcţii şi ansamblul argumentelor sale, ca număr şi tipuri, se numeşte ***semnătura*** acelei funcţii. Se observă că funcţiile supraîncărcate au semnături diferite.
Să luăm ca exemplu funcţia care calculează valoarea absolută a unui număr care poate fi int, long sau double. În C_ANSI, pentru această operaţie există trei funcţii:

```
int abs (int); // se calculeaza valoarea absoluta a unui intreg
long labs (long); // se calculeaza valoarea absoluta a unui intreg lung
double fabs (double); // se calculeaza valoarea absoluta a unui numar real
```

În C++ se foloseşte numele "abs" pentru toate cele trei cazuri, declarate astfel:

```
int abs (int);  // functie de biblioteca
long abs (long); // functie de biblioteca
double abs (double);  // functie utilizator
```

```
#include <iostream>
using namespace std;
//#include <stdlib.h>
```

```
#include <conio.h>

double abs(double x)
{
	return x < 0 ? -x : x;
}

int main()
{
	int a = -5;
	long b = -5L;
	float f = -3.3f;
	double d = -5.5;
	cout << "a=" << abs(a) << endl;
	cout << "b=" << abs(b) << endl;
	cout << "c=" << abs(f) << endl;
	cout << "c=" << abs(d) << endl;
	_getch();
	return 0;
}
```

Compilatorul C++ selectează funcţia corectă prin compararea tipurilor argumentelor din apel cu cele din declaraţie.

Când facem supraîncărcarea funcţiilor, trebuie să avem grijă ca numărul şi/sau tipul argumentelor versiunilor supraîncărcate să fie diferite. Nu se pot face supraîncărcări dacă listele de argumente sunt identice:

```
int calcul (int);
double calcul (int);
```

O astfel de supraîncărcare (numai cu valoarea returnată diferită) este ambiguă. Compilatorul nu are posibilitatea să discearnă care variantă este corectă şi semnalează eroare.

Tema de laborator:

-revizuirea aplicatiei din laboratorul 1:
Crearea unui catalog pentru mai multe grupe de studenti, permitand citirea componentelor, afisarea, sortarea conform unor criterii (alfabetic, crescator dupa lungimea numelui sau descrescator dupa nota), avand in vedere alocarea dinamica de memorie si eliberarea acesteia cand nu mai este necesara.

//exemplu de continut Catalog.h:

```
#pragma once //pentru a nu fi inclus de mai multe ori
/*
echivalent cu cele 3 linii de directive pentru preprocesare:
#ifndef _CATALOG_
#define _CATALOG_
//macrodefinitiile, declaratiile de tipuri noi de date si de functii
#endif
```

```
*/

//putem seta aici toate incluziunile, astfel incat in fisierul "main" si cel de implementare a functiilor sa avem de scris o singura
//linie de forma #include "Catalog.h"
#include <iostream>
#include <stdlib.h> //sau se poate folosi cstdlib
#include <string.h>

using namespace std;

typedef struct _Student{
	private: //aceste campuri NU vor fi "vizibile" (accesibile) direct din afara structurii
		char *nume;
		int nota;
	public:
		int getNota(void); //functie accesor prin care se permite "vederea"notei luate de studentul curent
		void setNota(int v);//functie mutator prin care se seteaza valoarea notei pentru studentul curent
		char* getNume(void);//functie accesor prin care
			//se permite accesul la pointerul la nume (deja alocat!) pentru studentul curent
		void setNume(char unNume[]);//functie mutator prin care se aloca dinamic Z.M. si se va copia continutul sirului
			//obs.: atentie la numararea caracterelor! Trebuie inclus SI terminatorul de sir '\0'
		void elibMem(void); //eliberam Z.M. ocupata de nume si setam pe NULL pointerul
		void citire(void); //citim, apelam setarile...
		void afisare(void); //afisarea datelor - efectuata cu ajutorul functiilor accesor
}Student;

typedef int (*PFnComparare)(Student a, Student b);

typedef struct _Grupa{
	//DUPA realizarea sarcinilor de lucru din cadrul laboratorului se pot
	//"ascunde" campurile de tip date astfel incat accesul catre ele sa fie realizate prin metode accesor/mutator
	int nrStud;
	Student *tabStudenti; //alocat dinamic in cadrul functiei de citire. eliberat de catre functia de eliberare
	char* numeGrupa;	//denumirea grupei: 1208B, 1207A etc.
	void citire(void);	//DECLARATIA metodei de citire => PROTOTIPUL functiei
	void afisare(void);	//DECLARATIA metodei de afisare => PROTOTIPUL functiei
	PFnComparare comparator; //pointer catre o functie de comparatie, definita global
		//acest pointer va fi setat de catre un "obiect" de tip structura Catalog, pentru fiecare grupa in parte.
	void bSort(void); //nu mai e necesar pointerul la functia de comparatie ca parametru,
		//deoarece este camp al "obiectului" curent de tip grupa!
```

```
	void elibMem(void); //a se vedea implementarea!
} Grupa;

typedef struct _Catalog{
	int nrGrupe;
	Grupa *tabGrupe;
	void setComparator(PFnComparare comparator);//iteram fiecare grupa din tablou si facem initializarea campului
		//cu parametrul formal primit
	void citire(void);//citim numarul de grupe, alocam dinamic tabloul, apoi apelam citirea pentru fiecare grupa in parte
	void afisare(void);//afisam, pe rand, continutul fiecarei grupe. A se vedea functia de afisare din structura Grupa!
	void sortare(void); //considerand acel comparator care a fost setat pentru fiecare grupa
		//se apeleaza bSort pentru fiecare grupa in parte
	void elibMemorie(void);//in care apelam elibMem pentru fiecare grupa in parte
} Catalog;

//declaratii de functii "globale":
int comparNumeAlfabetic(Student a, Student b);
int comparNoteDescrescator(Student a, Student b);
int comparNumeDupaLungimeCrescator(Student a, Student b);

//exemplu de continut fisier Catalog.cpp
//va trebui sa completati cu implementarile (definitiile) tuturor functiilor declarate si pe care va trebui sa le utilizati
#include "Catalog.h"
	//exemplu de DEFINIRE a unei metode
void Catalog::setComparator(PFnComparare comparator)
{
	int i; //contor pentru a parcurge grupele
	for(i=0;i<nrGrupe;i++)
		tabGrupe[i].comparator=comparator;
		//setam pentru fiecare grupa in parte pointerul la functia de comparatie
}
void Grupa::elibMem(void)
{
	int s; //contor pentru parcurgerea studentilor din cadrul unei grupe
	if(numeGrupa)
	{
		cout<<"eliberarea memoriei pentru grupa "<<numeGrupa<<endl;
		free(numeGrupa);
		numeGrupa=NULL;
	}
	for(s=0;s<nrStud;s++)  //nrStud este CAMP al structurii grupa
			tabStudenti[s].elibMem(); //eliberam Z.M. ocupata de numele fiecarui student in parte
	free(tabStudenti);
```

```
	tabStudenti=NULL;
}

int comparNumeDupaLungimeCrescator(Student a, Student b)
{
	int rez=strlen(a.nume)-strlen(b.nume);
	if(rez>0)
		rez=1;
	else
		if(rez<0)
			rez=-1;
	//daca e 0 ramane 0
	return rez;
}

//exemplu de continut L02Main.cpp
#include "Catalog.h"
int main(void)
{
	int operatie;
	PFnComparare unPointerLaOFunctieDeComparare;
	Catalog catalogulAnului1, catalogulAnului2, catalogulAnului3, catalogulAnului4;
	Catalog catalogMaster[2];

	catalogulAnului2.citire();
	catalogulAnului2.afisare();
	do{
		do{
			cout<< "Ce doriti sa efectuati?"<<endl;
			cout<< "0.Iesire din program;"<<endl;
			cout<< "1 - sortare alfabetica a numelor;"<<endl;
			cout<< "2 - sortare descrescatoare dupa nota;"<<endl;
			cout<< "3 - sortare dupa lungimea numelui - crescator."<<endl;
			cin>>operatie;
		}while((operatie<0) || (operatie>3));
		switch(operatie)
		{
		case 1:
			cout<<"1 - sortare alfabetica a numelor;"<<endl;
			unPointerLaOFunctieDeComparare=comparNumeAlfabetic;
			//se putea face apelul si direct, dars-a dorit evidentierea atribuirii pentru o
variabila
			//de tip pointer la functie!
			catalogulAnului2.setComparator(unPointerLaOFunctieDeComparare);
			//se putea apela si direct
catalogulAnului2.setComparator(comparNumeAlfabetic);
			break;
		case 2:
```

```
                cout<< "2 - sortare descrescatoare dupa nota;"<<endl;
                unPointerLaOFunctieDeComparare=comparNoteDescrescator;
                catalogulAnului2.setComparator(unPointerLaOFunctieDeComparare);
                break;
            case 3:
                cout<< "3 - sortare dupa lungimea numelui - crescator."<<endl;
                unPointerLaOFunctieDeComparare=comparNumeDupaLungimeCrescator;
                catalogulAnului2.setComparator(unPointerLaOFunctieDeComparare);
                break;
            default:
                cout<< "Sfarsitul executiei programului."<<endl;
            }
        }while(operatie);
        catalogulAnului2.elibMemorie();
        //folositi si cataloagele celorlalti ani de studiu sau stergeti declaratiile variabilelor nefolosite in
program!
        return EXIT_SUCCESS;
}
//de completat apelurile necesare in main!
//de completat declaratiile de functii
```

-Cerinte (scenariu):

*.Completarea programului cu implementarile functiilor declarate, conform logicii programului!
*.pentru implementarea functiei bSort (bubbleSort) - preluati logica implementarii de la laboratorul 1,
faceti modificarea a.i. sa folositi comparatorul dat ca pointer
si interschimbarea - realizati-o cu ajutorul unei functii "swap2" care are ca parametri doua referinte la structuri Student
*.citirea numarului de grupe =>alocare
*.citirea numarului de studenti pentru fiecare grupa in parte =>alocare tablou
*.citirea datelor pentru fiecare student in parte
*.afisarea datelor citite pana acum (pentru verificarea corectitudinii citirilor efectuate)
*.citirea operatiei dorite:
0 - iesire din program => apelul eliberarii Zonelor de Memorie care au fost alocate dinamic
1 - sortare alfabetica a numelor (setare pointer la functia de comparatie corespunzatoare,)
2 - sortare descrescatoare dupa nota (apoi apelul pentru fiecare grupa in parte a)
3 - sortare dupa lungimea numelui - crescator (sortarii bubbleSort, avand criteriul de comparatie setat)