Clase

Noțiuni generale

Clasele reprezintă tipuri de data abstracte, asemănătoare structurilor, care încapsulează comportamentul şi datele asociate unei entități. Comportamentul este descris cu ajutorul metodelor (funcții incluse), iar datele cu ajutorul atributelor. O instanță (realizare concretă) a unei clase se numeşte obiect. Sintaxa utilizată pentru definirea claselor este asemănătoare cu cea pentru structuri:

```
class nume_clasa
{
        // atribute si metode
};
```

Obiectele pot fi create ca orice variabilă cu sintaxa: ***nume_clasa nume_obiect;***.

Metodele unei clase sunt similare funcțiilor obişnuite, cu două excepții:

- primul parametru (adăugat automat de către compilator) este un pointer către obiectul curent
- metodele pot accesa direct atributele (nu necesită folosirea operatorului „.”)

Accesul din exterior la atributele şi metodele unei clase se face folosind operatorul „.” (sau „->” în cazul pointerilor).

Atributele şi metodele unei clase pot fi grupate din punctul de vedere al drepturilor de acces in trei categorii:

- **private:** pot fi accesate doar din interiorul clasei
- **protected:** pot fi accesate din interiorul clasei şi din clase derivate
- **public:** pot fi accesate atât din interiorul cât şi din exteriorul clasei

O clasă poate oferi acces nerestricționat la membrii proprii prin utilizarea cuvântului cheie *friend* în una din formele:***friend prorotip_funcție;*** sau ***friend nume_clasă;***.

Controlul accesului permite dezvoltatorului clasei limitarea modului de utilizare al clasei şi ascunderea implementării interne a operațiilor. Prin aceasta cel care implementează clasa are posibilitatea de a modifica modul în care sunt stocate intern datele sau sunt implementate operațiile fără a afecta aplicațiile deja construite.

Modificatorii de acces pot fi utilizați atât in cadrul claselor cât şi în cadrul structurilor. Accesul este implicit *public* pentru structuri şi *private* pentru clase.

Metodele pot fi definite atât în interiorul clasei (inline) cât şi în afara acestora, folosind operatorul de rezoluție „::”.

Exemplu – Clasa număr complex:

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

```
// definire clasa
class Complex
{
public:             // sectiunea publica (accesibila din exterior)

       // definire metode inline
       double& getReal() { return real; }
       double& getImag() { return imag; }

       // declarare metoda definita in afara clasei
       double Modul();

private:     // sectiunea privata
       // definire atribute
       double real, imag;
};


// definire metoda in afara clasei
double Complex::Modul()
{
       return sqrt(real*real + imag*imag);
}

void main()
{
       // definire obiect
       Complex c1;

       // apelare metode pentru setarea valorii
       c1.getReal() = 4;
       c1.getImag() = 3;

       cout << "Modulul este " << c1.Modul() << endl;
}
```

Constructori

Constructorii sunt funcții speciale apelate automat de compilator la crearea unui nou obiect. În cazul în care nu există nici un constructor definit de către programator, compilatorul va sintetiza un constructor implicit fără parametri.

Constructorii sunt metode ale clasei cu următoarele caracteristici:

- au numele identic cu al clasei din care fac parte
- nu au tip returnat

Rolul constructorilor este de a inițializa atributele obiectului. O clasă poate avea mai mulți constructori care să difere prin numărul şi/sau tipul parametrilor.

Spre deosebire de funcțiile obişnuite, constructorii pot avea o listă de inițializare pentru atributele clasei de forma:

constructor(param constructor) : atribut_1(param), …, atribut_n(param) {…}

Exemple de constructori pentru clasa prezentată anterior:

```
class Complex
{
```

```
public:             // sectiunea publica (accesibila din exterior)

       // constructor implicit
       Complex() { real = 0; imag = 0; }

       // constructor folosind lista de initializare
       Complex(double r, double i) : real(r), imag(i) { }

       ...................
};

// definire obiect
Complex c1(4, 3);
```

Constructori de copiere

Constructorii de copiere sunt constructori care primesc ca parametru o referință la un obiect din aceeaşi clasă. În cazul în care nu există un asemenea constructor, compilatorul va genera un constructor de copiere care va copia bit cu bit atributele.

Exemplu de constructor de copiere pentru clasa complex:

```
class Complex
{
public:             // sectiunea publica (accesibila din exterior)

       ..............................

       // constructor de copiere definit in afara clasei
       Complex(Complex& c);

       ..............................

};

// definire constructor de copiere
Complex::Complex(Complex& c)
{
       real = c.real;
       imag = c.imag;
}
```

Constructorii de copiere apelați în următoarele situații:

- explicit de către programator la inițializarea unui alt obiect:

```
// apelare constructor de copiere
Complex c2 = c1;
// sau
Complex c2(c1);
```

ATENȚIE: constructorii de copiere se apelează numai la inițializare, nu şi la atribuire; construcția `Complex c2; c2 = c1;` nu va apela constructorul de copiere.

- automat de către compilator când este necesară crearea unei copii temporare (de exemplu când obiectul este trimis ca parametru prin valoare într-o funcție):

```
void AfisareComplex(Complex c)
{
```

```
	cout << "(" << c.getReal() << "," << c.getImag() << ")" << endl;
}

// trimitere ca parametru
AfisareComplex(c1);
```

Constructorii de copiere sunt utili atunci când comportamentul implicit sintetizat de către compilator nu este satisfăcător. Un astfel de caz sunt clasele care conțin atribute pointeri. Constructorul de copiere implicit va copia doar pointerul, nu şi zona de memorie adresată.

Destructori

Destructorii sunt metode speciale apelate de către compilator la ştergerea din memorie a obiectului. Numele destructorilor este format din caracterul „~” şi numele clasei. Destructorii nu au tip returnat şi nu pot primi parametri.

Exemplu:

```
// definire clasa
class Complex
{
public:			// sectiunea publica (accesibila din exterior)

	.................................

	// destructor
	~Complex()
	{
		// nu avem nimic de facut
	}

	.................................
};
```

Destructorii de folosesc pentru a elibera resursele alocate (dealocare zone de memorie alocate dinamic, închidere fişiere, …) la distrugerea obiectului.

Clasa Complex (exemplul complet):

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

// definire clasa
class Complex
{
public:			// sectiunea publica (accesibila din exterior)

	// constructor implicit
	Complex() { real = 0; imag = 0; }

	// constructor folosind lista de initializare
	Complex(double r, double i) : real(r), imag(i) { }

	// constructor de copiere definit in afara clasei
	Complex(Complex& c);

	// definire metode inline
	double& getReal() { return real; }
	double& getImag() { return imag; }
```

```
        // declarare metoda definita in afara clasei
        double Modul();

        // destructor
        ~Complex()
        {
                // nu avem nimic de facut
        }

private:        // sectiunea privata
        // definire atribute
        double real, imag;
};

// definire constructor de copiere
Complex::Complex(Complex& c)
{
        real = c.real;
        imag = c.imag;
}

// definire metoda in afara clasei
double Complex::Modul()
{
        return sqrt(real*real + imag*imag);
}

void AfisareComplex(Complex c)
{
        cout << "(" << c.getReal() << "," << c.getImag() << ")" << endl;
}

void main()
{
        // definire si initializare obiect
        Complex c1(8, 10);

        // apelare metode pentru setarea valorii
        c1.getReal() = 4;
        c1.getImag() = 3;

        // afisarea modulului
        cout << "Modulul este " << c1.Modul() << endl;

        // folosire constructor de copiere
        Complex c2 = c1;

        // trimitere ca parametru
        AfisareComplex(c1);
}
```

Aplicație

Sa se definească o clasa care sa încapsuleze principalele operații necesare lucrului cu un vector alocat dinamic.

Varianta 1

Vectorul are dimensiune fixa specificata in constructor. Atributele necesare pentru clasă vor fi un pointer către vectorul alocat dinamic şi numărul de elemente al vectorului.

Operațiile implementate de clasă sunt:

- constructor pentru alocarea memoriei
- constructor de copiere care creează o zonă de memorie nouă şi copiază elementele vectorului
- destructor pentru dealocarea zonei alocate
- funcții membru pentru obținerea numărului de elemente al vectorului şi pentru obținerea unei referințe la un element

Codul sursă complet, împreună cu un exemplu de utilizare:

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

class Vector
{
public:
	// Constructor
	Vector(int n);

	// Constructor de copiere
	Vector(Vector& v);

	// Destructor
	~Vector();

	// Intoarce dimensiunea vectorului
	int getDim();

	// Obtine o referinta la un element
	// al vectorului
	int& elem(int i);

private:
	int *vector, dim;
};

// Construieste un vector nou
Vector::Vector(int n) : dim(n)
{
	// alocam memoria pentru vector
	vector = new int[dim];
}

// Construieste un vector pe baza unui
// vector existent (pentru initializare
// sau trimitere de parametri prin
// valoare in functii)
Vector::Vector(Vector& v)
{
	// construim noul vector
	dim = v.dim;
	vector = new int[dim];

	// si copiem elementele
	for (int i = 0; i < dim; i++)
		vector[i] = v.vector[i];
}

// Elibereaza memoria alocata dinamic
Vector::~Vector()
{
	// dezalocam memoria alocata
	// in constructor
```

```
	delete [] vector;
}

int Vector::getDim()
{
	return dim;
}

int& Vector::elem(int i)
{
	return vector[i];
}


// Exemplu de functie pentru calculul mediei
double CalculMedie(Vector v)
{
	int suma = 0;
	for (int i = 0; i < v.getDim(); i++)
		suma += v.elem(i);

	return (double)suma / v.getDim();
}

void main()
{
	// exemplu de utilizare clasa pentru calculul mediei

	// citire dimensiune
	int n;
	cout << "n=";
	cin >> n;

	// creare vector si citire elemente
	Vector v(n);
	for (int i = 0; i < n; i++)
	{
		cout << "Elementul " << i + 1 << ":";
		cin >> v.elem(i);
	}

	// calcul medie

	cout << "Media este " << CalculMedie(v) << endl;
}
```

Varianta 2

Dimensiunea vectorului poate varia în timpul execuției programului. În acest caz vom avea nevoie de un atribut suplimentar pentru a reține zona de memorie efectiv utilizată.

În plus față de prima variantă au fost introduse următoarele operații:

- adaugaElement: adaugă un element nou la sfârşitul vectorului, mărind zona alocată dacă este cazul
- stergeElemente: şterge o zonă din vector
- rezerva: o metoda privată pentru redimensionarea zonei de memorie

Codul sursă:

```
#include <iostream>
```

```
using namespace std;

class Vector
{
public:
	// Constructor
	Vector(int n);

	// Constructor de copiere
	Vector(Vector& v);

	// Destructor
	~Vector();

	// Intoarce dimensiunea vectorului
	int getDim();

	// Operatii cu elemente
	int& elem(int i);
	void adaugaElement(int val);
	void stergeElemente(int pozStart, int pozStop);

private:

	// Rezerva memorie suplimentara pentru vector
	void rezerva(int n);

	int *vector, dimAlocata, dimUtilizata;
};

// Construieste un vector nou
Vector::Vector(int n = 0) : dimUtilizata(n)
{
	// alocam memoria pentru vector
	dimAlocata = n;
	vector = new int[dimAlocata];
}

// Construieste un vector pe baza unui
// vector existent (pentru initializare
// sau trimitere de parametri prin
// valoare in functii)
Vector::Vector(Vector& v)
{
	// construim noul vector
	dimUtilizata = v.dimUtilizata;
	dimAlocata = v.dimAlocata;
	vector = new int[dimAlocata];

	// si copiem elementele
	for (int i = 0; i < dimUtilizata; i++)
		vector[i] = v.vector[i];
}

// Elibereaza memoria alocata dinamic
Vector::~Vector()
{
	// dezalocam memoria alocata
	// in constructor
	delete [] vector;
}

int Vector::getDim()
{
	return dimUtilizata;
}

int& Vector::elem(int i)
```

```
{
	return vector[i];
}

// Adauga un element la sfarsitul
// vectorului
void Vector::adaugaElement(int val)
{
	// alocam memorie suplimentara
	// daca este cazul
	if (dimUtilizata == dimAlocata)
		rezerva((int)(dimAlocata*1.2) + 1);

	// si adaugam elementul
	vector[dimUtilizata] = val;
	dimUtilizata++;
}

// Sterge elemente din vector de la
// pozitia pozStart (inclusiv) pana
// la pozitia pozStop (exclusiv)
void Vector::stergeElemente(int pozStart, int pozStop)
{
	// mutam elementele
	for (int i = pozStop; i < dimUtilizata; i++)
		vector[i - (pozStop - pozStart)] = vector[i];

	// ajustam dimensiunea
	dimUtilizata = dimUtilizata - (pozStop - pozStart);
}

void Vector::rezerva(int n)
{
	if (n > dimAlocata)
	{
		// alocam spatiu pentru
		// noul vector
		dimAlocata = n;
		int* temp = vector;
		vector = new int[dimAlocata];

		// copiem elementele semnificative
		for (int i = 0; i < dimUtilizata; i++)
			vector[i] = temp[i];

		// stergem vechiul vector
		delete [] temp;
	}
}

// Exemplu de functie pentru calculul mediei
double CalculMedie(Vector v)
{
	int suma = 0;
	for (int i = 0; i < v.getDim(); i++)
		suma += v.elem(i);

	return (double)suma / v.getDim();
}

void main()
{
	// exemplu de utilizare clasa

	// citire dimensiune
	int n;
	cout << "n=";
	cin >> n;
```

```
	// creare vector si citire elemente
	Vector v; // nu este necesar sa specificam dimensiunea
	for (int i = 0; i < n; i++)
	{
		cout << "Elementul " << i + 1 << ":";
		int val;
		cin >> val;

		// zona de memorie se va redimensiona automat
		v.adaugaElement(val);
	}

	// calcul medie
	cout << "Media este " << CalculMedie(v) << endl;

	// stergere elemente
	if (n > 2)
	{
		// stergem elementele 1 si 2
		v.stergeElemente(1, 3);

		// afisam vectorul
		for (int j = 0; j < v.getDim(); j++)
			cout << v.elem(j) << " ";
		cout << endl;
	}

}
```