

LUCRARE DE LABORATOR NR. 3

Tema: Supraîncărcarea operatorilor

Scopul lucrării:

- Studiarea necesității supraîncărcării operatorilor;
- Studiarea sintaxei de definire a operatorilor;
- Studiarea tipurilor de operatori;
- Studiarea formelor de supraîncărcare;

Noțiuni de bază

Avantajul utilizării operatorilor (reducerea codului) în același timp complică înțelegerea codului, deoarece nu este posibil întotdeauna sau este complicat să se urmărească dacă se utilizează un operator predefinit sau supraîncărcat. Însă supraîncărcarea operatorilor este necesară. Sunt probleme care se soluționează numai prin supraîncărcarea operatorilor, cum ar fi operatorul de atribuire. Desigur, compilatorul poate să genereze codul necesar, sau să definim metoda, de exemplu *Assign*(Java). Dar ambele soluții nu sunt ideale.

Neajunsul primei metode constă în utilizarea copierii bit cu bit, ceea ce este satisfăcător atît timp cît în clasă nu se utilizează pointerii:

```
class Book{
    char* name;
public:
    Book(char n){
        name = new char[strlen(n)+1];
        strcpy(name,n);
    }
    ~Book(){
        delete[] name;
    }
    void print(){
        cout<<name;
    }
};

void main(){
    Book b("Stroustrup"), b2("Lippman");
    b2 = b;
    b.print();
    b2.print();
}
```

Exemplul dat ilustrează problema legată de pointeri. În primul rînd, după atribuire memoria alocată anterior (care conține mesajul *Lippman*), nu a fost eliberată, și ambele obiecte referă aceeași locație de memorie (modificarea unui obiect duce și la modificarea celui alt). Altă problemă și mai esențială – la apelul destructorului se va încerca eliberarea memoriei deja dezalocate, ceea ce poate duce la erori de sistem.

La utilizarea metodelor special predestinate atribuirii nu se poate de garantat utilizarea lor în toate cazurile, înafară de aceasta contrazicem ideea că clasele trebuie să se comporte ca și tipurile predefinite de date. De asemenea pot apare ambiguități, deoarece se va utiliza operatorul de atribuire pentru lucrul cu pointerii și funcția pentru utilizarea obiectelor însuși.

Din cele expuse mai sus, concluzia este că cel puțin un operator este strict necesar.

Definire și utilizare

Ce este, de fapt, un operator? Operatorul este o funcție cu un nume predefinit și o sintaxă specială la apelare. Operatorii se definesc cu ajutorul cuvântului cheie *operator*, urmat de simbolul operației. În rest, este o funcție obișnuită, care are parametri și returnează un rezultat.

```
Complex operator+(const Complex& r){  
    return Complex(re+r.re,im+r.im);  
}
```

În continuare, operatorul dat poate fi utilizat ca și un operator predefinit, dar se poate de utilizat și forma funcțională de apelare:

```
Complex c1,c2,c3;  
c3=c1+c2;           // utilizarea operatorului  
c3=c1.operator+(c2); // utilizarea funcției
```

Tipurile de operatori

Toți operatorii se divizează în două grupuri: unari și binari. Operatorii unari se numesc operatorii cu un singur operand. De ex., operatorul & - de referențiere, ++ - de incrementare, ! - de negare ș. a. Operatorii binari au doi operanzi: + - adunarea, * - înmulțirea ș. a.

Unii operatori unari au două forme. De exemplu, operatorul ++ poate fi scris ca *i++*, sau *++i*. Cum deosebește compilatorul care realizare să apeleze? În acest scop a fost introdusă regula: operatorul postfix are un parametru suplimentar de tip întreg:

```
Complex& operator++();           // forma prefixă  
Complex operator++(int);         // forma postfixă
```

Forme de supraîncărcare

Există două forme de supraîncărcare: ca funcții membru și ca funcții prieten. Operatorul prieten se definește conform regulilor caracteristice funcțiilor prieten. De ce oare se utilizează ambele forme de supraîncărcare? Nu este oare suficientă supraîncărcarea cu ajutorul metodelor clasei? De ce este nevoie ca operatorul să fie prieten al clasei? Pentru a răspunde la această întrebare, trebuie de menționat că operatorul << întotdeauna se definește ca prieten. Motivul constă în modalitatea de apelare a funcțiilor membru. Practic întotdeauna se transmite obiectul, utilizând implicit pointerul *this*. La transferul altor date se utilizează parametri obișnuiți. Altfel spus, primul operand trebuie să fie neapărat reprezentant al clasei date. În cazul operatorului <<, primul operand este de tip *ostream*, și nu de tip utilizator. Adunarea unui număr complex cu un întreg poate fi realizată atât prin funcție membru, cât și prin funcție prieten, însă adunarea unui întreg cu un număr complex – numai prin funcții prieten. În acest caz, dacă se schimbă termenii cu locul, rezultatul *se schimbă*.

Specificul operatorilor

Limbajul C++ permite supraîncărcarea numai a operatorilor existenți în limbaj. Dintre aceștia nu pot fi supraîncărcați operatorii:

\".", \".*\", \"?:\", \"::\", \"sizeof\", \"#\", \"##\".

Însă pot fi supraîncărcați operatorii:

`"->", "[]", "()", "new" и "new[]", "delete" и "delete[]"` ș. a.

Trebuie de menționat că operatorii de atribuire "=", de indexare "[]", de apel funcție"()" și operatorul "->" pot fi definiți numai ca funcții membru.

Un specific aparte îl are operatorul "->", la supraîncărcarea lui variabila se comportă ca și un pointer, ceea ce induce tipul pointerilor inteligenți. Operatorul de apel funcție se utilizează pentru definirea obiectelor funcționale.

Note

- Pentru tipurile predefinite nu pot fi definiți operatori, dar ei pot fi utilizați;
- Unul și același operator nu poate fi definit și ca funcție membru, și ca funcție prieten;
- Toți operatorii trebuie să returneze un tip diferit de *void*, excepție fiind operatorul ();
- La supraîncărcare prioritatea operatorilor nu se schimbă;
- Operatorul () poate avea un număr arbitrar de parametri, el nu face parte din clasificarea obișnuită a operatorilor.

Întrebări de control:

1. Este oare supraîncărcarea operatorilor absolut necesară?
2. Ce cuvinte cheie se utilizează pentru definirea operatorilor?
3. Pot oare operatorii să nu întoarcă niciun rezultat?
4. Cum se clasifică operatorii?
5. Cum deosebește compilatorul forma prefixă și cea postfixă a operatorilor unari?
6. Care operatori nu pot fi supraîncărcați?
7. Care este sintaxa de apel a operatorilor?
8. Cum se utilizează operatorul "()"?
9. Cum se utilizează operatorul "->"?
10. De ce sunt necesare două forme de supraîncărcare (ca funcții membru și ca funcții prieten)?
11. Cărui tip aparține operatorul "<<" pentru ieșiri de obiecte?
12. În care caz este necesar să se definească operatorul de atribuire?

Sarcina

Varianta 1

a) Să se creeze o clasă de numere întregi. Să se definească operatorii "++" și "+", ca metode ale clasei, iar operatorii "- -" și "-" ca funcții prietene. Operatorii trebuie să permită efectuarea operațiilor atât cu variabilele clasei date, cât și cu variabilele întregi de tip predefinit.

b) Să se creeze o clasă *Set* – mulțimea numerelor întregi, utilizând memoria dinamică. Să se definească operatorii de lucru cu mulțimile: "+" – uniunea, "*" – intersecția, "-" scăderea, ca funcții prietene, iar "+=" – înserarea unui nou element în mulțime, "==" – comparare la egalitate, ș. a. ca metode ale clasei. Să se definească operatorii "<<" și ">>". Să se definească funcția de verificare a apartenenței unui element la o mulțime.

Varianta 2

a) Să se creeze clasa 2-D de coordonate în plan. Să se definească operatorii "+" și "-" ca funcții prietene, iar operatorii de atribuire și de comparare – ca metode ale clasei. De prevăzut posibilitatea efectuării operațiilor atât între coordonate, cât și între coordonate și numere obișnuite.

b) Să se creeze clasa *Stack* – stivă, utilizând memoria dinamică. Să se definească următorii operatori ca metode ale clasei: "+" – de adunare a două stive, "=" – de atribuire, "()" – de returnare a unei stive noi care conține ultimele n elemente ale stivei date. Să se definească operatorii de comparare: "==", "!=", "<", ">" ca funcții prietene. Pentru realizarea ultimilor doi operatori să se definească funcția de calcul a normei elementelor stivei. Să se definească operatorii "<<" și ">>" pentru ieșiri/intrări de obiecte. Clasa trebuie să fie absolut funcțională, adică să conțină toți constructorii necesari și destructorul.

Varianta 3

a) Să se creeze clasa 3-D de coordonate în spațiu. Să se definească operatorii "+", "-", "=", ca metode ale clasei, iar operatorii de comparare – ca funcții prietene. De prevăzut posibilitatea realizării de operații atât între coordonate cât și între coordonate și numere obișnuite.

b) Să se creeze clasa *Queue* –coadă, utilizând memoria dinamică. Să se definească operatorii "+" – de adunare a două cozi, "=" – de atribuire ca metode ale clasei. Să se definească operatorii de comparare "==", "!=", "<", ">" ca funcții prietene. Pentru realizarea ultimilor doi operatori să se definească funcția de calcul a normei elementelor cozii. Să se definească operatorii "<<" și ">>" pentru ieșiri/intrări de obiecte, precum și pentru inserarea/eliminarea elementelor în/din coadă.

Varianta 4

a) Să se creeze clasa *Date* – data, care conține câmpurile: ziua, luna, anul. Să se definească operatorii "+" și "-" ca metode ale clasei, iar operatorii "++" și "--" în ambele variante(prefixă și postfixă) ca funcții prietene. Operatorul "+" trebuie să permită realizarea operațiilor numai cu variabilele de tip predefinit *int* ($x=y+5$). De prevăzut prelucrarea corectă a anilor bisecți.

b) Să se creeze clasa *List* – coadă. Să se definească operatorii "+" – de adunare a listelor, "=" – de atribuire ca metode ale clasei. Să se definească operatorii de comparare "==", "!=", "<", ">" ca funcții prietene. Să se supraîncarce operatorii "<<" și ">>" pentru ieșiri/intrări de obiecte, precum și pentru inserarea/eliminarea elementelor în/din coadă. Clasa trebuie să fie absolut funcțională, adică să conțină toți constructorii necesari și destructorul. Pentru simplificarea lucrului să se utilizeze clasa sau structura *ListItem* pentru reprezentarea elementelor listei spre care referă *List*.

Varianta 5

a) Să se creeze clasa numerelor reale. Să se definească operatorii "++" și "+" ca metode ale clasei, iar operatorii "- -" și "-" – ca funcții prietene. Operatorii trebuie să permită realizarea operațiilor atât cu variabilele clasei date, cât și cu variabilele de tip predefinit *double*.

b) Să se creeze clasa *Vector* – vector, utilizând memoria dinamică. Să se definească operatorii "+" – adunarea element cu element a vectorilor, "-" – scăderea element cu element a vectorilor, "=" – atribuirea ca metode ale clasei. Să se definească operatorii de comparare: "==", "!=", "<", ">" ca funcții prietene. Pentru realizarea ultimilor doi operatori să se definească funcția de

calcul a modulului elementelor vectorului. Să se supraîncarce operatorii "<<" și ">>" pentru ieșiri/intrări de obiecte. Clasa trebuie să conțină toți constructorii necesari și destructorul.

Varianta 6

a) Să se creeze clasa numerelor mari întregi. Să se definească operatorii "+" și "*" ca metode ale clasei, iar "-" și "/" - ca funcții prietene. Să se supraîncarce operatorii de incrementare și de decrementare în ambele forme (prefixă și postfixă). Operatorii trebuie să permită realizarea operațiilor atât cu variabilele clasei date, cât și cu variabilele de tip predefinit *long*.

b) Să se creeze clasa *Matrix* – matrice, utilizând memoria dinamică. Să se definească operatorii "+" – adunarea matricilor, "-" – scăderea matricilor, "=" – atribuirea ca metode ale clasei. Să se definească operatorii de comparare: "==", "!=", "<", ">" ca funcții prietene. Pentru realizarea ultimilor doi operatori să se definească funcția de calcul a normei elementelor matricei. Să se definească operatorul "[]" astfel încât să fie cu sens adresarea [][] la elemente. Să se supraîncarce operatorii "<<" și ">>" pentru ieșiri/intrări de obiecte.

Varianta 7

a) Să se creeze clasa *Bool* – variabile logice. Să se definească operatorii "+" – SAU logic, "*" – ȘI logic, "^" – SAU EXCLUSIV, ca metode ale clasei, iar operatorii "==" și "!=" – ca funcții prietene. Operatorii trebuie să permită realizarea operațiilor atât cu variabilele clasei date, cât și cu variabilele de tip predefinit *int*. (Dacă numărul întreg este diferit de zero, se consideră că variabila este adevăr, altfel – fals.)

b) Să se creeze clasa *String* – șir, utilizând memoria dinamică. Să se definească operatorii "+" – adunarea șirurilor, "=" și "+=" – atribuirea ca metode ale clasei. Să se definească operatorii de comparare: "==", "!=", "<", ">" ca funcții prietene. Operatorii trebuie să lucreze atât cu *String*, cât și cu *char**. Să se definească operatorul "[" de acces la fiecare simbol în parte. Să se supraîncarce operatorii "<<" și ">>" pentru ieșiri/intrări de obiecte.

Varianta 8

a) Să se creeze clasa *Complex* – numere complexe. Să se definească toți operatorii dintr-un simbol ca funcții prietene, iar operatorii ce constau din două simboluri – ca metode ale clasei. Excepție – operatorul de atribuire, care poate fi numai metodă a clasei și operatorii pentru ieșiri/intrări de obiecte. Adunarea și scăderea trebuie realizată atât cu numere complexe cât și cu tipul predefinit *double*.

b) Să se creeze clasa *Stack* – stivă, utilizând memoria dinamică. Să se definească următorii operatori ca metode ale clasei: "+" – de adunare a două stive, "=" – de atribuire, "()" – de returnare a unei stive noi care conține ultimele n elemente ale stivei date. Să se definească operatorii de comparare: "==", "!=", "<", ">" ca funcții prietene. Pentru realizarea ultimilor doi operatori să se definească funcția de calcul a normei elementelor stivei. Să se definească operatorii "<<" și ">>" pentru ieșiri/intrări de obiecte. Clasa trebuie să fie absolut funcțională, adică să conțină toți constructorii necesari și destructorul

Varianta 9

a) Să se creeze clasa *Time* – timpul, care conține câmpurile: ore, minute, secunde. Să se definească operatorii "+" și "-", ca funcții prietene, iar "++" și "--" în ambele forme (prefixă și

postfixă) ca metode ale clasei. Operatorii trebuie să permită realizarea operațiilor atât cu variabilele clasei date, cât și cu variabilele de tip predefinit *int* (desemnează secunde).

b) Să se creeze clasa *Queue* – coadă, utilizând memoria dinamică. Să se definească operatorii "+" – de adunare a două cozi, "=" – de atribuire ca metode ale clasei. Să se definească operatorii de comparare "==", "!=", "<", ">" ca funcții prietene. Pentru realizarea ultimilor doi operatori să se definească funcția de calcul a normei elementelor cozii. Să se definească operatorii "<<" și ">>" pentru ieșiri/intrări de obiecte, precum și pentru inserarea/eliminarea elementelor în/din coadă.

Varianta 10

a) Să se creeze clasa *Bool* – variabile logice. Să se definească operatorii "+" – SAU logic, "*" – ȘI logic, "^" – SAU EXCLUSIV, ca funcții prietene, iar operatorii "==" și "!=" – ca metode ale clasei. Operatorii trebuie să permită realizarea operațiilor atât cu variabilele clasei date, cât și cu variabilele de tip predefinit *int*. (Dacă numărul întreg este diferit de zero, se consideră că variabila este adevăr, altfel – fals.)

b) Să se creeze o clasă *Set* – mulțimea numerelor întregi, utilizând memoria dinamică. Să se definească operatorii de lucru cu mulțimile: "+" – uniunea, "*" – intersecția, "-" scăderea, ca metode ale clasei, iar "+=" – înserarea unui nou element în mulțime, "==" – comparare la egalitate, ș. a. ca funcții prietene. Să se definească operatorii "<<" și ">>". Să se definească funcția de verificare a apartenenței unui element la o mulțime.

Varianta 11

a) Să se creeze clasa *Date* – data, care conține câmpurile: ziua, luna, anul. Să se definească operatorii "+" și "-" ca metode ale clasei, iar operatorii "++" și "--" în ambele variante(prefixă și postfixă) ca funcții prietene. Operatorul "+" trebuie să permită realizarea operațiilor numai cu variabilele de tip predefinit *int* ($x=y+5$). De prevăzut prelucrarea corectă a anilor bisecți.

b) Să se creeze clasa *String* – șir, utilizând memoria dinamică. Să se definească operatorii "+" – adunarea șirurilor, "=" și "+=" – atribuirea ca funcții prietene. Să se definească operatorii de comparare: "==", "!=", "<", ">" ca metode ale clasei. Operatorii trebuie să lucreze atât cu *String*, cât și cu *char**. Să se definească operatorul "[]" de acces la fiecare simbol în parte. Să se supraîncarce operatorii "<<" și ">>" pentru ieșiri/intrări de obiecte.

Varianta 12

a) Să se creeze clasa 2-D de coordonate în plan. Să se definească operatorii "+" și "-" ca funcții prietene, iar operatorii de atribuire și de comparare – ca metode ale clasei. De prevăzut posibilitatea efectuării operațiilor atât între coordonate, cât și între coordonate și numere obișnuite.

b) Să se creeze clasa *List* – coadă. Să se definească operatorii "+" – de adunare a listelor, "-" – de scădere(ca la mulțimi) ca funcții prietene. Să se definească operatorii de comparare "==", "!=", "<", ">" ca metode ale clasei. Să se supraîncarce operatorii "<<" și ">>" pentru ieșiri/intrări de obiecte, precum și pentru inserarea/eliminarea elementelor în/din coadă. Clasa trebuie să fie absolut funcțională, adică să conțină toți constructorii necesari și destructorul. Pentru simplificarea lucrului să se utilizeze clasa sau structura *ListItem* pentru reprezentarea elementelor listei spre care referă *List*.