Supra încărcarea operatorilor

"When in doubt, do as the *int*s do!"

(Scott Meyers, *More Effective C++*)

Limbajul C++ permite extinderea operatorilor astfel încât aceştia să suporte şi tipurile introduse de utilizator. Acest lucru se realizează prin definirea unor funcţii al căror nume conţine cuvântul cheie *operator*, urmat de identificatorul operatorului; o astfel de funcţie, numită *funcţie operator*, este declarată şi poate fi apelată exact ca orice altă funcţie. Deoarece o funcţie operator trebuie să conţină măcar un argument de un tip-utilizator (ceea ce garantează că sensul unei expresii formate doar din instanţe ale tipurilor predefinite nu poate fi modificat), ea *supraîncarcă* (şi nu *redefineşte*) semnificaţia unui operator.

1. Caracteristicile funcţiilor operator

O *funcţie operator* poate fi o funcţie membru ne-statică (cu excepţia *new* şi *delete*) a unei clase sau o funcţie globală (*friend* sau nu). Un operator poate fi supraîncărcat pentru un tip simultan prin funcţii membru şi funcţii globale; într-o astfel de situaţie rezoluţia supraîncărcării determină ce versiune a funcţiei operator va fi aleasă la apel.

În cazul unei funcţii operator globale, măcar un argument trebuie să fie de un tip-utilizator. În particular, nu se pot defini funcţii operator care să acţioneze exclusiv pe pointeri. Cu alte cuvinte, limbajul C++ este *extensibil*, dar nu modificabil.

Spre deosebire de funcţiile obişnuite, funcţiile operator (cu excepţia lui *operator()*()) nu pot avea valori implicite pentru argumente. Dacă sunt funcţii membru, funcţiile operator sunt moştenite în acelaşi mod în care sunt moştenite funcţiile membru ale unei clase de bază; excepţie face *operator=*(), această funcţie fiind implicit declarată pentru orice clasă, ceea ce face ca varianta din clasa de bază să fie întotdeauna ascunsă de varianta din clasa derivată!

Numărul de argumente şi *precedenţa* unui operator nu pot fi modificate prin supraîncărcare; de asemenea, nu se pot introduce operatori noi!

Înţelesul predefinit al operatorilor nu este păstrat de compilator pentru funcţiile operator introduse de utilizator. De exemplu, pentru x de tip *int*, expresia *++x* este echivalentă cu expresia *x += 1* şi cu expresia *x = x + 1*. Pentru un tip utilizator Y, aceste relaţii nu vor avea loc decât dacă utilizatorul are grijă să le definească corespunzător; compilatorul nu va genera o definiţie a lui Y::*operator+=*() din definiţiile lui Y::*operator+*() şi Y::*operator=*(), chiar dacă acest lucru este posibil! Un prim motivele care justifică această decizie se referă la eficienţă: versiunea creată automat a lui += ar utiliza obligatoriu un obiect temporar şi deci ar putea fi cu mult mai puţin eficientă decât o versiune definită explicit de utilizator, care ar putea să evite temporarul. Un al doilea motiv ar fi faptul că utilizatorul ar putea să nu dorească să-şi doteze clasa cu operatorul +=!

2. Funcţie membru sau funcţie globală?

Un operator binar **x@y** poate fi supraîncărcat fie prin funcţii membru cu un singur argument (**x.operator@(y)**), fie prin funcţii globale cu două argumente (**operator@(x, y)**). Rezoluţia supraîncărcării determină care versiune va fi selectată:

```
//exemplul 1
class X{
        public:
        void operator + (int);
        X(int);
};
void operator + (X, X);
void operator + (X, double);
...
X a;
a+1;            //a.operator+(1)
a+1;            //::operator+(X(1), a)
a+1.0;          //::operator+(a, 1.0)
```

Un operator unar **@x** (sau **x@**) poate fi supraîncărcat fie prin funcţii membru fără argumente (**x.operator@()**), fie prin funcţii globale globale cu un argument (**operator@(x)**). Deoarece ++ şi -- sunt singurii operatori unari care pot apărea atât în formă *pre-fixată* cât şi în formă *post-fixată*, un argument suplimentar, de tip *int*, este utilizat în cazul lor pentru a face distincţia între cele două forme.

În general, *se recomandă* supraîncărcarea operatorilor prin *funcţii membru*; supraîncărcarea prin *funcţii globale* să se utilizeze doar atunci *când este necesar*! Când este necesar? Când trebuie să se asigure *simetria*, *comutativitatea* sau când se doreşte *utilizarea conversiilor implicite*!

```
//exemplul 2
class X{
        public:
        void operator + (X);
        bool operator == (const X&) const;
        X(int);
        f();
};

X a;

//+ este comutativ; a+1 şi 1+a sunt echivalente; şi totuşi:

a + 1;          // OK, a.operator+(X(1))
1 + a;          // eroare la compilare: nu există int::operator + (X) şi nici ::operator + (int, X)!
```

De ce compilatorul nu evaluează expresia "**1+a**" prin "**1.operator+(a)**" şi deci prin "**X(1).operator+(a)**"? Din acelaşi motiv pentru care expresia "**1.f()**" nu este evaluată prin "**X(1).f()**": standardul C++ interzice conversiile implicite definite de utilizator şi aplicate operandului stâng al lui **.** sau al lui **->**! Acest lucru are două consecințe imediate. Prima este aceea că, dacă primul operand al unui operator este de tip predefinit, atunci operatorul trebuie supraîncărcat prin funcție globală:

```
//dacă operator + este supraîncărcat prin funcția globală
void operator + (X, X);
//atunci:
a + 1;          //OK, ::operator + (a, X(1))
1 + a;          //OK, ::operator + (X(1), a)
```

A doua consecință este faptul că, dacă un operator impune ca primul operand să fie *l-value*, acel operator trebuie obligatoriu supraîncărcat prin funcție membru. Este cazul operatorilor = (şi variantele sale +=, - =, *=, /=, %=), (), [], -> şi ->*, care se supraîncarcă întotdeauna prin funcții membru.

Faptul că operatorul == este supraîncărcat prin funcție membru face ca el să fie asimetric. De ce? Pentru că primeşte argumente de tipuri diferite, pentru care nu există definită nici o conversie standard: un pointer (de tipul "const X* const this", deoarece este funcție membru const) şi o referință (const X&). Pentru a se asigura simetria, este necesar ca operatorul să fie supraîncărcat prin funcție globală:

```
//bool operator==(const X&, const X&);
```

De ce ar fi necesară simetria? Algoritmii STL se bazează pe existența unor versiuni simetrice ale operatorului ==; de exemplu, un container STL de obiecte aparținând unui tip care nu îndeplinește această cerință nu poate fi sortat printr-un algoritm STL!

3. Care operatori nu pot/nu se recomandă a fi supraîncărcați?

Următorii operatori **nu pot** fi supraîncărcați:

- **.** (operatorul de acces direct la membru)
- **.*** (operatorul de dereferențiere a unui pointer la membru)
- **::** (operatorul de rezoluție a domeniului de vizibilitate)
- **sizeof** (operatorul de calcul a dimensiunii)
- **typeid** (operatorul de returnare a informației despre tip)
- **?:** (operatorul condițional)
- **static_cast<>**, **const_cast<>**, **reinterpret_cast<>**, **dynamic_cast<>** (operatorii de conversie)
- **new** şi **delete**

În cazul lui *new* şi *delete*, nu se poate modifica *ce* fac aceşti operatori, ci doar *cum* fac! Pentru mai multe detalii, consultați materialul dedicat acestor doi operatori.

Operatorii **&&**, || şi **,** se recomandă a nu fi supraîncărcați niciodată!

Semantica de tip operator face ca C++ să *scurt-circuiteze* evaluarea expresiilor booleene: în momentul în care valoarea de adevăr a unei expresii a fost stabilită, evaluarea expresiei încetează, chiar dacă anumite părți din expresie nu au fost încă evaluate! Mai mult decât atât, funcționarea corectă a unor programe depinde de scurt-circuitarea evaluării:

```
//exemplul 3
char* p;
...
if ((p != 0) && strlen(p) > 10) ...
```

Deoarece invocarea lui *strlen*() asupra unui pointer *NULL* conduce la comportament nedefinit (conform standardelor C şi C++), este foarte important ca, dacă p == 0, partea a doua a expresiei să nu se evalueze!

C++ permite supraîncărcarea operatorilor && şi ||; dacă veți face acest lucru, regulile jocului se schimbă pentru că veți utiliza pentru aceşti operatori o *semantică de tip funcție*! De exemplu, "**e_1&&e_2**" devine "**operator&&(e_1, e_2)**"; în acest caz, deoarece parametrii unei funcții sunt întotdeauna evaluați, scurt-circuitarea dispare! Mai mult decât atât, cum ordinea de evaluare a parametrilor unei funcții este lăsată nespecificată de către standard, este posibil ca e_2 să se evalueze înaintea lui e_1, în directă contradicție cu semantica operator, în care argumentele se evaluează întotdeauna de la stânga spre dreapta. Aceeaşi problemă apare şi pentru **,** (operatorul de înlănțuire a expresiilor) care-şi evaluează întotdeauna întâi operandul stâng, apoi pe cel drept.

Deoarece nu veți putea reuşi să faceți aceşti operatori să se comporte aşa cum s-ar aştepta utilizatorii, este recomandat să nu-i supraîncărcați.

4. operator= ()

Deoarece operandul stâng trebuie să fie *l-value*, operatorul de atribuire se supraîncarcă întotdeauna prin funcție membru; mai mult decât atât, compilatorul generează automat operator de atribuire pentru fiecare clasă.

```
//exemplul 4
class X{
        public:
        X(const char*);
        X();
};
X a,b,c;
a = b = c = "sir de caractere";
```

Deoarece operatorul de atribuire este drept-asociativ, compilatorul traduce atribuirea înlănțuită din exemplul 4 prin **a.operator= (b.operator= (c.operator= (X("sir de caractere"))))**. Se observă că operatorul de atribuire primeşte ca argument un obiect de acelaşi tip; din motive de eficiență, acest obiect este transmis prin referință (se poate transmite şi prin valoare, dar implică suplimentar un apel de constructor de copie şi un apel de destructor). Deoarece atribuirea nu modifică

argumentul, se utilizează referințe const; se pot utiliza şi referințe obişnuite, dar în acest caz apelurile de tip **c.operator=("sir de caractere")** nu ar mai reuşi. Aceasta deoarece, prin conversia implicită **X("sir de caractere"**), este implicat suplimentar un obiect temporar; obiectele temporare sunt întotdeauna const, şi deci se pot apela doar metodele **const**, ceea ce, evident, operatorul de atribuire nu este! Deci tipul argumentului este **const X&**.

Atribuirile înlănțuite fac necesar ca tipul returnat de operatorul de atribuire să fie compatibil cu tipul recepționat; cum tipul recepționat este **const X&**, tipul returnat poate fi **X&** sau **const X&**. Din dorința de a împiedica atribuirile de genul "(a=b)=c", puteți fi tentați să returnați **const X&**. Să nu faceți asta! De ce? Un prim motiv ar fi faptul că introduceți o inconsistență în limbaj: dacă a, b şi c ar fi de tip *int*, expresia ar fi validă! Un al doilea motiv, mult mai serios, e acela că tipul X nu ar mai fi compatibil cu containerele STL, care cer explicit ca operatorul de atribuire să returnze referințe **ne-const**! Aşadar, tipul returnat este **X&**. Ajungem aşadar la următorul prototip al operatorului de atribuire:

X& X::operator= (const X& other);

O altă întrebare este: pe cine returnați, obiectul primit ca argument sau obiectul curent **(*this**)? Cum tipul recepționat este **const X&** iar tipul returnat este **X&**, încercarea de a returna obiectul primit ca argument (other) ar genera eroare de compilare (se elimină **const**)! Se returnează deci, întotdeauna, referință la obiectul curent; cum operator=() este funcție membru, această referință va fi ***this**.

Din motive de eficiență trebuie întotdeauna să vă protejați împotriva auto-atribuirilor:

```
//exemplul 5
X a;
a = a;          //nu veți scrie aşa ceva niciodată, nu?
X& b = a;
...
a = b;          //sunteți sigur?
```

Discuția e mai amplă, pentru că implică a stabili ce înseamnă că două obiecte sunt egale. De exemplu, două instanțe s_1 şi s_2 ale clasei *std::string*, care conțin şiruri de caractere identice, sunt egale sau nu? Dacă răspunsul este da, atunci atribuirea $s_1 = s_2$ nu ar trebui să facă nimic (o tehnică uzuală de implementare a atribuirii ar consta în

eliberarea resursele din s_1, alocarea de memorie şi copierea şirul de caractere din s_2, ceea ce, în acest caz, ar fi ineficient)! A stabili ce însemnă că două obiecte sunt egale depinde de problema modelată; deoarece fiecare obiect are o adresă unică, un test de egalitate între *this* şi *&other* este, adesea, suficient.

Nu uitați că, pentru fiecare nouă clasă, compilatorul generează automat operatorul de atribuire; în cazul moştenirii, aceasta are o importanță deosebită pentru că varianta din clasa derivată nu gestionează şi atribuirea către membrii proveniți din clasa de bază! Pentru a extinde funcționalitatea operatorului de atribuire din clasa derivată se procedează astfel:

```
//exemplul 6
class Y : public X{
        ...
        public:
        Y& operator= (const Y& other){
                if (this == &other) return *this;
                X::operator= (other);
                ...
                return *this;
        }
};
```

5. Nu încercați să returnați prin referință când trebuie să returnați prin valoare!

Să presupunem că, din motivele enumerate la 2, pentru un tip T supraîncărcați operatorul binar + prin funcție globală; în acest caz, expresia **a+b** devine **operator+(a, b)**. Această funcție operator recepționează argumentele prin referință (se poate şi prin valoare, dar nu ar fi la fel de eficient); cum a şi b nu se modifică în urma execuției funcției operator, se utilizează referințe const. Evident, "suma" a două valori de tip T este o valoare de tip T, deci funcția operator trebuie să returneze o valoare de tip T; poate să o returneze prin referință (mai eficient)?

```
//exemplul 7
T& operator+ (const T& x, const T& y){
        T aux;
        //calculează rezultatul în aux
        return aux;
}
```

Deoarece aux este o variabilă locală, memoria alocată ei va fi dealocată la terminarea funcției operator şi referința returnată va fi invalidă! Putem încerca să

evităm acest lucru garantând existența locației de memorie a variabilei aux şi în afara funcției operator:

```
//exemplul 8
T& operator+ (const T& x, const T& y){
        static T aux;
        //calculează rezultatul în aux
        return aux;
}
```

Pare ok; însă, în acest caz, o expresie de tipul **(a+b)==(c+d)** se va evalua întotdeauna la true (deoarece se vor testa de egalitate două referințe către aceeaşi zonă de memorie)! Ultima încercare: salvăm rezultatul într-o variabilă alocată dinamic, a.î. variabila există şi după terminarea funcției operator iar apelurile succesive ale funcției operator plasează rezultatul în variabile distincte:

```
//exemplul 9
T& operator+ (const T& x, const T& y){
        T* aux = new T;
        //calculează rezultatul în *aux
        return *aux;
}
```

Se pare că ați reuşit şi meritați un premiu! Da, meritați cu adevărat un premiu dacă veți reuşi să convingeți un client să utilizeze acest operator în felul următor:

```
//exemplul 10
T a, b;
T* p(&(a+b));
//utilizează *p
delete p;
```

Cu alte cuvinte, trebuie să vă convingeți clientul ca, după fiecare apelare a operatorului +, să elibereze zona de memorie în care este păstrat rezultatul! Vă urez succes!

Şi dacă clientul trebuie să scrie o expresie de forma **a+b+c**?

Resemnați-vă şi acceptați că operatorul + are următorul prototip:

const T operator+(const T&, const T&);

Stai puțin: *const T*? De ce *const*?

Argumentul filosofic: “suma” a două valori de tip T este o valoare de tip T, iar o valoare nu poate fi modificată! Argumentul practic: lipsa lui *const* ar da posibilitatea clientului să scrie expresia **(a+b)=c**, evident eronată d.p.d.v. logic. Cum

compilatorul refuză o astfel de expresie dacă a, b şi c ar fi de tip int, motto-ul articolului ne invită să respingem această expresie şi pentru clasa noastră!

N-ar trebui operator+() să fie funcție friend? Răspunsul, la secțiunea 7!

6. Dotați-vă clasele cu operatorii de atribuire compusă!

Ştiți din C că **a=a+b** este echivalentă cu **a+=b**; de câte ori ați folosit a doua formă? Dacă răspunsul diferă de întotdeauna, atunci aflați că *a doua formă este mult mai eficientă decât prima*!

Operatorul + este obligat să calculeze rezultatul într-o variabilă locală auxiliară şi să-l returneze prin valoare. Operatorul += este supraîncărcat obligatoriu prin funcție membru:

```
T& T::operator+=(const T& b){
//calculează rezultatul în *this
return *this;
}
```

Nu mai este nevoie de variabila auxiliară, căci rezultatul se calculează în obiectul curent; nu se mai returnează prin valoare, ci se returnează referință la obiectul curent! Q.E.D.

7. Fereşte-mă, Doamne, de prieteni...

Oferiți unei funcții globale statutul de friend (adică acces la detaliile de implementare) doar dacă acea funcție are, cu adevărat, nevoie de acest lucru. Are nevoie, de exemplu, *operator*+ să fie friend?

Dacă ați urmat sfatul de la punctul 6 atunci v-ați dotat clasa cu operatorul +=(); acesta este supraîncărcat prin funcție membru şi, evident, publică; atunci următoarea definiție este suficientă pentru operatorul +:

```
//exemplul 11
inline const T operator+ (const T& x, const T& y){
        return T aux(x)+=y;
}
```

8. ++ şi - -

Aceşti doi operatori sunt unici în C++, în sensul că ei pot fi utilizați atât în formă prefixată cât şi în formă postfixată. Un argument suplimentar, de tip int, este utilizat pentru a permite compilatorului să selecteze versiunea postfixată; versiunea prefixată nu are argumente. Mult mai important, cele două versiuni returnează tipuri diferite, iar *versiunea prefixată este întotdeauna mai eficientă*, aşa cum se va vedea în continuare.

O implementare uzuală a pre-incrementării poate arăta astfel:

```
//exemplul 12
T& T::operator++(){
        *this += 1;
        return *this;
}
```

Ştiți din C că, pentru a de tip int, expresia **++a** înseamnă "incrementează-l pe a şi returnează noua valoare"; cu alte cuvinte, operatorul returnează valoarea calculată, adică ***this**. De asemenea, expresia **++++a** este validă şi reprezintă două incrementări succesive ale lui a; motto-ul articolului ne îndeamnă să asigurăm claselor noastre aceeaşi funcționalitate, ceea ce însemnă că operatorul de pre-incrementare trebuie să returneze ***this** prin referință!

Tot din C ştiți că, pentru a de tip int, expresia **a++** însemnă "incrementează-l pe a şi returnează valoarea actuală"; aceasta înseamnă că valoarea actuală trebuie salvată mai întâi într-o variabilă auxiliară, pentru a putea fi returnată la sfârşit, ceea ce conduca la necesitatea de a returna prin valoare! De asemenea, expresia **a++++** este invalidă şi compilatorul semnalează eroare; asigurăm aceeaşi funcționalitate claselor noastre declarând const valoarea returnată. E chiar indicat să facem acest lucru pentru că, în mod evident, **a++++** este o eroare logică şi nu face ceea ce un utilizator mai puțin experimentat ar crede că face. O implementare uzuală a post-incrementării poate arăta astfel:

```
//exemplul 13
const T T::operator++(int){
        T oldValue = *this;
        ++(*this);          //pre-incrementare
        return oldValue;
}
```

9. De ce << şi >> sunt supraîncărcaţi prin funcţii globale?

Pare şi este logic ca cei doi operatori să fie supraîncărcați prin funcții membru:

```
//exemplul 14
class T{
        public:
        ostream& operator << (ostream& output){}
};
```

Apare însă imediat o problemă de sintaxă: deoarece este funcție membru, operatorul este apelat prin intermediul unui obiect, **o.operator<<(cout)**, ceea ce însemnă că ar trebui să scriem exact invers decât în mod obişnuit:

```
//dacă << este supraîncărcat prin funcție membru
T o;
o << cout;          //!!!!!
```

Aceeaşi problemă apare şi pentru >>; de aceea cei doi operatori se supraîncarcă prin funcții globale. Nu uitați să returnați fluxul primit ca argument pentru a permite afişările/citirile înlănțuite! Ca de obicei, feriți-vă de a declara cei doi operatori funcții friend. O tehnică uzuală pentru a evita acest lucru constă în dotarea clasei cu o funcție membru virtuală publică (numită, deseori, *print()*) responsabilă cu afişarea unei reprezentări a clasei. Avantajul imediat este că orice clasă derivată trebuie doar să supraîncarce această funcție:

```
//exemplul 15
#include <iostream>
using namespace std;

class base{
        int a;
        public:
        base(int x) : a(x){}
        virtual void print(ostream& output) const{
                output << "base: " << a <<  endl;
        }
};

ostream& operator<<(ostream& out_stream, const base& x){
        x.print(out_stream);
        return out_stream;
}

class derived: public base{
        int aa;
        public:
        derived (int x, int y): base(x), aa(y){}
        void print(ostream& output) const{
                //extindem versiunea din clasa de bază
                base::print(output);
                output << "derived: " << aa << endl;
        }
};
```

```
int main(){
        base o1(5);
        cout << o1;
        derived o2(10,10);
        cout << o2;         //nu mai trebuie supraîncărcat << pentru derived!
}
```

10. Operatori de conversie

Se ştie că compilatorul tratează orice constructor cu un singur argument ca pe un *operator implicit de conversie*, pe care îl utilizează pentru a rezolva apeluri de funcţii:

```
//exemplul 16
class T{
        public:
        T(int);
};
void f(T);
f(1);               //eroare de compilare? NU: f(T(1))
```

Nu uitaţi că conversiile standard au prioritate faţă de conversiile utilizator:

```
//exemplul 17
class T{
        public:
        T(int);
};
void f(T);
void f(double);
f(1);               //f(T(1)) sau f(double(1))? f(double(1))
```

De asemenea, nu uitaţi că operatorii supraîncărcaţi de utilizator trebuie să aibă cel puţin un operand de un tip utilizator:

```
//exemplul 18
class T{
        public:
        T(int);
};
const T operator+ (const T&, const T&);
int x, y;
T r1 = x + y;       //adunare de întregi!
T r2 = T(x) + y;    //adunare de T!
```

Un constructor nu poate însă specifica:

- o conversie implicită de la un tip utilizator la un tip de bază (tipurile predefinite nu sunt clase);
- o conversie de la o clasă nouă la una deja definită (fără a modifica definiţia vechii clase).

O funcţie membru T::operator X(), unde X este un tip, defineşte un operator de conversie de la T la X. Tipul X (spre care se converteşte) este parte a numelui

operatorului şi nu poate fi repetat ca valoare de retur. De exemplu, declarația unui operator de conversie către int arată astfel:

int T::operator int() const; //eroare de compilare

T::operator int() const; //OK

Se recomandă ca aceşti operatori de conversie să fie introduşi în mod strict necesar; definiți în exces, ei pot conduce la ambiguități:

```
//exemplul 19
class T{
        public:
        T(int);
        operator int() const;
};
const T operator+ (const T&, const T&);
T t;
int i;
t + i;      //ambiguitate: int(t) + i sau operator+(t, T(i))?
        //nu uitați că tipul returnat nu se ia în calcul la rezoluția supraîncărcării!
```

Încheiem acest subiect cu următoarea observație: într-o secvență de conversii implicite este permisă o singură conversie utilizator:

```
//exemplul 20
class X{
        public:
        X(int);
        X(char*);
};
class Y{
        public:
        Y(int);
};
class Z{
        public:
        Z(X);
};
X f(X);
Y f(Y);
Z g(Z);
f(1);      //ambiguitate: f(X(1)) sau f(Y(1))?
g(X("supraîncărcarea"));   //OK, g( Z( X("supraîncărcarea"))); o singură conversie utilizator implicită: din X în Z
g(Z("operatorilor"));      //OK, g( Z( X("operatorilor"))); o singură conversie utilizator implicită: din char* în X
g("supraîncărcarea operatorilor");   //eroare de compilare;
                                     //două conversii utilizator implicite: din char* în X şi din X în Z!

//exemplul 21
class T1{
    public:
    operator int(){return 3;}
};

class T2{
    public:
    T2(int){}
};
```

```
void f(T2){}
...
T1 x;
f(x);       //eroare de compilare;
            // două conversii utilizator implicite: din T1 în int şi din int în T2!
```

11. Operatorul de indexare []

Este folosit atunci când se doreşte ca un obiect al unei clase să poată fi utilizat asemănător cu un tablou unidimensional. Se supraîncarcă întotdeauna prin funcție membru şi are doar un singur argument (un index, care poate fi de orice tip). (Dacă doriți să simulați tablouri multidimensionale, trebuie să supraîncărcați operatorul apel de funcție:().) Nu uitați să vă dotați clasa şi cu *versiunea const* a funcției operator[] (care va fi aplicată obiectelor const)!

```
//exemplul 21
#include <cstdlib>
#include <iostream>
using namespace std;

class int_array{
            size_t d;
            int* t;
public:
            int_array(size_t dd): d(dd){
                        t = new int[d];
            }
            int& operator[](size_t d){
                        cout << "operator[]" << endl;
                        return t[d];
            }
            const int& operator[](size_t d) const{                  //versiunea const
                        cout << "operator[] const" << endl;
                        return t[d];
            }
};

void print(const int_array& t){
            cout << t[0] << endl;
}

int main(){
            size_t dim = static_cast<size_t>(10);
            int_array tablou(dim);
            tablou[0] = 5;                          //operator[]
            cout << tablou[0] << endl;              //operator[]
            print(tablou);                          //operator[] const!
}
```

Acest exemplu demonstrează necesitatea versiunii const a funcției operator[]; desigur că, doar pentru a simula tablouri a căror dimensiune se stabileşte dinamic (la

execuție) şi nu static (la compilare), nu trebuie să reinventați roata! Folosiți cu încredere containerele din STL, precum vector.

12. Operatorul apel de funcție ()

Acest operator, supraîncărcat prin funcție membru, este unic în C++, în sensul că acceptă oricâte argumente, de orice tip (şi chiar acceptă valori implicite pentru argumente!); de aceea, uneori, el este supraîncărcat ca operator de indexare pentru tablouri multidimensionale. Utilizarea lui uzuală este însă alta: el generează *obiecte care se comportă precum funcțiile*, obiecte numite functori. În general, e vorba despre tipuri care au o singură operație sau pentru care o operație este predominantă.

Următorul exemplu prezintă cum se poate executa o operație asupra fiecărui membru al unui vector:

```
//exemplul 22
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

class T{
public:
	int val;
	T(int x = 0): val(x){}
	operator int(){
		return val;
	}
};

void neg(T& t){
    t.val = -t.val;
}

void aduna_10(T& t){
    t.val += 10;
}

int main(){
	vector<T> tablou;
	for (int i=1; i<=10; ++i)
		tablou.push_back(T(i));
	for (int i=0; i<10; ++i)
		cout << tablou[i] << " ";
	cout << endl;
	for_each(tablou.begin(), tablou.end(), neg);
	for_each(tablou.begin(), tablou.end(), aduna_10);
	for (int i=0; i<10; ++i)
		cout << tablou[i] << " ";
	cout << endl;
}
```

Cum am putea să adunăm la fiecare membru al vectorului o valoare arbitrară?

```
//exemplul 23
class aduna{                                        //functor
        T de_adunat;
public:
        aduna(T x): de_adunat(x){}
        aduna(int x): de_adunat(T(x)){}
        void operator()(T& t) const {               //operația dominantă
                t.val += de_adunat.val;
        }
};

for_each(tablou.begin(), tablou.end(), aduna(100));
for (int i=0; i<10; ++i)
        cout << tablou[i] << " ";
cout << endl;

for_each(tablou.begin(), tablou.end(), aduna(T(1)));
for (int i=0; i<10; ++i)
        cout << tablou[i] << " ";
cout << endl;
```

adună(100) (respectiv, aduna(T(1))) este un obiect construit o singură dată; funcția membru operator()() va fi cea care se va apela repetat, pentru fiecare componentă a vectorului.

Desigur, se poate utiliza şi o funcție obişnuită care să aibă un argument suplimentar (valoarea arbitrară ce trebuie să o adunăm). Ce se intâmplă dacă funcția trebuie să adune la componenta curentă o valoare care depinde şi de factori externi (de exemplu, de componenta anterioară)? Funcția trebuie să-şi păstreze starea între apeluri şi poate face asta doar prin intermediul variabilelor locale **statice**. Există însă o singură copie a variabilelor locale statice, ceea ce face ca apelurile simultane ale funcției să partajeze starea, lucru evident de nedorit în medii cu mai multe fire de execuție!

Functorii sunt însă obiecte şi îşi conservă starea internă; mai mult, chiar dacă au fost inițializate cu aceeaşi informație, două obiecte functori sunt distincte şi deci au stări interne distincte.

*13. Operatorii de selecție a membrului -> şi dereferențiere **

Operatorii **->** şi ***** sunt supraîncărcați, de obicei împreună, pentru a proiecta obiecte care să se comporte ca nişte pointeri şi care să ofere, suplimentar, funcționalitate extinsă. Aceste obiecte sunt cunoscute în literatura de specialitate sub

numele de smart pointers; şi librăria standard C++ defineşte un şablon de pointer inteligent, şabon numit auto_ptr<>.

Dacă Ptr_to_X este o clasă de obiecte pointer către tipul X, atunci operatorul de dereferențiere * supraîncărcat în Ptr_to_X execută funcționaliatea suplimentară care se doreşte a fi adăugată unui pointer şi returnează obiectul referit. Deşi compilatorul vă permite să returnați prin valoare obiectul referit, **trebuie** să returnați prin referință! De ce? Dacă *p* este de tip *T**, **p* este o *l-value* de tip *T*! Mai mult, un obiect Ptr_to_X poate să refere nu doar obiecte de tipul X, ci şi obiecte de un tip derivat din X! Dacă returnați prin valoare, Ptr_to_X nu va oferi suport pentru apelarea funcțiilor virtuale definite în X. În concluzie, prototipul funcției operator operator*() arată astfel:

X& Ptr_to_X::operator();*

```
//exemplul 24
#include <iostream>
using namespace std;

class X{
public:
        int data;
        void function() const{
                cout << "function member "    << endl;
        };
        virtual void virtual_function() const{
                cout << "I'm an X object" << endl;
        };
        X(int d=0): data(d){}
};

class Y: public X{
public:
        Y(int d=0): X(d){}
        void virtual_function() const{
                cout << "I'm an Y object" << endl;
        };
};

class Ptr_to_X{
        X* adr;                         //pointer la obiectul referit
public:
        Ptr_to_X(X* a=0): adr(a){}      //garantează inițializarea cu 0
        X& operator*(){                 //returnați prin referință!
                //execută functionalitate suplimentară, apoi
                return *adr;
        }
};

int main(){
   X o1(5);
   Y o2(10);
   Ptr_to_X p1(&o1), p2(&o2);
```

```
    cout << (*p1).data << endl;        //parantezele necesare căci operatorul . de selecție a membrului este prioritar!
    (*p1).function();
    (*p1).virtual_function();
    cout << endl;

    cout << (*p2).data << endl;
    (*p2).function();
    (*p2).virtual_function();
     cout << endl;

     //Ptr_to_X p3;                    //NULL
     //*p3;                            //???
     return 0;
}
```

Ce se întâmplă dacă se dereferențiază un obiect pointer null? Deoarece, în general, dereferențierea unui pointer null generează comportament nedefinit, aveți libertatea de a trata o astfel de situație în orice mod doriți.

Operatorul -> de selecție a membrului trebuie supraîncărcat printr-o funcție operator membră a clasei, care să returneze ceva pentru care să fie legal să se apeleze operatorul -> global! Din acest motiv, ea va returna pointerul către obiectul referit, ceea ce oferă şi suport pentru apelarea funcțiilor virtuale definite în X; prototipul acestei funcții operator este:

X Ptr_to_X::operator->() const;*

```
//exemplul 25
#include <iostream>
using namespace std;

class X{
public:
        int data;
        void function() const{
                cout << "function member "    << endl;
        };
        virtual void virtual_function() const{
                cout << "I'm an X object" << endl;
        };
        X(int d=0): data(d){}
};

class Y: public X{
public:
        Y(int d=0): X(d){}
        void virtual_function() const{
                cout << "I'm an Y object" << endl;
        };
};

class Ptr_to_X{
    X* adr;
public:
    Ptr_to_X(X* a=0): adr(a){}
```

```
    X& operator*(){
      return *adr;
    }

    X* operator->() const{
      return adr;
    }
};

int main(){
   X o1(5);
   Y o2(10);
   Ptr_to_X p1(&o1);
   Ptr_to_X p2(&o2);

   cout << p1->data << endl;          //(p1.operator->())->data
   p1->function();                    //(p1.operator->())->function()
   p1->virtual_function();            //(p1.operator->())->virtual_function()
   cout << endl;

   //X* q1 = p1->;                    //eroare de sintaxă!
   //X* q1 = p1.operator->();         //OK!

   cout << p2->data << endl;
   p2->function();
   p2->virtual_function();
   return 0;
}
```

Se observă că funcția operator operator->() returnează un pointer la X indiferent de membrul selectat; în acest sens, **->** este un **operator unar** în formă postfixată. Nu s-a modificat însă sintaxa, ceea ce face necesară prezența unui membru după ->! Evident, funcția operator poate fi apelată explicit.

Pentru pointerii obişnuiți, (p->m) == ((*p).m) == p[0].m; dacă se doreşte o astfel de echivalență şi pentru clasele utilizator, ea trebuie furnizată explicit.

Observație: C++ permite crearea, prin supraîncărcarea lui **->** şi **->***, a unor obiecte smart pointer către X; dereferențiind un astfel de obiect, "ar trebui" să obținem un obiect "smart reference" către X. Această simetrie nu este însă realizabilă datorită faptului că **.** şi **.*** nu pot fi supraîncărcați!

*14. Operatorul de selecție prin pointer la membru ->**

Un pointer la membru identifică un membru al unei clase şi se obține aplicând operatorul de adresă & unui nume complet calificat de membru. Pointerii la funcții membru sunt utili atunci când (analog pointerilor la funcții) trebuie să apelăm o funcție membru fără a-i cunoaşte numele.

Dacă un pointer la funcție este o adresă de memorie, un pointer la funcție membru virtuală este un index (sau un offset); cu alte cuvinte, el nu depinde de localizarea obiectului în memorie! În consecință, un pointer la funcție membru virtuală poate fi transferat între spații de adrese diferite (evident, cu condiția ca acestea să folosească acelaşi model-obiect)! Pointerii la funcții şi pointerii la funcții membru nevirtuale nu pot fi transferați între spații de adrese.

Un pointer la membru poate fi utilizat doar în combinație cu un obiect; operatorii **.*** şi **->*** permit programatorului să exprime această compinație. Dacă *pm* este un pointer la membru, *o* este un obiect şi *p* este un pointer la *o*, atunci *o*.**pm* leagă *pm* de obiectul *o*, în timp ce *p*->**pm* leagă *pm* de obiectul referit de *p*. Un membru static nu este asociat cu un obiect anume, deci pointerii la membri statici sunt pointeri obişnuiți.

```
//exemplul 26
#include <iostream>
using namespace std;

class X{
public:
        int data;
        void function() const{
                cout << "function member "    << endl;
        };
        virtual void virtual_function() const{
                cout << "I'm an X object" << endl;
        };
        X(int d=0): data(d){}
};

typedef void (X::* PFM)()const;

class Y: public X{
public:
        Y(int d=0): X(d){}
        void virtual_function() const{
                cout << "I'm an Y object" << endl;
        };
};

int main(){
   X o1(5);
   Y o2(10);
   PFM pf1 = &X::function;
   PFM pf2 = &X::virtual_function;
   X *p1=&o1, *p2=&o2;              //p1 şi p2 sunt pointeri built-in la X
   (p1->*pf1)();                    //observați utilizarea operatorului apel de funcție ()
   (p1->*pf2)();
   (p2->*pf1)();
   (p2->*pf2)();
}
```

Ca şi ->, operatorul ->* se supraîncarcă prin funcție membru şi este utilizat pentru o simula comportamentul pointer-la-membru pentru obiectele smart pointer; ->* este însă un operator binar. Cheia implementării funcției *operator* ->*() constă în observația că această funcție **trebuie** să returneze un obiect pentru care să poată fi apelată funcția *operator()*() cu argumentele funcției membru ce trebuie apelată; o clasă auxiliară ne va ajuta să definim acest obiect.

```
//exemplul 27
#include <iostream>
using namespace std;

class X{
public:
	int data;
	void function() const{
		cout << "function member "    << endl;
	};
	virtual void virtual_function() const{
		cout << "I'm an X object" << endl;
	};
	X(int d=0): data(d){}
};

class Y: public X{
public:
	Y(int d=0): X(d){}
	void virtual_function() const{
		cout << "I'm an Y object" << endl;
	};
};
typedef void (X::* PFM)()const;     //alias pt. pointer la funcție membru const, fără argumente şi care nu returnează

class Ptr_to_X{
	X* adr;
public:
	Ptr_to_X(X* a=0): adr(a){}
	X& operator*(){
		return *adr;
	}
	X* operator->() const{
		return adr;
	}
protected:
	//operator ->*() trebuie sa returneze un obiect pt. care este definit operator()()
	class auxiliar{
		X* p;
		PFM pfm;
	public:
		auxiliar(X* const p1, PFM p2):p(p1), pfm(p2){}
		auxiliar(const auxiliar& other){
			p = other.p;
			pfm = other.pfm;
		}
		void operator()() const{
			(p->*pfm)();      //apelul propriu-zis al functiei membru
		}
	};      //sfârşit definiție pt. clasa auxiliar
```

```
public:
        auxiliar operator->*(PFM pfm){          //supraîncărcarea ->*
                return auxiliar(adr, pfm);      //apelare automata a lui auxiliar::operator()()
        }
};

int main(){
   X o1(5);
   Y o2(10);

   Ptr_to_X p1(&o1);                            //p1 şi p2 sunt acum obiecte smart pointer la X
   Ptr_to_X p2(&o2);

   PFM pf1 = &X::function;
   PFM pf2 = &X::virtual_function;

   (p1->*pf1)();                                //aceeaşi sintaxă cu cea a pointerilor built-in
   (p1->*pf2)();

   (p2->*pf1)();
   (p2->*pf2)();
   return 0;
}
```

Am supraîncărcat ->* doar pentru funcții membru fără argumente şi care nu returnează; cu ajutorul template-urilor se poate generaliza acest mecanism.

Norocel PETRACHE

Bibliografie:

1. Bjarne Stroustrup, *The C++ Programming Language*
2. *ANSI/ISO C++ Professional Programmer's Handbook*
3. Scott Meyers, *Effective C++*
4. Scott Meyers, *More Effective C++*
5. Herb Sutter – *Exceptional C++*
6. Bruce Eckel – *Thinking in C++*