LUCRARE DE LABORATOR NR. 2

Tema: Supraîncărcarea operatorilor

Scopul lucrării:

- Studierea necesităţii supraîncărcării operatorilor;
- Studierea sintaxei de definire a operatorilor;
- Studierea tipurilor de operatori;
- Studierea formelor de supraîncărcare;

Noţiuni de bază

Avantajul utilizării operatorilor (reducerea codului) în acelaşi timp complică înţelegerea 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 soluţionează 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 soluţii 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 conţine mesajul *Lippman*), nu a fost eliberată, şi ambele obiecte referă aceeaşi locaţie de memorie (modificarea unui obiect duce şi la modificarea celuilalt). Altă problemă şi mai esenţială – 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 şi tipurile predefinite de date. De asemenea pot apare ambiguităţi, deoarece se va utiliza operatorul de atribuire pentru lucrul cu pointerii şi funcţia pentru utilizarea obiectelor însuşi.

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

Definire şi utilizare

Ce este, de fapt, un operator? Operatorul este o funcţie cu un nume predefinit şi o sintaxă specială la apelare. Operatorii se definesc cu ajutorul cuvîntului cheie *operator,* urmat de simbolul operaţiei. În rest, este o funcţie obişnuită, care are parametri şi returnează un rezultat.

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

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

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

Tipurile de operatori

Toţi operatorii se divizează în două grupuri: unari şi binari. Operatorii unari se numesc operatorii cu un singur operand. De ex., operatorul & - de referenţiere, ++ - de incrementare, ! – de negare ş. a. Operatorii binari au doi operanzi: + - adunarea, * - înmulţirea ş. a.

Unii operatori unari au două forme. De exemplu, operatorul ++ poate fi scris ca *i*++, sau ++*i*. Cum deosebeşte 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 funcţii membru şi ca funcţii prieten. Operatorul prieten se defineşte conform regulilor caracteristice funcţiilor 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 menţionat că operatorul << întotdeauna se defineşte ca prieten. Motivul constă în modalitatea de apelare a funcţiilor membru. Practic întotdeauna se transmite obiectul, utilizînd implicit pointerul *this*. La transferul altor date se utilizează parametri obişnuiţi. Altfel spus, primul operand trebuie să fie neapărat reprezentant al clasei date. În cazul operatorului <<, primul operand este de tip *ostream,* şi nu de tip utilizator. Adunarea unui număr complex cu un întreg poate fi realizată atît prin funcţie membru, cît şi prin funcţie prieten, însă adunarea unui întreg cu un număr complex – numai prin funcţii prieten. În acest caz, dacă se schimbă termenii cu locul, rezultatul *se schimbă*.

Specificul operatorilor

Limbajul C++ permite supraîncărcarea numai a operatorilor existenţi în limbaj. Dintre aceştia nu pot fi supraîncărcaţi operatorii:

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

Însă pot fi supraîncărcaţi operatorii:

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

Trebuie de menţionat că operatorii de atribuire "=", de indexare "[]", de apel funcţie"()" şi operatorul "->" pot fi definiţi numai ca funcţii membru.

Un specific aparte îl are operatorul ->, la supraîncărcarea lui variabila se comportă ca şi un pointer, ceea ce induce tipul pointerilor inteligenţi. Operatorul de apel funcţie se utilizează pentru definirea obiectelor funcţionale.

Note

- Pentru tipurile predefinite nu pot fi definiţi operatori, dar ei pot fi utilizaţi;
- Unul şi acelaşi operator nu poate fi definit şi ca funcţie membru, şi ca funcţie prieten;
- Toţi operatorii trebuie să returneze un tip diferit de *void,* excepţie fiind operatorul ();
- La supraîncărcare prioritatea operatorilor nu se schimbă;
- Operatorul () poate avea un număr arbitrar de parametri, el nu face parte din clasificarea obişnuită 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 deosebeşte compilatorul forma prefixă şi cea postfixă a operatorilor unari?
6. Care operatori nu pot fi supraîncărcaţi?
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 funcţii membru şi ca funcţii prieten)?
11. Cărui tip aparţine operatorul "<<" pentru ieşiri de obiecte?
12. În care caz este necesar să se definească operatorul de atribuire?

Sarcina

Varianta 1

Să se creeze o clasă de numere întregi *Int*. Să se definească operatorii "++" şi "+", ca metode ale clasei, iar operatorii "--" şi "-" ca funcţii prietene. Operatorii trebuie să permită efectuarea operaţiilor atît cu variabilele clasei date, cît şi cu variabilele întregi de tip predefinit *int*. Să se definească operatorii "<<" şi ">>".

Varianta 2

Să se creeze clasa *Vector* – vector de tip *long*, utilizînd memoria dinamică. Să se definească operatorii "+" – adunarea element cu element a vectorilor, "-" – scăderea element cu element a vectorilor, ca funcţii prietene. Să se definească "=" – operator de atribuire și operatorii de comparare: "==", "!=", "<", ">" ca metode ale clasei. Pentru realizarea ultimilor doi operatori să se definească funcţia de calcul a modulului elementelor vectorului. Să se supraîncarce operatorii "<<" şi ">>" pentru ieşiri/intrări de obiecte. Clasa trebuie să conţină toţi constructorii necesari şi destructorul.

Varianta 3

Să se creeze clasa *2-D de coordonate* de tip *int* în plan. Să se definească operatorii "+" şi "-" ca funcţii prietene, iar operatorii de atribuire şi de comparare – ca metode ale clasei. De prevăzut posibilitatea efectuării operaţiilor atît între coordonate, cît şi între coordonate şi numere obişnuite. Să se definească operatorii "<<" şi ">>".

Varianta 4

Să se creeze clasa *3-D de coordonate* de tip *int* în spaţiu. Să se definească operatorii "+", "-" și operator de atribuire ca metode ale clasei, iar operatorii de comparare– ca funcţii prietene. De prevăzut posibilitatea realizării de operaţii atît între coordonate cît şi între coordonate şi numere obişnuite. Să se definească operatorii "<<" şi ">>".

Varianta 5

Să se creeze clasa *Fraction* – fracția rațională. Să se definească toţi operatorii dintr-un simbol ca funcţii prietene, iar operatorii ce constau din două simboluri – ca metode ale clasei. Excepţie – operatorul de atribuire, care poate fi numai metodă a clasei şi operatorii pentru ieşiri/intrări de obiecte ca funcţii prietene. Adunarea şi scăderea trebuie realizată atît cu fracții cît şi cu tipul predefinit *int*.

Varianta 6

Să se creeze clasa numerelor reale *Double*. Să se definească operatorii "++" şi "+" ca metode ale clasei, iar operatorii "- -" şi "-" – ca funcţii prietene. Operatorii trebuie să permită realizarea

operaţiilor atît cu variabilele clasei date, cît şi cu variabilele de tip predefinit *double*. Să se definească operatorii "<<" şi ">>".

Varianta 7

Să se creeze clasa numerelor mari întregi *Long*. Să se definească operatorii "+" şi "*" ca metode ale clasei, iar "-" şi "/" - ca funcţii prietene. Să se supraîncarce operatorii de incrementare şi de decrementare în ambele forme (prefixă şi postfixă). Operatorii trebuie să permită realizarea operaţiilor atît cu variabilele clasei date, cît şi cu variabilele de tip predefinit *long*. Să se definească operatorii "<<" şi ">>".

Varianta 8

Să se creeze clasa *Bool* – variabile logice. Să se definească operatorii "+" – SAU logic, "*" – ŞI logic, "^" – SAU EXCLUSIV, ca metode ale clasei, iar operatorii "==" şi "!=" – ca funcţii prietene. Operatorii trebuie să permită realizarea operaţiilor atît cu variabilele clasei date, cît şi cu variabilele de tip predefinit *int*. (Dacă numărul întreg este diferit de zero, se consideră că variabila este adevăr, altfel – fals). Să se definească operatorii "<<" şi ">>".

Varianta 9

Să se creeze clasa *Complex* – numere complexe. Să se definească toţi operatorii dintr-un simbol ca metode ale clasei, iar operatorii ce constau din două simboluri – ca funcţii prietene. Excepţie – operatorul de atribuire, care poate fi numai metodă a clasei şi operatorii pentru ieşiri/intrări de obiecte ca funcţii prietene. Adunarea şi scăderea trebuie realizată atît cu numere complexe cît şi cu tipul predefinit *double*.

Varianta 10

Să se creeze clasa *Vector* – vector de tip *float*, utilizînd memoria dinamică. Să se definească operatorii "+" – adunarea element cu element a vectorilor, "-" – scăderea element cu element a vectorilor, ca funcţii prietene. Să se definească "=" – operator de atribuire și operatorii de comparare: "==", "!=", "<", ">" ca metode ale clasei. Pentru realizarea ultimilor doi operatori să se definească funcţia de calcul a modulului elementelor vectorului. Să se supraîncarce operatorii "<<" şi ">>" pentru ieşiri/intrări de obiecte. Clasa trebuie să conţină toţi constructorii necesari şi destructorul.

Varianta 11

Să se creeze clasa *3-D de coordonate* de tip *float* în spaţiu. Să se definească operatorii "+", "-", ca funcţii prietene, iar operatorii de comparare și "="– ca metode ale clasei. De prevăzut posibilitatea realizării de operaţii atît între coordonate cît şi între coordonate şi numere obişnuite. Să se definească operatorii "<<" şi ">>".

Varianta 12

Să se creeze clasa *Bool* – variabile logice. Să se definească operatorii "+" – SAU logic, "*" – ŞI logic, "^" – SAU EXCLUSIV, ca funcţii prietene, iar operatorii "==" şi "!=" – ca metode ale clasei. Operatorii trebuie să permită realizarea operaţiilor atît cu variabilele clasei date, cît şi cu variabilele de tip predefinit *int*. (Dacă numărul întreg este diferit de zero, se consideră că variabila este adevăr, altfel – fals). Să se definească operatorii "<<" şi ">>".

Varianta 13

Să se creeze clasa *Complex* – numere complexe. Să se definească toţi operatorii dintr-un simbol ca funcţii prietene, iar operatorii ce constau din două simboluri – ca metode ale clasei. Excepţie – operatorul de atribuire, care poate fi numai metodă a clasei şi operatorii pentru ieşiri/intrări de obiecte ca funcţii prietene. Adunarea şi scăderea trebuie realizată atît cu numere complexe cît şi cu tipul predefinit *double*.

Varianta 14

Să se creeze o clasă *Set* – mulţimea numerelor întregi, utilizînd memoria dinamică. Să se definească operatorii de lucru cu mulţimile: "+" – uniunea, "*" – intersecţia, "-" scăderea, ca funcţii prietene, iar "+=" – înserarea unui nou element în mulţime, "==" – comparare la egalitate, ş. a. ca metode ale clasei. Să se definească operatorii "<<" şi ">>". Să se definească funcţia de verificare a apartenenţei unui element la o mulţime.

Varianta 15

Să se creeze clasa *String* – şir, utilizînd memoria dinamică. Să se definească operatorii "+" – adunarea şirurilor, "=" şi "+=" – atribuirea ca metode ale clasei. Să se definească operatorii de comparare: "==", "!=", "<", ">" ca funcţii prietene. Operatorii trebuie să lucreze atît cu *String*, cît şi cu *char**. Să se definească operatorul "*[]*" de acces la fiecare simbol în parte. Să se supraîncarce operatorii "<<" şi ">>" pentru ieşiri/intrări de obiecte.

Varianta 16

Să se creeze clasa *2-D de coordonate* de tip *float* în plan. Să se definească operatorii "=", "+" şi "-" ca metode ale clasei, iar operatorii de comparare – ca funcţii prietene. De prevăzut posibilitatea efectuării operaţiilor atît între coordonate, cît şi între coordonate şi numere obişnuite. Să se definească operatorii "<<" şi ">>".

Varianta 17

Să se creeze clasa *Vector* – vector de tip *double*, utilizînd memoria dinamică. Să se definească operatorii "+" – adunarea element cu element a vectorilor, "-" – scăderea element cu element a vectorilor și "=" – atribuirea, ca metode ale clasei. Să se definească operatorii de comparare: "==", "!=", "<", ">" ca funcţii prietene. Pentru realizarea ultimilor doi operatori să se definească funcţia de calcul a modulului elementelor vectorului. Să se supraîncarce operatorii

"<<" şi ">>" pentru ieşiri/intrări de obiecte. Clasa trebuie să conţină toţi constructorii necesari şi destructorul.

Varianta 18

Să se creeze clasa *Set* – mulţimea numerelor întregi, utilizînd memoria dinamică. Să se definească operatorii de lucru cu mulţimile: "+" – uniunea, "*" – intersecţia, "-" scăderea, ca metode ale clasei, iar "+=" – înserarea unui nou element în mulţime, "==" – comparare la egalitate, ş. a. ca funcţii prietene. Să se definească operatorii "<<" şi ">>". Să se definească funcţia de verificare a apartenenţei unui element la o mulţime.

Varianta 19

Să se creeze clasa *String* – şir, utilizînd memoria dinamică. Să se definească operatorii "+" – adunarea şirurilor ca funcţii prietene. Să se definească operatorii de comparare: "==", "!=", "<", ">" şi "+=" – atribuirea ca metode ale clasei. Operatorii trebuie să lucreze atît cu *String*, cît şi cu *char**. Să se definească operatorul "*[]*" de acces la fiecare simbol în parte. Să se supraîncarce operatorii "<<" şi ">>" pentru ieşiri/intrări de obiecte.

Varianta 20

Să se creeze clasa *Vector* – vector de tip *int*, utilizînd memoria dinamică. Să se definească operatorii "+" – adunarea element cu element a vectorilor, "-" – scăderea element cu element a vectorilor, ca funcţii prietene. Să se definească "=" – operator de atribuire și operatorii de comparare: "==", "!=", "<", ">" ca metode ale clasei. Pentru realizarea ultimilor doi operatori să se definească funcţia de calcul a modulului elementelor vectorului. Să se supraîncarce operatorii "<<" şi ">>" pentru ieşiri/intrări de obiecte. Clasa trebuie să conţină toţi constructorii necesari şi destructorul.

Varianta 21

Să se creeze clasa *2-D de coordonate* de tip *double* în plan. Să se definească operatorii "+" şi "-" ca funcţii prietene, iar operatorii de atribuire şi de comparare – ca metode ale clasei. De prevăzut posibilitatea efectuării operaţiilor atît între coordonate, cît şi între coordonate şi numere obişnuite. Să se definească operatorii "<<" şi ">>".

Varianta 22

Să se creeze clasa numerelor reale *Float*. Să se definească operatorii "++" şi "+" ca funcţii prietene metode ale clasei, iar operatorii "- -" şi "-" – ca metode ale clasei. Operatorii trebuie să permită realizarea operaţiilor atît cu variabilele clasei date, cît şi cu variabilele de tip predefinit *double*. Să se definească operatorii "<<" şi ">>".