Lab 2 – P2

Cuprins

1. Namespace

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

Se pot grupa declarații / definiții la nivelul unor namespace-uri și în acest caz, numele unor variabile sau funcții se pot repeta dar la nivelul unor namespace-uri diferite. Datorită faptului că ele vor fi plasate în namespace-uri (spații de nume) diferite nu vor exista coliziuni.

Librăriile limbajului C++ standard sunt plasate în spațiul 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 <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;
	return 0;
}
```

Rulare:

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

Un namespace poate fi continuat în mai multe fișiere header, nu se consideră o redefinire a spațiului 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 <iostream>

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

using namespace std;
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();
	return 0;
}
```

Rulare:

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

Spre deosebire de o **directivă using**, care tratează numele introduse ca fiind globale, o **declarație using** este o declarație care se face în interiorul domeniului de valabilitate curent. Practic, prin utilizarea **declarației 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>
#include "Header.h"

using namespace std;

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();
	return 0;
}
```

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

Notă: La declarația using: `using Functii2::f;` s-a folosit numai numele identificatorului (funcția **f**) fără nici o informație despre tipul argumentelor funcției.
Dacă namespace-ul conține un set de funcții supraîncărcate cu același nume, prin declarația `using` se introduc toate funcțiile care se află în acest set.

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

2. Mecanisme de transfer al parametrilor

Referința este un alias pentru o anume variabilă. Dacă în C, transmiterea parametrilor unei funcții se face prin valoare (inclusiv și pentru pointeri), în C++ se adaugă și transmiterea parametrilor prin referință.
Dacă tipul pointer se introduce prin construcția: **tip ***, tipul referință se introduce prin **tip &**.

O variabilă referință trebuie să fie inițializată 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 referință se poate accesa întregul păstrat în zona de memorie alocată lui **i.**

Exemplu3.cpp

```
#include <iostream>

using namespace std;

int main()
{
	int x = 5;
	int &y = x;

	y = 10;
	cout << x << endl;

	x = 15;
	cout << y << endl;
	return 0;
}
```

```
Rulare:
10
15
```

În limbajul C++, parametri pot fi transferați în două moduri: prin valoare (valoare directă sau adresă) și prin referință. În cazul transferului prin valoare parametrii actuali (specificați în momentul apelului) sunt copiați în zona de memorie rezervată pentru parametri formali (specificați în momentul definiției funcției). Orice modificare efectuată asupra parametrilor formali nu va implica și modificarea parametrilor actuali (de apel).

Modificarea parametrilor actuali poate fi realizată dacă în momentul apelului se transmite adresa de memorie a acestora. Astfel secvențele de instrucțiuni ale funcției pot modifica conținutul memoriei de la adresele transmise și implicit valorile parametrilor actuali (de apel).

În cazul transferului prin referință, funcției i se transmit nu valorile parametrilor actuali ci un alias al acestora. În acest fel secvența de instrucțiuni a funcției poate modifica valorile parametrilor actuali.

Exemplu4.cpp

```
#include <iostream>

using namespace std;

void f(int x) //transfer prin valoare directă
{
	x = 5;
}

void g(int *x) //transfer prin adresă
{
	*x = 10;
}

void h(int &x) //transfer prin referință
{
	x = 15;
}

int main()
{
	int x = 0;

	cout << x << endl;

	f(x);
	cout << x << endl;

	g(&x);
	cout << x << endl;

	h(x);
	cout << x << endl;
	return 0;
}
```

Rulare:

```
0
0
10
15
```

3. Domeniul de valabilitate și vizibilitate

Prin domeniu de valabilitate (vizibilitate) se înțelege zona de program în care este valabilă declararea unui identificator (variabilă, funcție). Astfel, toți identificatorii declarați într-un bloc (secvență de program delimitată de acolade) sau modul (fișier sursă) sunt cunoscuți blocului/modulului respectiv și se numesc variabile locale sau globale. Dacă în interiorul unui bloc se definește 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 declarație.

Se poate face apel la variabilele globale prin utilizarea operatorului ::

Exemplu5.cpp

```
#include <iostream>

using namespace std;

int a = 10;

int main()
{
	int a = 100;
	cout << a << endl;
	cout << ::a << endl;
	return 0;
}
```

Rulare:

```
100
10
```

4. Funcții cu parametri impliciți

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

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

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

Dacă se lucrează cu funcții ce au parametri impliciți, trebuie să se evite situațiile de ambiguitate ce pot să apară la supraîncărcarea funcțiilor.

```
void F1(void)
{
	cout << "Apel F1\n";
}

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

5. Supraîncărcarea funcțiilor

În C++ pot exista mai multe funcții cu același nume, dar cu liste diferite de argumente. Aceste funcții sunt **supraîncărcate**. Supraîncărcarea se utilizează pentru a da același nume tuturor funcțiilor care fac același tip de operație.

Numele unei funcții și ansamblul argumentelor sale, ca număr și tipuri, se numește ***semnătura*** acelei funcții. Se observă că funcțiile supraîncărcate au semnături diferite.

Să luăm ca exemplu funcția care calculează valoarea absolută a unui număr care poate fi int, long sau double. În C_ANSI, pentru această operație există trei funcții:

```
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 folosește numele `abs` pentru toate cele trei cazuri, declarate astfel (obs. Există mai multe supraîncărcări ale funcției `abs` în C++ în afara celor de mai jos):

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

#include <iostream>
using namespace std;

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;
	return 0;
}
```

Compilatorul C++ selectează funcția corectă prin compararea tipurilor argumentelor din apel cu cele din declarație.

Când facem supraîncărcarea funcțiilor, trebuie să avem grijă ca numărul și/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 și semnalează eroare.

6. Exerciţii

1. Supraincarcati o functie de afisare care să aibă următoarele prototipuri. Dati exemple de utilizare pentru fiecare funcție scrisă:

```
void Afisare(int a);
void Afisare(double a);
void Afisare(int a[], int n);
void Afisare(Data a);

struct Data
{
	int zi, luna, an;
}
```

2. Definiti spatiul de nume AC. In cadrul lui definiti:

- structura Student, cu membrii nota si anul de studiu
- metodele asociate unei structuri student: citeste, afiseaza
- structura Grupa cu membrii: numeleGrupei, nrStudenti, tabloulDeStudenti (alocat dinamic!);
- metodele asociate ueni grupe: afiseaza, citeste, dealoca, sorteaza(cu parametru predefinit alfabetic cu valoarea 1), notaMaxima, mediaGrupei, sorteaza