Laborator 2:Pointeri catre obiecte. Pointerul this. Functii friend. Supraincarcarea functiilor si a operatorilor.

Pointeri catre obiecte

Pointerii catre obiecte sunt similari celor catre oricare alt tip de variabila. Accesul membrilor unei clase (date si metode) cu ajutorul unui pointer catre un obiect se face utilizand operatorul **->** in locul operatorului punct.

Exemplu:

```
# include <iostream>
using namespace std;
class MyClass
{
	int a;
public:
	int b;
	void  seta(int i)
	{
		a=i;
	}
	int geta()
	{
		return a;
	}
};

int main()
{
	MyClass x, *p;
	p=&x; // p primeste adresa lui x
	p->seta(3);
	p->b=5;
	cout<<p->geta()<<" "<<p->b<<endl; /* folosim op -> pentru a apela seta
si geta*/
	system("PAUSE");
	return EXIT_SUCCESS;
}
```

Pointerul this.

In momentul in care este apelata o functie membru a unei clase, i se paseaza automat un pointer implicit care este un pointer catre obiectul care a generat apelarea (obiectul care a invocat functia). Acest pointer este numit *pointerul this.*

```
# include <iostream>
using namespace std;
class MyClass
{
	int a;
public:
	int MyFunc()
	{
		int s;
		this->a=9;
		s=this->a+7;
		return s;
	}
};

int main()
{
	MyClass x;
	cout<<x.MyFunc()<<endl;
	system("PAUSE");
	return EXIT_SUCCESS;
}
```

Functii friend

Accesul la membrii unei clase poate fi acordat si altor functii care nu sunt functii membru ale clasei respective. Astfel de functii se numesc functii prietene clasei respective (functii **friend**) si sunt declarate astfel cu ajutorul specificatorului **friend**. Functiile **friend** pot fi functii membru ale altor clase sau pot fi functii independente. O functie friend are acces la membrii private si protected ai clasei careia ii este prietena. Pentru a declara o functie friend, includeti prototipul ei in acea clasa, precedat de cuvantul cheie **friend.**

!! Deoarece nu sunt membre ale clasei functiile friend nu se apeleaza prin intermediul unui obiect si nu au pointer this.

Exemplu:

```
# include <iostream>
using namespace std;
class MyClass
{
	int a,b;
public:
	void  seta(int i, int j)
	{
		a=i;
		b=j;
	}
	friend int sum (MyClass x);
};
int sum(MyClass x)
{
	return x.a+x.b;
} /*din sum avem acces la membrii privati a si b deoarece sum este friend*/
int main()
```

```
{
	MyClass x;
	x.seta(3,4);
	cout<<sum(x)<<endl;
	system("PAUSE");
	return EXIT_SUCCESS;
}
```

Supraincarcarea functiilor.

Un mod prin care se realizeaza polimorfismul in C++ este *supraincarcarea (overloading)* functiilor. Aceasta inseamna ca doua sau mai multe functii pot avea acelasi nume atata timp cat declaratiile parametrilor sunt diferite: fiecare redefinire a functiei trebuie sa foloseasca sau tipuri diferite sau numar diferit de parametrii. Nu pot fi supraincarcate doua functii care difera doar prin tipul returnat!

Exemplu:

Functia “aduna” este supraincarcata pentru a putea fi realizata adunarea dintre un numar complex si un double, un double si un numar complex si doua numere complexe.

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

class complex
{
	double re;
	double im;
public:
	void setre(double x)
	{
		re=x;
	}
	void setim(double y)
	{
		im=y;
	}
	double getre()
	{
		return re;
	}
	double getim()
	{
		return im;
	}
};
void read(complex& a)
{
	double x,y;
	cout<<"Introduceti partea reala: ";
	cin>>x;
	a.setre(x);
	cout<<"Introduceti partea imaginara: ";
```

```
	cin>>y;
	a.setim(y);
}
void write(complex a)
{
	cout<<a.getre()<<"+i*"<<a.getim()<<endl;
}
complex aduna(complex a, double b)
{
	complex c;
	c.setim(a.getim());
	c.setre(a.getre()+b);
	return c;
}
complex aduna(double a, complex b)
{
	complex c;
	c.setim(b.getim());
	c.setre(b.getre()+a);
	return c;
}
complex aduna(complex a, complex b)
{
	complex c;
	c.setim(b.getim()+a.getim());
	c.setre(b.getre()+a.getre());
	return c;
}

int main()
{
	complex a,c,d;
	double x=3.5;
	read(a);
	cout<<"Rezultatele adunarii:"<<endl;
	//adunare complex-double
	c=aduna(a,x);
	write(c);
	//adunare double-complex
	c=aduna(x,a);
	write(c);
	//adunare complex-complex
	read(c);
	d=aduna(a,c);
	write(d);
	system("PAUSE");
	return EXIT_SUCCESS;
}
```

Cazuri de ambiguitate

- Conversia automata a tipului in C++. Compilatorul incearca sa converteasca automat argumentele care sunt folosite pentru a apela o functie in tipurile argumentelor asteptate de functie.

Exemplu

```
# include<iostream>
using namespace std;
void func(double a)
{
	cout<<"functia primeste un double ca parametru: "<<a<<endl;
}
void func(float a)
{
	cout<<"funcia primeste un float ca parametru: "<<a<<endl;
}
int main()
{
	double d;
	cin>>d;
	float f;
	cin>>f;
	func(d);//nu este ambigua deoarce parametru a fost decalarat explicit
ca fiind double
	func(f);//nu este ambigua deoarce parametru a fost decalarat explicit
ca fiind float
	func(8.5);//nu este ambigua deoarce daca nu sunt
	//specificate ca find float, toate constantele in virgula mobila sunt
tratate //ca double
	func(5); //ambigua deoarece compilatorul nu stie daca sa converteasca 5
la //double sau float
	system("PAUSE");
	return EXIT_SUCCESS;

}
```

- Folosirea parametrilor cu valori implicite (default) pentru functii supraincarcate

Exemplu:

```
# include<iostream>
using namespace std;
void func(double a)
{
	cout<<"un singur parametru"<<endl;
}
void func(double a, double b=3.4)
{
	cout<<"doi parametrii"<<endl;
}
int main()
{
	func(7.5,8); // nu este ambigua deoarece doar cea de-a doua varianta a
lui //func poate fi apelata
	func(7.8); //ambigua deoarece ambele variante a lui func pot fi apelate
(in //cel de-al doilea caz, al doilea parametru este implicit)

	system("PAUSE");
	return EXIT_SUCCESS;
```

```
}
```

Supraincarcarea operatorilor

Limbajul C++ permite programatorului sa defineasca diverse operatii cu obiecte ale claselor, utilizand simbolurile operatorilor standard.
Termenul **supraincarcarea operatorilor** este întâlnit în literatura de specialitate si sub denumirea de redefinirea (*overloading*) sau supradefinirea operatorilor. Principalele avantaje ale redefinirii operatorilor sunt claritatea si usurinta cu care se exprima anumite operatii specifice unei clase. Solutia alternativa ar fi definirea unor functii si apelul functiilor în cadrul unor expresii.
Limbajul C++ nu permite crearea de noi operatori, în schimb permite redefinirea majoritatii operatorilor existenti astfel încât atunci când acesti operatori sunt aplicati obiectelor unor clase sa aiba semnificatia corespunzatoare noilor tipuri de date (adica, ale claselor respective).
Operatorii se supraincarca prin crearea functiilor **operator.** Functiile operator pot fi sau nu membri ai clasei in care vor opera. Functiile operator care nu sunt de tip membru sunt de obicei functii **friend** ai clasei in care vor opera. Felul in care sunt scrise functiile operator difera pentru cele de tip membru de cele de tip **friend.**

Exemplu

Utilizarea functiilor operator ca membri ai clasei. Folosind o functie membru pentru a supraincarca un operator, obiectul din stanga operatorului este cel care genereaza apelarea functiei operator supraincarcate. Apoi prin intermediul pointerului this este transmis un pointer catre acest obiect.

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

class complex
{
	double re;
	double im;
public:
	void setre(double x)
	{
		re=x;
	}
	void setim(double y)
	{
		im=y;
	}
	double getre()
	{
		return re;
	}
	double getim()
	{
		return im;
	}
	complex operator +(double b);
	complex operator +(complex b);
	complex operator++();
};
void read(complex& a)
```

```
{
	double x,y;
	cout<<"Introduceti partea reala: ";
	cin>>x;
	a.setre(x);
	cout<<"Introduceti partea imaginara: ";
	cin>>y;
	a.setim(y);
}
void write(complex a)

{
	cout<<a.getre()<<"+i*"<<a.getim()<<endl;
}
complex complex::operator +(complex b)
{
	complex c;
	c.re=re+b.re;
	c.im=im+b.im;
	return c;
}
complex  complex::operator +(double b)
{
	complex c;
	c.re=re+b;
	c.im=im;
	return c;
}
complex complex::operator ++()
{
	re++;
	return *this; //returneaza obiectul care a generat apelarea
}

int main()
{
	complex a,c,d;
	double x=3.5;
	read(a);
	//incrementare
	cout<<"increment:"<<endl;
	++a;
	write(a);
	cout<<"Rezultatele adunarii:"<<endl;
	//adunare complex-double
	c=a+x;
	write(c);

	//adunare complex-complex
	read(c);
	d=a+c;
	write(d);

	system("PAUSE");
	return EXIT_SUCCESS;
}
```

Utilizarea operatorilor ca functii friend. In exemplul anterior, nu putem folosi functii operator membre pentru a define adunarea dintre un double si un complex . De ce?

Solutia este sa folosim functii friend in care sunt pasate explicit ambele argumente.

Exemplu:

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

class complex
{
	double re;
	double im;
public:
	void setre(double x)
	{
		re=x;
	}
	void setim(double y)
	{
		im=y;
	}
	double getre()
	{
		return re;
	}
	double getim()
	{
		return im;
	}
	friend complex operator+ (double a, complex b);
};
void read(complex& a)
{
	double x,y;
	cout<<"Introduceti partea reala: ";
	cin>>x;
	a.setre(x);
	cout<<"Introduceti partea imaginara: ";
	cin>>y;
	a.setim(y);}
void write(complex a)

{
	cout<<a.getre()<<"+i*"<<a.getim()<<endl;
}

complex operator+(double a, complex b)
{
	complex c;
	c.re=a+b.re;
	c.im=b.im;
	return c;
}
```

```
int main()
{
	complex a,c;
	double x=3.5;
	read(a);
	//adunare double-complex
	c=x+a;
	write(c);

	system("PAUSE");
	return EXIT_SUCCESS;
}
```

Exercitii

1. Pornind de la codul de mai jos, sa se supraincarce functia `scade` pentru a fi posibila executia codului din functia `main`.

```
#include <string>
#include <iostream>
#include <math.h>

using namespace std;

class complex
{
	double re;
	double im;
public :
	void SetRe(double value)
	{
		re = value;
	};
	void SetIm(double value)
	{
		im = value;
	};
	double GetRe()
	{
		return re;
	};
	double GetIm()
	{
		return im;
	};

};

complex scade(complex a, complex b)
{
```

```
	complex c;
	c.SetRe(a.GetRe() - b.GetRe());
	c.SetIm(a.GetIm() - b.GetIm());
	return c;
};


int main()
{

	complex a, b, c;
	a.SetRe(4);
	a.SetIm(5);
	b.SetRe(2);
	b.SetIm(7);
	c = scade(a, b);
	double d = 4;
	c = scade(a, d);
	cout<<c.GetRe()<<" "<<c.GetIm()<<endl;

	c = scade(-d, a);
	cout<<c.GetRe()<<" "<<c.GetIm()<<endl;

	system("PAUSE");
	return EXIT_SUCCESS;
}
```

2. Pornind de la codul de mai jos, scrieti o definitie pentru a supraincarca operatorul `=` astfel incat sa fie posibila executia codului din functia `main`.

```
#include <string>
#include <iostream>
#include <math.h>

using namespace std;

class complex
{
    double re;
    double im;
public :
    void SetRe(double value)
    {
        re = value;
    };
    void SetIm(double value)
    {
        im = value;
    };
    double GetRe()
    {
        return re;
    };
    double GetIm()
    {
```

```
            return im;
        };


    };


    int main()
    {
        complex a;
        double b = 20;
        a = b;
        cout<<a.GetRe()<<" "<<a.GetIm()<<endl;


        system("PAUSE");
        return EXIT_SUCCESS;
    }
```

3. Pornind de la codul de mai sus, adaugati metode ale clasei ce permit executia codului din cadrul functiei `main` mai jos ilustrate.

```
    int main()
{
        complex a,b;
        a.SetRe(20);
        a.SetIm(10);
        b = - a;
        cout<<a.GetRe()<<" "<<a.GetIm()<<endl;
        cout<<b.GetRe()<<" "<<b.GetIm()<<endl;



        system("PAUSE");
        return EXIT_SUCCESS;
}
```

4. Pornind de la codul de mai jos, supraincarcati operatorul – pentru a fi posibila executia codului din cadrul functiei `main`.

```
#include <string>
#include <iostream>
#include <math.h>

using namespace std;

class complex
{
      double re;
      double im;
public :
      void SetRe(double value)
      {
            re = value;
```

```
        };
        void SetIm(double value)
        {
                im = value;
        };
        double GetRe()
        {
                return re;
        };
        double GetIm()
        {
                return im;
        };
        complex operator-(complex a);
        complex operator-(double b);

};

int main()
{

        complex a, b, c;
        a.SetRe(4);
        a.SetIm(5);
        b.SetRe(2);
        b.SetIm(7);
        double d = 4;
        c = a - d;
        cout<<c.GetRe()<<" "<<c.GetIm()<<endl;

        c = b - a;
        cout<<c.GetRe()<<" "<<c.GetIm()<<endl;

        system("PAUSE");
        return EXIT_SUCCESS;
}
```

5. Pornind de la codul de mai jos, scrieti o definitie pentru functia `abs` ca fiind functie friend a clasei `complex` pentru a calcula modulul unui numar complex(|a+b*i| = radical(a^2 + b^2)).

```
#include <string>
#include <iostream>
#include <math.h>

using namespace std;

class complex
{
        double re;
        double im;
public :
        void SetRe(double value)
        {
                re = value;
        };
```

```
	void SetIm(double value)
	{
		im = value;
	};
	double GetRe()
	{
		return re;
	};
	double GetIm()
	{
		return im;
	};
};

int main()
{

	complex a;
	a.SetRe(5);
	a.SetIm(10);
	cout<<abs(a)<<endl;
	system("PAUSE");
	return EXIT_SUCCESS;
}
```

6. Pornind de la codul de mai jos, scrieti o definitie pentru supraincarcarea operatorului – pentru a fi posibila executia codului ilustrat in `main` (indicatie, supraincarcati operatorul folosind conceptul de functie friend)

```
#include <string>
#include <iostream>
#include <math.h>

using namespace std;

class complex
{
	double re;
	double im;
public :
	void SetRe(double value)
	{
		re = value;
	};
	void SetIm(double value)
	{
		im = value;
	};
	double GetRe()
	{
		return re;
	};
	double GetIm()
	{
		return im;
	};
```

```
};

int main()
{
	complex a, b, c;
	a.SetRe(4);
	a.SetIm(5);
	b.SetRe(2);
	b.SetIm(7);
	double d = 6;

	c = d - a;
	cout<<c.GetRe()<<" "<<c.GetIm()<<endl;

	c = b - a;
	cout<<c.GetRe()<<" "<<c.GetIm()<<endl;

	system("PAUSE");
	return EXIT_SUCCESS;
}
```