Supraîncărcarea operatorilor

Noțiuni generale

Supraîncărcarea operatorilor permite utilizarea operatorilor standard ai limbajului pentru clasele proprii. Aceasta se poate realiza prin definirea unor funcții cu denumire specială (se foloseşte cuvântul cheie *operator*).

La supraîncărcarea operatorilor există unele restricții:

- nu se pot modifica caracteristicile de bază ale operatorilor (prioritate, direcție de evaluare, asociativitate, cardinalitate)
- nu pot fi introduşi operatori noi față de cei existenți în limbaj
- nu se pot supraîncărca operatorii pentru tipurile de bază
- nu se pot supraîncărca operatorii . .* :: ? şi *sizeof*

Operatorii pot fi supraîncărcați prin funcții membre în clasă sau prin funcții normale. În cazul în care se folosesc funcții membre numărul parametrilor funcției ca fi cu 1 mai mic decât cardinalitatea operatorului (primul operand este considerat pointerul *this*).

Pentru detalii, vezi cartea „Programarea orientată obiect în limbajul C++”, paginile 52-78.

Aplicație

Să se construiască o clasă *Colectie* memorată sub formă de listă simplu înlănțuită pentru stocarea numerelor întregi care să permită efectuarea principalelor operații folosind operatori şi care să permită parcurgerea colecției folosind un iterator. Să se scrie un program principal pentru demonstrarea funcționării clasei.

Pentru rezolvare vom avea nevoie de trei clase:

- clasa principală numită *Colectie* care conține implementarea operațiilor
- clasa privata *Nod* (inclusă în clasa *Colectie*) care conține datele referitoare la un nod din listă
- clasa *Iterator* (inclusă în clasa *Colectie*) care permite enumerarea elementelor colecției

Implementarea C++ a claselor şi a programului principal pentru testare:

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

/*****************************************************************
 * Clasa Colectie
 * --------------------------
 * Implementeaza o colectie de numere intregi sub forma de lista
 * simplu inlantuita.
 *****************************************************************/
class Colectie
{
        struct Nod;    // anuntare clasa Nod (pentru Iterator)
```

```
	/*** Interfata publica ***/

public:

	// Constructori
	Colectie() : cap(NULL) {}	// constructorul implicit
	Colectie(const Colectie& c)	// constructorul de copiere
	{
		cap = NULL;

		// parcurgem colectia cu un iterator
		for (Iterator i = c.Inceput(); i != c.Sfarsit(); i++)
			// si adaugam nodurile in colectia noastra
			Adaugare(*i);
	}

	// operatorul de atribuire
	const Colectie& operator=(const Colectie& c)
	{
		// stergem colectia curenta
		GolireColectie();

		// si copiem elementele noi
		for (Iterator i = c.Inceput(); i != c.Sfarsit(); i++)
			Adaugare(*i);

		return (*this);
	}

	// destructorul
	~Colectie()
	{
		GolireColectie();
	}

	// operatorul pentru acces direct la elemente
	int& operator[] (int k) const
	{
		// presupunem ca indicele este in interiorul colectiei
		Nod *p = cap;
		for (int i = 0; i < k; i++)
			p = p->Urmator;

		return p->Valoare;
	}

	// operatii cu elemente
	void Adaugare(int valoare)	// adaugare element la sfarsit
	{
		// cazul 1: lista vida
		if (cap == NULL)
			cap = new Nod(valoare);
		else
		{
			// cazul 2: lista nu este vida
			Nod *p = cap;
			while (p->Urmator != NULL)
				p = p->Urmator;

			p->Urmator = new Nod(valoare);
		}
	}

	const Colectie& operator +=(int valoare)
	{
		Adaugare(valoare);
		return (*this);
	}

	void Inserare(int poz, int val)		// inserare element
	{
		// cazul 1: inserare la inceputul colectiei
		if (poz == 0)
		{
			cap = new Nod(val, cap);
			return;
		}
```

```
		// cazul 2: inserare in interiorul colectiei

		// cautam pozitia de inserare
		Nod *p = cap;
		for (int i = 0; i < poz-1; i++)
			p = p->Urmator;

		// si inseram nodul
		p->Urmator = new Nod(val, p->Urmator);
	}

	void Stergere(int poz)		// sterge elementul de pe pozitia
specificata
	{
		// cazul 1: stergere la inceputul colectiei
		if (poz == 0)
		{
			Nod *temp = cap;
			cap = cap->Urmator;
			delete temp;

			return;
		}

		// cazul 2: stergere din interiorul colectiei

		// cautam pozitia de sters
		Nod *p = cap;
		for (int i = 0; i < poz-1; i++)
			p = p->Urmator;

		// si stergem nodul
		Nod *temp = p->Urmator;
		p->Urmator = p->Urmator->Urmator;
		delete temp;
	}

	void GolireColectie()		// sterge toate elementele din colectie
	{
		while (cap != NULL)
			Stergere(0);
	}

	// concatenare colectii
	const Colectie& operator +=(const Colectie &c)
	{
		// adaugam elementele noi
		for (Iterator i = c.Inceput(); i != c.Sfarsit(); i++)
			Adaugare(*i);
	}

	const Colectie operator + (const Colectie &c)
	{
		Colectie rezultat = (*this);

		// adaugam elementele noi
		for (Iterator i = c.Inceput(); i != c.Sfarsit(); i++)
			rezultat.Adaugare(*i);

		return rezultat;
	}

	// operatii diverse
	int NumarElemente() const		// obtine numarul de elemente
	{
		Nod *p = cap;
		int i = 0;

		while (p != NULL)
		{
			p = p->Urmator;
			i++;
		}

		return i;
	}
```

```
	bool EGoala() const                          // testare colectie vida
	{
		return cap == NULL;
	}
	bool operator!() const
	{
		return !EGoala();
	}

	// operatori de verificare egalitate
	bool operator == (const Colectie& c) const
	{
		Iterator i1 = Inceput();
		Iterator i2 = c.Inceput();

		while (i1 != Sfarsit() && i2 != c.Sfarsit())
		{
			if ((*i1) != (*i2))
				return false;

			i1++;
			i2++;
		}

		// daca nu au aceeasi lungime sunt diferite
		if ((i1 == Sfarsit() && i2 != c.Sfarsit()) || (i1 != Sfarsit() && i2 ==
c.Sfarsit()))
			return false;

		// au toate elementele egale si aceeasi dimensiune
		return true;
	}

	bool operator != (const Colectie& c) const
	{
		return !((*this) == c);
	}

	// implementare iterator
	class Iterator
	{
	public:
		// operatorul de dereferentiere
		int& operator *() const
		{
			return nodCurent->Valoare;
		}

		// operatorul de avansare in colectie (postincrementare)
		const Iterator& operator++(int)
		{
			nodCurent = nodCurent->Urmator;
			return (*this);
		}

		// operatorul de atribuire
		const Iterator& operator=(const Iterator& iterator)
		{
			nodCurent = iterator.nodCurent;
			return (*this);
		}

		// operatorii de comparatie
		bool operator==(const Iterator& iterator)
		{
			return nodCurent == iterator.nodCurent;
		}

		bool operator != (const Iterator& iterator)
		{
			return !((*this) == iterator);
		}

	private:
		// clasa colectie trebuie sa aiba acces la
		// sectiunea privata a Iteratorului
```

```
                friend class Colectie;

                // constructor privat (va fi apelat doar
                // de catre clasa Colectie)
                Iterator(Nod *nod)
                        : nodCurent(nod) {}

                Nod *nodCurent;
        };

        // functii pentru iteratori
        // intoarce un iterator la inceputul colectiei
        Iterator Inceput() const
        {
                return Iterator(cap);
        };

        // intoarce un iterator dupa sfarsitul colectiei
        Iterator Sfarsit() const
        {
                return Iterator(NULL);
        };

        /*** Sectiunea privata ***/

private:

        // clasa folosita intern pentru memorarea unui nod al listei
        struct Nod
        {
                // constructor
                Nod(int valoare, Nod *urmator = NULL)
                        : Valoare(valoare), Urmator(urmator) {}

                // date membru
                int Valoare;
                Nod *Urmator;
        };

        // capul listei
        Nod *cap;
};

// operatorii de I/E pentru Colectie
ostream& operator << (ostream& out, const Colectie& c)
{
        for (Colectie::Iterator i = c.Inceput(); i != c.Sfarsit(); i++)
                out << (*i) << " ";

        return out;
}

void main()
{
        // creare colectie
        Colectie c1;

        // adaugare elemente in colectie
        c1.Adaugare(7);
        c1.Adaugare(8);
        c1 += 23;
        c1 += 72;

        // afisare colectie folosind operatorul <<
        cout << "Elemente dupa adaugare: " << c1 << endl;

        // inserare elemente
        c1.Inserare(0, 2);
        c1.Inserare(5, 22);
        c1.Inserare(5, 21);
        cout << "Elemente dupa inserare: " << c1 << endl;

        // stergere elemente
        c1.Stergere(3);
        cout << "Elemente dupa stergere: " << c1 << endl;

        // accesare elemente folosind operatorul []
```

```
	cout << "Primul element este:" << c1[0] << endl;
	cout << "Ultimul element este:" << c1[c1.NumarElemente() - 1] << endl;

	// test coada vida
	if (c1.EGoala())
		cout << "Colectia este goala." << endl;
	else
		cout << "Colectia nu este goala." << endl;

	// construim o colectie noua folosind constructorul de copiere
	Colectie c2 = c1;

	// stergere colectie
	c1.GolireColectie();
	cout << "Dupa golire:" << endl;
	if (c1.EGoala())
		cout << "Colectia este goala." << endl;
	else
		cout << "Colectia nu este goala." << endl;

	// afisare colectia 2
	cout << "Colectia 2:" << c2 << endl;

	cout << "Inainte de operator =" << endl;

	// verificare egalitate
	if (c1 == c2)
		cout << "Colectiile sunt egale" << endl;
	if (c1 != c2)
		cout << "Colectiile nu sunt egale" << endl;

	// testare operator =
	c1 = c2;

	cout << "Dupa de operator =" << endl;
	// verificare egalitate
	if (c1 == c2)
		cout << "Colectiile sunt egale" << endl;
	if (c1 != c2)
		cout << "Colectiile nu sunt egale" << endl;

	// afisare colectia 1
	cout << "Colectia 1 dupa atribuire:" << c1 << endl;

	// test concatenare
	cout << "Concatenare: " << c1 + c2 << endl;
}
```