Moştenire

Noțiuni generale

Moştenirea este un mecanism prin care se pot clase noi (numite derivate) prin extinderea unor clase existente (numite clase de bază). Clasa derivată este, de obicei, o specializare a clasei de bază; între cele două trebuie să existe o relație de tip *este un/o*. Exemple: Student *este o* Persoană, Vector *este o* Colecție, Autoturism *este un* Vehicul …

Sintaxa utilizată pentru crearea unei clase derivate este

class ClasaDerivata : tip_derivare ClasaDeBaza {……};.

Clasa derivată va moşteni funcțiile şi valorile membre de la clasa de bază. Tipul de derivare determină vizibilitatea membrilor moşteniți în clasa derivată. Efectele tipului de derivare sunt sintetizate în tabelul următor:

Tip derivare → ↓ Vizibilitate inițială	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	inaccesibil	inaccesibil	inaccesibil

Restricțiile impuse membrilor moşteniți prin derivarea de tip protected şi private pot fi relaxate prin mecanismul de publicizare. Acesta implică citarea în secțiunea publică a clasei derivate a membrilor care se doresc a fi publicizați.

Este permisă convertirea unei instanțe a clasei derivate într-o instanță a clasei de bază. În felul acesta se pot apela metodele publice ale clasei de bază indiferent de tipul de derivare utilizat.

Exemple:

```
// Clasa de baza
class Baza
{
public:
	void Metoda1() {}
	void Metoda2() {}
	int atribut1;
protected:
	void Metoda3() {}
	int atribut2;
private:
	void Metoda4() {}
	int atribut3;
};

// Clasa derivata public (contine toate
// metodele si atributele clasei Baza)
class Derivata1 : public Baza
{
public:
	void Metoda5()
	{
		// poate accesa metodele publice
		// si protected ale clasei de baza
		Metoda1();
```

```
                Metoda3();

                // dar nu are acces la membrii
                // privati ai acesteia
                // Metoda4();  => eroare la compilare
                // atribut3++; => eroare la compilare
        }
};

// Clasa derivata protected
class Derivata2 : protected Baza
{
public:
        void Metoda5()
        {
                // poate accesa metodele publice
                // si protected ale clasei de baza
                Metoda1();
                Metoda3();

                // dar nu are acces la membrii
                // privati ai acesteia
                // Metoda3();  => eroare la compilare
                // atribut3++; => eroare la compilare
        }

        // publicizare a metodei
        using Baza::Metoda2;
};

void main()
{
        Derivata1 d1;

        // sunt accesibile toti membrii
        // publici ai clasei de baza si
        // ai clasei derivate
        d1.atribut1++;
        d1.Metoda1(); // OK: metoda mostenita public din clasa de baza
        d1.Metoda2(); // OK: metoda mostenita public din clasa de baza
        // d1.Metoda3(); => eroare la compilare (accesare functie protected)
        d1.Metoda5(); // OK: metoda publica in clasa derivata

        Derivata2 d2;

        // sunt accesibili doar membrii publici ai clasei derivate
        // d2.atribut1++; => eroare: atributul devine protected in clasa derivata
        // d2.Metoda1(); => eroare: metoda devine protected in clasa derivata
        d1.Metoda2(); // OK: metoda este publicizata in clasa derivata
        d2.Metoda5(); // OK: metoda publica in clasa derivata

        // convertire la clasa de baza
        Baza b = (Baza)d2;
        b.Metoda1();   // acum este permisa accesarea metodei
                       // publice 1 din clasa de baza

}
```

Inițializarea

Inițializarea instanțelor claselor care compun o ierarhie se face începând de la clasa de bază si continuă în ordine cu clasele derivate. În cazul în care clasa de bază nu are un constructor implicit, clasa derivată este obligată să inițializeze clasa de bază folosind constructorul acesteia şi lista de inițializare.

Exemplu:

```
// Clasa de baza Persoana: Contine datele referitoare la o persoana.
class Persoana
{
public:
```

```
        // constructor
        Persoana(int cod, char* nume)
        {
                // copiem datele primite ca parametri
                _cod = cod;
                strcpy(_nume, nume);
        }

        // functii de acces
        int GetCod()          const { return _cod;  }
        const char* GetNume() const { return _nume; }

private:
        // date membre
        int _cod;
        char _nume[DIM_MAX_NUME];
};

// Clasa derivata Student : contine toate datele
// si functiile din clasa Persoana la care adauga
// datele regeritoare la grupa si an de studiu
class Student : public Persoana
{
public:
        // constructorul clasei Student este obligat sa
        // utilizeze constructorul clasei de baza Persoana
        // pentru initializarea atributelor mostenite
        Student(int cod, char* nume, int an, int grupa)
                : Persoana(cod, nume), _an(an), _grupa(grupa){}

        // adaugam functiile de acces pentru atributele adaugate
        int GetAn()    { return _an;    }
        int GetGrupa() { return _grupa; }
private:
        int _an, _grupa;
};
```

Funcții nemoştenibile

În general, clasele derivate moştenesc toate funcțiile şi atributele membre din clasa de bază. Există însă câteva excepții:

- **constructorii şi destructorii**: constructorii clasei de bază nu sunt moşteniți în clasa derivată; ei trebuie apelați folosind lista de inițializare a clasei derivate;
- **operatorul** =: operatorul nu este moştenit; compilatorul sintetizează un operator = care copiază bit cu bit atributele specifice clasei derivate şi apelează operatorul clasei de bază; în cazul în care clasa derivată conține un operator =, acesta trebuie să apeleze explicit operatorul din clasa de bază folosind sintaxa *Baza::operator=(param);*

Funcții virtuale şi clase abstracte

Clasele derivate pot suprascrie funcțiile clasei de bază după cum se poate vedea din exemplul următor:

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

// Clasa de baza
class Baza
{
public:
        void Functie()
        {
                cout << "[Baza] Functie()" << endl;
        }
};

// Clasa derivata
```

```
class Derivata : public Baza
{
	// Suprascrierea functiei
	void Functie()
	{
		cout << "[Derivata] Functie()" << endl;
	}
};

// Functia primeste ca parametru o
// referinta la o instanta a clasei
// de baza sau a clasei derivate
void Apelator(Baza& obiect)
{
	obiect.Functie();
}

void main()
{
	Baza b;
	Derivata d;

	Apelator(b);
	Apelator(d);
}
```

Implicit, funcția care va fi apelata de funcția *Apelator* este fixată la compilare. Din acest motiv, în ambele cazuri va fi apelată metoda *Functie* din clasa de bază. Limbajul C++ conține şi un mecanism de legare întârziată care permite selectarea funcției care trebuie executată la momentul execuției în funcție de tipul obiectului primit ca parametru. Acest mecanism poate fi utilizat prin folosirea cuvântului cheie *virtual* la declararea metodei. În exemplul prezentat clasa de baza va deveni:

```
// Clasa de baza
class Baza
{
public:
	virtual void Functie()
	{
		cout << "[Baza] Functie()" << endl;
	}
};
```

După efectuarea modificării va fi apelată versiunea corectă a funcției.

Mecanismul de legare întârziată funcționează numai când apelul este făcut printr-un pointer sau printr-o referință la obiect.

În cazul în care nu se dorește specificarea unei implementări în clasa de bază se pot folosi funcții virtuale pure declarate cu sintaxa ***virtual prototip_funcție =0.*** Clasele care includ astfel de funcții se numesc clase abstracte şi nu pot fi instanțiate direct. Acestea se folosesc atunci când nu există nici o implementare posibilă pentru clasa de bază.
Exemplu de utilizare clasa abstractă pentru calculul salariului în condițiile în care există două tipuri de angajați:

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

// clasa abstracta Anagajat
class Angajat
{
public:
	// functie virtuala pura pentru calculul salariului
	virtual double Salariu() = 0;
};
```

```
// clsa concreta AngajatOra (pentru angajatii platiti per ora)
class AngajatOra : public Angajat
{
public:

	// constructorul
	AngajatOra(double salariuOra, int numarOre)
		: _salariuOra(salariuOra), _numarOre(numarOre) {}

	// suprascrierea functiei de calcul pentru salariu
	double Salariu()
	{
		return _salariuOra * _numarOre;
	}
private:
	double _salariuOra;
	int _numarOre;
};

// clsa concreta AngajatContact (pentru angajatii cu salariu fix)
class AngajatContact : public Angajat
{
public:

	// constructorul
	AngajatContact(double salariuLunar)
		: _salariuLunar(salariuLunar){}

	// suprascrierea functiei de calcul pentru salariu
	double Salariu()
	{
		return _salariuLunar;
	}
private:
	double _salariuLunar;
};

// Folosirea clasei abstracte pentru calculul
// si afisarea salariului
void AfisareSalariu(Angajat& angajat)
{
	// folosim functia virtuala pentru calculul
	// salariului (indiferent de tipul de anagajat)
	cout << "Salariu: " << angajat.Salariu() << " lei." << endl;
}

void main()
{
	// construire obiecte
	AngajatOra anOra(40, 120);
	AngajatContact anContr(3900);

	// apelare functie de calcul salariu
	// pentru tipuri diferite de angajati
	AfisareSalariu(anOra);
	AfisareSalariu(anContr);
}
```

Se observă faptul că funcția de afișare pentru salarii poate primi o referință la un obiect din orice clasă derivată din *Angajat*. Astfel, putem adăuga noi tipuri de angajați fără să fie necesare modificări în codul deja existent.

Aplicație

Să se construiască o bibliotecă pentru rezolvarea ecuațiilor neliniare pe un interval dat şi să se exemplifice utilizarea acesteia pentru două funcții diferite.

Rezolvare:

Vom construi o clasă abstractă care conține interfața corespunzătoare unei funcții reale de un parametru real. Pentru Rezolvarea ecuației vom implementa o funcție care va primi ca parametrii o referință la un obiect reprezentând ecuația de rezolvat şi marginile intervalului.

Codul sursă al bibliotecii este:

```
#ifndef ECUATIE_H
#define ECUATIE_H

// precizia
const double eps = 0.000001;

// clasa abstracta care reprezinta o functie
// de un parametru real
class Functie
{
public:
	// operatorul () - functie virtuala pura
	virtual double operator() (double) = 0;
};

// functie pentru gasirea solutiei folosind
// metota bisectiei succesive
double Bisectie(Functie& f, double x1, double x2)
{
	double x;

	do
	{
		// obtinem mijlocul intervalului
		x = (x1 + x2) / 2;

		// retinem intervalul cu solutia
		if (f(x1) * f(x) < 0)
			x2 = x;
		else
			x1 = x;
	}
	// continuam pana cand intervalul se
	// incadreaza in precizia stabilita
	while (x2 - x1 > eps);

	// intoarcem solutia
	return x;
}

#endif //ECUATIE_H
```

Pentru exemplificarea modului de utilizare vom construi două funcții (o funcție liniară şi una neliniară). Construirea funcțiilor se face prin derivare din clasa abstractă *Funcție* şi suprascrierea operatorului (). După construirea claselor derivate corespunzătoare ecuațiilor de rezolvat este utilizată funcția din bibliotecă pentru obținerea soluției.

Codul sursă al programului este:

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

// includem biblioteca pentru rezolvarea ecuatiilor
#include "ecuatie.h"

// definim ecuatiile de rezolvat prin derivare
class FunctieLiniara : public Functie
{
public:
```

```
        double operator() (double x)
        {
                return 2*x + 3;
        }
};

class FunctieNeliniara : public Functie
{
public:
        double operator() (double x)
        {
                return log(x) + 3;
        }
};

void main()
{
        // construim functiile
        FunctieLiniara f1;
        FunctieNeliniara f2;

        // setam modul de afisare
        cout << fixed << setprecision(4);

        // afisam solutiile
        cout << "Solutia pentru functia liniara este: " << Bisectie(f1, -20, 20)
                << endl;
        cout << "Solutia pentru functia neliniara este: " << Bisectie(f2, 0, 5)
                << endl;
}
```