Facultatea de Automatică și Calculatoare

Universitatea Politehnica Timișoara

Programare orientată pe obiecte

Curs 5

Pointerul *this*

Când este apelată o funcţie membru i se pasează automat un argument implicit (care nu apare în lista de parametri) care este un pointer către obiectul care a generat apelul : **this** **(Program19)**

```
class putere {
  double b;
  int e;
  double rez;
public:
  putere(double, int);
  double rezultat() { return rez; }
};

putere::putere(double baza, int exp)
{
  b = baza;
  e = exp;
  rez = 1;
  if (exp == 0) return;
  for (; exp > 0; exp--) rez = rez * b;
}
```

în care :

membrii unei clase pot fi accesaţi direct fără specificator de obiect sau clasă:

b=baza;

(din interiorul funcţiei putere() determină ca valoarea conţinută în baza să fie atribuită unei copii a lui b asociată obiectului care a generat apelarea)

aceeaşi instrucţiune poate fi scrisă şi astfel :

this ->b=baza;

funcţia putere() ar trebui să arate astfel :

```
putere::putere(double baza, int exp)
{
        this->b=baza;
        this->e=exp;
        this->val=1;
        if (exp==0) return;
        for (; exp>0; exp--)
                this->rez=this->rez * this->b;
}
```

Pointerul *this* – continuare 1

- Operatorul this este foarte important la supraîncărcarea operatorilor şi ori de câte ori o funcţie membră trebuie să utilizeze un pointer către obiectul care a apelat-o
- Pointerul this este transmis automat către toate funcţiile membre :

```
double rezultat() {return this->rez; }
```

iar la apelul

```
y.rezultat();
```

pointerul **this** va indica spre obiectul **y.**

- ***Restricţii*** legate de **this** :
  - Funcţiile **friend** nu sunt membri ai clasei, deci nu le sunt pasaţi pointeri **this.**
  - Funcţiile membre de tip **static** nu au nici ele un pointer **this.**

Referințe

- O *referinţă* este un pointer implicit, care acţionează ca un **alt nume** al unui obiect.
- **utilizare** - crearea funcţiilor care folosesc transmiterea parametrului prin referinţă şi nu metoda implicită din **C++**, apelarea prin valoare

  exemplu **Program24 - modalitatea ne-automatizată**

Programul25 - modalitatea automatizată

```
void neg(int *);

int main()
{
  int in;
  in = 100;
  cout << in << " este negat  ";
  neg(&in);// se transmite adresa explicit
  cout << in << "\n";
  return 0;
}

void neg(int *i)
{
   *i = -*i;
}
```

```
100 este negat  -100
```

```
void neg(int &i);// i este referinta

int main()
{
int in;
in = 100;
cout << in << " este negat  ";
neg(in);// nu e nevoie de &
cout << in << "\n";
_getch();
return 0;
}

void neg(int &i)
{
i = -i;// i este referinta deci nu e nevoie de *
}
```

```
100 este negat  -100
```

Referințe – continuare 1

- Nu este posibilă modificarea în interiorul funcţiei a adresei spre care “indică” parametrul (deci în funcţia **neg(.)** nu se poate introduce spre exemplu o instrucţiune : **`i++; )`**
- Alt exemplu **Program26**

```
void schimb(int &, int &);

int main()
{
  int a, b;
  a = 1;
  b = 2;

  cout << " a si b: " << a << " " << b << " ";
  schimb(a, b);// aici schimbam ; vedem ca nu trebuie
               //     scris cu &
  cout << " a si b: " << a << " " << b << " ";
  return 0;
}

void schimb(int &i, int &j)
{
  int temp;

  temp = i;// nu trebuie sa lucram cu *
  i = j;
  j = temp;
 }
```

```
a si b: 1 2  a si b: 2 1
```

Transmiterea referinţelor către obiecte, returnarea referinţelor

- Atunci când un obiect este transmis unei funcţii ca argument, se face automat o copie a acelui obiect şi nici nu e apelat constructorul obişnuit al clase în schimb, la terminarea funcţiei, este apelat destructorul copiei.
- Dacă nu se doreşte apelarea destructorului, se poate realiza transmiterea obiectului prin referinţă (la acest gen de apel, aşa cum ne şi aşteptăm, nu se face o copie a obiectului ci manipularea datelor are loc chiar în obiectul transmis ca referinţă)
- exemplu **Program27**

```
class nu_distruge {
int k;
public:
int i;
nu_distruge(int);
~nu_distruge();
void neg(nu_distruge &ob) { ob.i = -ob.i; } // nu se va
//   crea obiect temporar
};
nu_distruge::nu_distruge(int valinit) // constructorul
{
  cout << "Construieste " << valinit << "\n";
  k = valinit;
}
nu_distruge::~nu_distruge()// destructorul
{
  cout << "Distruge " << k << "\n";
}
```

```
int main()
{
  nu_distruge ob(1);// un obiect initializat

  ob.i = 10;
  ob.neg(ob);    // la apel se transmite obiectul prin
   // referinta

  cout << ob.i << "\n";

   return 0;
}
```

```
Construieste 1
-10
Distruge 1
```

Transmiterea referințelor către obiecte, returnarea referințelor - continuare 1

Returnarea referinţelor

- O funcţie poate să returneze o referinţă. Ca urmare, paradoxal, o asemenea funcţie poate să apară în membrul stâng al unei instrucţiuni de atribuire exemplu **Program28**

```
char &inlocuire(int);// returneaza o referinta

char s[30] = "Salutare tuturor studentilor";

int main()
{
  inlocuire(2) = '*';// se suprascrie un X pe a 4-a
  //      pozitie in sir
  cout << s;
  _getch();
  return 0;
}

char &inlocuire(int i)
{
   return s[i];
}
```

```
SaXutare tuturor studentilor
```

Referințe independente

- pot fi declarate şi referinţe ca simple variabile - *referinţe independente*
- la crearea unei referinţe independente, tot ceea ce se creează este de fapt un al doilea nume pentru o variabilă.
- Toate variabilele de tip referinţă independentă trebuie iniţializate la creare
- exemplu **Program29**
- Se poate folosi o referinţă independentă chiar şi pentru constante. Spre exemplu, **`int &max_val=100;`**

```
int main()
{
  int a;
  int &ref = a;// aceasta e o referinta independenta

  a = 1;
  cout << a << " " << ref << "\n";

  ref = 10;
  cout << a << " " << ref << "\n";

  int b = 39;
  ref = b;// aici, valoarea lui b trece de fapt in a
  cout << a << " " << ref << "\n";
  ref--;// decrementeaza pe a in realitate
  cout << a << " " << ref << "\n";
  return 0;
}
```

```
1 1
10 10
39 39
38 38
```

Restricții referitoare la referințe

- O referinţă nu se poate referi la altă referinţă (adică nu poate fi obţinută adresa unei referinţe).
- Nu pot fi create matrice de referinţe.
- Nu se poate crea un pointer spre o referinţă.
- O referinţă nu se poate defini pentru un câmp de biţi.
- O variabilă de tip referinţă trebuie iniţializată la declarare, dar nu trebuie iniţializată dacă este membru al unei clase, parametru de funcţie sau valoare returnată.
- Referinţele nule nu sunt permise.

- ***Observaţie*** :
  - Unii programatori adoptă nişte convenţii legate de asocierea operatorilor * sau &. Astfel :

```
int& p;       // asociat cu tipul
int &p;       // asociat cu variabila
```

  - cele două declaraţii sunt echivalente funcţional dar asocierea cu numele tipului reflectă dorinţa programatorilor ca limbajul **C++** să pună la dispoziţie un pointer distinct pentru tip. Dar sintaxa din **C++**nu presupune distributivitatea operatorilor într-o listă de parametri. De aceea se pot uşor formula declaraţii greşite. Astfel :

```
int* a,b;
```

creează un pointer de tip întreg, nu doi, aşa cum am fi dorit.

Operatorii de alocare dinamică din C++

- Operatorul **new** returnează un pointer către memoria alocată
  - Ca şi **malloc()**, **new** alocă memorie din zona *heap* (zona de memorie liberă).
  - Dacă nu există memorie suficientă pentru alocare, se returnează un pointer nul.
- Operatorul **delete** eliberează memoria alocată anterior prin utilizarea operatorului **new**.
- Formele generale de utilizare ale celor doi operatori sunt:
- exemplu **Program30**

```
p_var=new tip;     // p_var de tip pointer
...
delete p_var;
```

Observaţie : *Operatorul **delete** trebuie să fie folosit doar cu un pointer valid, alocat deja prin* **new.** *Folosirea oricărui alt tip de pointer cu **delete** duce la un rezultat imprevizibil şi aproape sigur determină probleme, chiar căderea sistemului.*

```
int main()
{
  int *p;
  p = new int;// aloca spatiu pentru un intreg
  if (!p) {
     cout << "Eroare de alocare\n";
     exit(1);
  }
 *p = 190;

 cout << "La " << p << " ";
 cout << " este valoarea " << *p << "\n";

 delete p;
 return 0;
 }
```

```
La 0085D548  este valoarea 190
```

Operatorii de alocare dinamică din C++ - continuare 1

- **Avantaje:**
  - **new** alocă automat memorie suficientă pentru păstrarea obiectelor de tipul specificat (deci nu trebuie determinată mărimea obiectului cu **sizeof**). Astfel, se elimină orice posibilitate de eroare în privinţa spaţiului de alocare.
  - **new** returnează automat un pointer de tipul specificat, nefiind necesară utilizarea unui modelator de tip, aşa cum trebuia făcut cu **malloc()**.
  - Atât **new** cât şi **delete** pot fi *supraîncărcaţi*, permiţând crearea unui sistem de alocare propriu.
  - Prin utilizarea **new** memoria poate fi iniţializată cu o valoare cunoscută, scriind în instrucţiunea cu **new** o valoare după numele tipului :

p_var=new tip_var (initializator)

exemplu **Program31**

```
La 001C6108  este valoarea 314
```

```
int main()
{
  int *p;
  p = new int(314);// aloca spatiu pentru un intreg
  //  dar si initializeaza
   if (!p) {
       cout << "Eroare de alocare\n";
       exit(1);
    }
  //*p = 190;
  cout << "La " << p << " ";
  cout << " este valoarea " << *p << "\n";
  delete p;
  return 0;
}
```

Operatorii de alocare dinamică din C++ - continuare 2

- **Pentru matrice :**

  p_var = new tip_matrice[marime];

  ...

  delete [] p_var;

  exemplu **Program32**

Trebuie precizat că matricele alocate astfel nu pot primi valori inițiale.
Deci în astfel de situații nu se poate specifica o inițializare, așa cum s-a făcut pentru variabilele distincte.

```
int main()
{
  int *p, i;// (ati retinut ca doar primul e pointer ?)

  p = new int[10];// se aloca memorie pentru 10 intregi
  if (!p) {
      cout << "Eroare de alocare\n";
     exit(1);
   }
   for (i = 0; i<10; i++) p[i] = i;
   for (i = 0; i<10; i++) cout << p[i] << " ";

    delete[] p;   // aici eliberam memoria ocupata de cei 10 intregi
    return 0;
}
```

```
0 1 2 3 4 5 6 7 8 9
```

Alocare de memorie pentru obiecte

- Obiectelor li se poate aloca memorie dinamic, folosind **new**.
- Când se face această alocare, se creează un obiect şi se returnează un pointer către el.
- Obiectul creat dinamic se comportă ca oricare alt obiect.
- Când este creat, este apelat *constructorul*, dacă există.
- Atunci când este apelat **delete** (deci se eliberează memoria ocupată), se execută funcţia *destructor*, dacă există.

  exemplu **Program33**

- Obiectele alocate dinamic pot avea constructori şi destructori, constructorii pot fi parametrizaţi, exemplu **Program34**

Capitolul 5. Supraîncărcarea

Supraîncărcarea funcţiilor şi a operatorilor (*overloading*) oferă posibilitatea realizării ***polimorfismului*** şi adaugă limbajului flexibilitate şi extensibilitate.

b **Supraîncărcarea funcţiilor**

- reprezintă folosirea aceluiaşi nume pentru două sau mai multe funcţii.
- Fiecare redefinire a funcţiei trebuie să folosească **tipuri diferite de parametri de apel sau număr diferit de parametri**
- exemplu **Program1** pentru **tipuri** diferite de parametri

```
int supra(int);// prima functie are un parametru intreg
double supra(double);// iar a doua are parametru double

int main()
{
  cout << supra(19) << " ";// apelul primei functii
  cout << supra(5.4) << "\n";//apelul celeilalte functii
  return 0;
}
```

```
// acum sint definite cele doua functii
int supra(int i)
{
    return i;
}

double supra(double d)
{
    return d;
}
```

```
19 5.4
```

exemplu **Program2** pentru **număr** diferit de parametri

```
int supra1(int);// un parametru
int supra1(int, int);// doi parametri (aceeasi denumire a
                     // functiei)

int main()
{
  cout << supra1(11) << " ";// apelul primei functii
  cout << supra1(5, 4) << "\n";//apelul celeilalte functii
return 0;
}
```

```
// acum sunt definite cele doua functii
int supra1(int i)
{
   return i;
}

int supra1(int i, int j)
{
  return i*j;
}
```

```
11 20
```

- nu pot fi supraîncărcate două funcţii care diferă doar prin tipul returnat:

```
int supra2(int );
float supra2(int );      // eroare, difera doar tipul returnat
```

- Trebuie mare atenţie la situaţiile care par să definească o supraîncărcare:

```
void supra4(int *p);  // *p este acelasi lucru cu p[],
void supra4(int p[]); //    deci aici e eroare
```

Ambiguități și anacronisme la supraîncărcarea funcțiilor

- **Sunt situaţii în care compilatorul nu poate discerne între funcţiile supraîncărcate - situaţie *ambiguă*.**
- **Aceste situaţii sunt depistate de compilator şi semnalate ca erori fatale şi programul nu este compilat.**
- **Principala cauză de ambiguităţi este conversia automată de tip pe care o realizează C++: compilatorul C++ încearcă să convertească automat argumentele folosite pentru apelul de funcţii în tipurile argumentelor aşteptate de funcţiile respective. De aceea, fragmentul următor de program nu este eronat :**

```
int exemplu(double );
...
cout << exemplu('x');        // nu e eroare, desi la apel nu a
                             // fost transmisa o variabila double
```

Ambiguități și anacronisme la supraîncărcarea funcțiilor – continuare 1

- exemplu de ambiguitate Program3 **(**la primul apel lucrurile sunt clare deoarece numărul care reprezintă parametrul de apel va fi considerat în lipsă de alte indicaţii, de tip **double -** în **C++** orice constantă în virgulă mobilă este implicit **double** - dar la al doilea apel nu se va şti în ce trebuie convertit întregul - în **float** sau în **double**)

```
float ambiguu(float);
double ambiguu(double);

int main()
{
  cout << ambiguu(22.34) << " "; // apel neambiguu (pentru
                                 //     double)
   cout << ambiguu(33) << "\n "; // ambiguu (in ce va fi
                                 //   convertit intregul ?)
 return 0;
}
```

```
float ambiguu(float f)
{
  return f;
}
double ambiguu(double d)
{
  return -d;
}
```

Ambiguități și anacronisme la supraîncărcarea funcțiilor – continuare 2

- exemplu de ambiguitate Program4 (compilatorul nu va şti în ce anume să convertească valoarea întreagă, deşi char şi unsigned char nu sunt de la sine ambigue în C++)

```
char ambiguu1(unsigned char);
char ambiguu1(char);

int main()
{
cout << ambiguu1('x') << " "; // apel neambiguu (pentru
                              //     caracter)
cout << ambiguu1(55) << "\n "; // ambiguu (in ce va fi
                              //   convertit intregul ?)

return 0;
}
```

```
char ambiguu1(unsigned char uc)
{
   return uc - 1;
}
char ambiguu1(char sc)
{
   return sc + 1;
}
```

Ambiguități și anacronisme la supraîncărcarea funcțiilor – continuare 2

b exemplu de ambiguitate **Program5** (Primul apel nu este ambiguu deoarece s-au precizat clar doi parametri, dar la al doilea apel compilatorul nu are de unde să știe dacă trebuie să apeleze versiunea cu un parametru sau cea cu al doilea parametru **implicit**.)

```
int ambiguu2(int i);
int ambiguu2(int i, int j = 1);

int main()
{
 cout << ambiguu2(4, 5) << " ";// apel neambiguu
 cout << ambiguu2(199) << "\n";// ambiguu (nu se stie
// daca e cu sau fara al doilea parametru)

return 0;
}
```

```
int ambiguu2(int i)
{
  return i;
}
int ambiguu2(int i, int j)
{
  return i*j;
}
```

Ambiguități și anacronisme la supraîncărcarea funcțiilor – continuare 4

- exemplu de ambiguitate **Program6** (două funcţii nu pot fi supraîncărcate dacă diferă doar prin modul de preluare a parametrilor. Într-adevăr, la scrierea apelului unei funcţii nu este nici o diferenţă între apelarea prin referinţă şi apelarea (implicită) prin valoare).

Anacronism :

overload nume_functie

```
void ambiguu3(int);
void ambiguu3(int &);// deja e eroare (de ce ?)

int main()
{
   int i = 129;

ambiguu3(i);// eroare, nu se stie care ambiguu3
return 0;
}
```

```
void ambiguu3(int i)
{
cout << "Aici sintem in ambiguu3
cu apel prin valoare\n";
}
void ambiguu3(int &i)
{
cout << "Aici sintem in ambiguu3 cu
apel prin referinta\n";
}
```

Supraîncărcarea funcțiilor constructor

- Ca oricare altă funcţie, *constructorii* pot fi şi aceştia supraîncărcaţi

- exemplu **Program7**

```
class date {
   int zi, luna, an;
public:
  date(char *);
  date(int, int, int);
  void arata_data();
};

// constructor de initializare cu sir
//(forma zz/ll/aa)
date::date(char *s)
{
  sscanf_s(s, "%d%*c%d%*c%d", &zi, &luna, &an);
}
// constructor de initializare cu intregi
//(zi, luna, an)
date::date(int z, int l, int a)
{
  zi = z;
  luna = l;
  an = a;
}
```

```
void date::arata_data()

{
  cout << zi << "/" << luna << "/" << an << "\n";
}

int main()
{
  date d1(4, 3, 98), d2("10/04/98");

  d1.arata_data();
  d2.arata_data();

  return 0;
}
```

```
4/3/98
10/4/98
```

Supraîncărcarea funcțiilor constructor – continuare 1

De obicei, supraîncărcarea unui constructor este folosită pentru a iniţializa un obiect în modul cel mai natural şi convenabil cu putinţă (aşa cum a fost şi în exemplul precedent : dacă nu ar fi existat cele două forme de constructor, nu ar fi fost posibilă iniţializarea "naturală" a datei sub formă de şir de caractere sau iniţializarea cerută de reprezentarea folosită în cadrul clasei, sub formă de întregi).

Exemplu Program8

```
class date {
  int zi, luna, an;
public:
  date(char *);
  date(int, int, int);
  void arata_data();
};

// constructor de initializare cu sir (forma zz/ll/aa)
date::date(char *s)
{
  sscanf_s(s, "%d%*c%d%*c%d", &zi, &luna, &an);
}
// constructor de initializare cu intregi (zi, luna, an)
date::date(int z, int l, int a)
{
  zi = z;
  luna = l;
  an = a;
}
```

```
void date::arata_data()
{
  cout << zi << "/" << luna << "/" << an << "\n";
}

int main()
{
  char s[10];// aici va fi introdusa data
  cout << "Introduceti data :";
  cin >> s;// aici este operatia de intrare pentru data
  date d(s);// se creeaza un obiect si se
            // initializeaza cu sir de caractere
  d.arata_data();
  return 0;
}
```

Această flexibilitate devine foarte importantă atunci cînd dorim să realizăm ierarhii de clase pentru a fi refolosite în alte aplicaţii. În această situaţie, evident trebuie să ne asigurăm că viitorul utilizator are la dispoziţie cît mai multe facilităţi, corespunzătoare situaţiilor pe care le poate întîlni şi necesităţilor care pot să apară în diverse aplicaţii

Adresa unei funcții supraîncărcate

- În C putem să atribuim adresa unei funcţii, unui pointer şi apoi să apelăm funcţia folosind pointerul. La fel se poate proceda (s-a văzut în cursurile precedente) şi în **C++**. O complicaţie survine la supraîncărcarea funcţiilor, deoarece în această situaţie doar declararea corespunzătoare a pointerului poate face diferenţierea.
- exemplu **Program9**

```
int func_sup(int);
int func_sup(int, int);

int main()
{
 int(*pf)(int i, int j); // pointer catre functia cu un param.

 pf = func_sup;// indica spre prima functie

 cout << pf(5,4);// aici utilizam functia
 return 0;
}
```

```
int func_sup(int i)
{
  return i;
}
int func_sup(int i, int j)
{
  return i*j;
}
```

Dacă declaraţia ar fi : **int (*pf)(int i, int j);** atunci funcţia apelată (evident, cu doi parametri) ar fi fost cea de-a doua versiune a funcţiei supraîncărcate.

b Declararea funcţiei de tip pointer trebuie deci să corespundă exact uneia şi numai uneia dintre declarările funcţiei redefinite (supraîncărcate).

Supraîncărcarea operatorilor

- În **C++** pot fi supraîncărcaţi aproape toţi operatorii.
- Supraîncărcarea acestora este strâns legată de supraîncărcarea funcţiilor.
- În **C++** supraîncărcarea operatorilor poate să determine efectuarea unor operaţii speciale relativ la clasele create.
- Operatorii se supraîncarcă folosind funcţiile **operator**.
- O astfel de funcţie defineşte operaţiile specifice pe care le va efectua operatorul supraîncărcat relativ la clasa în care este destinat să lucreze.
- Funcţiile **operator** pot sau nu să fie membre ale clasei în care vor opera.
- De obicei, când funcţiile **operator** nu sunt membre, sunt măcar funcţii **friend**.
- Cele două situaţii (cînd sunt membre, respectiv când sunt **friend**) trebuie tratate diferit.

Funcțiile operator membre

- Forma generică a definirii unor astfel de funcţii este :

```
tip_returnat nume_clasa::operator#(lista_argumente)
{
  ... // aici vor fi operatiile care reprezinta redefinirea
        // operatorului
}
```

cu # înlocuit cu operatorul dorit

Funcțiile operator membre – continuare 1

- exemplu **Program1**

```
class loc {
  int longit, latit;
public:
  loc() {}  // necesar pentru constructii de obiecte
              //temporare
  loc(int lg, int lt) {
    longit = lg;
    latit = lt;
  }

  void arata() {
    cout << longit << " ";
    cout << latit << "\n";
  }

   loc operator+(loc ot);
   };
```

Trebuie neapărat reţinut că la folosirea operatorilor binari supraîncărcaţi, obiectul din stânga generează apelarea funcţiei operator (şi deci va transmite un pointer this).

```
loc loc::operator+(loc ot) // supraincarcarea operatorului +
{
   loc tmp;// folosim un obiect temporar

   tmp.longit = ot.longit + longit; // aici sint operatiile
   tmp.latit = ot.latit + latit;

   return tmp;// obiectul este returnat
}

int main()
{
   loc ob1(10, 20), ob2(15, 25);// se creeaza doua obiecte
                                //  initializate

   ob1.arata();// se va afisa 10  20
   ob2.arata();// se va afisa 15  25

   ob1 = ob1 + ob2;// aici se foloseste supraincarcarea
   ob1.arata();// se va afisa 25  45
   return 0;
}
```

Operatorul + are doar un parametru şi nu doi, cum ne-am aştepta (+ fiind un operator binar) deoarece operandul din stânga operatorului + (în programul principal, nu în definirea fumcţiei operator !) este pasat implicit folosind pointerul this, iar operandul din dreapta este pasat prin parametrul ob2 (care corespunde actualizării parametrului formal de tip obiect loc, din definirea funcţiei operator).

Funcțiile operator membre – continuare 2

- **Este un lucru uzual ca o funcţie operator supraîncărcată să returneze un obiect din clasa asupra căreia operează. Câteodată, o neatenţie în acest sens poate duce la erori. Astfel, dacă funcţia operator+() returnează altceva decât un obiect de tipul obiectelor din expresia de mai jos, expresia nu e validă :**

```
ob1=ob1+ob2;
```

- **Rezultatul operaţiei trebuie să fie un obiect de acelaşi tip. Se permite chiar şi următoarea sintaxă :**

```
(ob1+ob2).arata();   // se afiseaza rezultatul ob1+ob2
```

- **În exemplul de mai sus, se generează un obiect temporar în care se reţine rezultatul operaţiei supraîncărcate şi acest obiect este distrus după afişare (după apelul funcţiei-membru arata()).**

- **Funcţia operator nu modifică nici unul dintre operanzi (proprietate “naturală” şi logică).**

Funcțiile operator membre – continuare 3

- exemplu **Program2**

```
class loc {
  int longit, latit;
public:
  loc() {}   // necesar pentru constructii de obiecte
              //    temporare
  loc(int lg, int lt) {
     longit = lg;
     latit = lt;
   }
  void arata() {
   cout << longit << " ";
   cout << latit << "\n";
}
  loc operator+(loc ot);
  loc operator-(loc ot);
  loc operator=(loc ot);
  loc operator++();
};
loc loc::operator+(loc ot) // supraincarcarea operatorului +
{
  loc tmp;// folosim un obiect temporar
  tmp.longit = ot.longit + longit; // aici sint operatiile
  tmp.latit = ot.latit + latit;
  return tmp;// obiectul este returnat
}
```

Funcțiile operator membre – continuare 4

- exemplu **Program 2 – continuare 1**

```
loc loc::operator-(loc ot) // supraincarcarea operatorului -
{
  loc tmp;// folosim un obiect temporar
          //atentie la ordinea operanzilor
  tmp.longit = longit - ot.longit; // aici sint operatiile
  tmp.latit = latit - ot.latit;
  return tmp;// obiectul este returnat
}
loc loc::operator=(loc ot) // supraincarcarea operatorului =
{
  loc tmp;// folosim un obiect temporar
  longit = ot.longit; // aici sint operatiile
  latit = ot.latit;
   return *this;// aici este returnat chiar obiectul
                // care a generat apelarea
}
// Atentie ! urmeaza un operator unar (nu are parametri -
oare - de ce ?)
loc loc::operator++() // supraincarcarea operatorului ++
{
  // nu mai folosim un obiect temporar
  longit++; // aici sint operatiile
  latit++;
  return *this;// de asemenea este returnat chiar
   //   obiectul care a generat apelarea, de fapt a operat
   // direct asupra obiectului
}
```

La operandul -, a trebuit să fim atenţi la ordine : operandul din dreapta semnului trebuie scăzut din operandul din stânga (ordinea naturală) şi ţinând cont de operandul care generează apelul funcţiei operator-() (adică acel operand care va transmite un pointer this), a rezultat exprimarea aparent curioasă din definirea funcţiei operand respective.

În lipsa unei supraîncărcări, operatorul = devine un operator de copiere membru cu membru. În exemplul prezentat, funcţia operator=() a returnat *this, ceea ce a permis realizarea atribuirilor multiple (de altfel, aceasta este raţiunea pentru care a fost supraîncărcat acest operator).

Supraîncărcarea *operatorului unar* : acesta nu are parametri, deoarece acţionează asupra unui singur operand, care este deja transmis prin this.

Atenţie! În exemplul prezentat, operatorii = şi ++ modifică valoarea unui operand !
O regulă de stil de programare este aceea de a supraîncărca operatorii cu semnificaţii asemănătoare celor originale (chiar dacă limbajul ne permite orice fantezii în sensul atribuirii unor semnificaţii foarte diferite de sensurile iniţiale).

Funcțiile operator membre – continuare 5

- exemplu **Program 2 – continuare 2**

```
int main()
{
  loc ob1(10, 20), ob2(15, 25), ob3(3, 7);// se creeaza
                        //    trei obiecte initializate

  ob1.arata();// se va afisa 10  20
  ob2.arata();// se va afisa 15  25

  ++ob1;// oare ce se intimpla ?
  ob1.arata();//     se afiseaza 11  21

  ob2 = ++ob1;
  ob1.arata();// afiseaza 12  22
  ob2.arata();//  afiseaza tot 12  22

  ob1 = ob2 = ob3;//   se poate si atribuire multipla
  ob1.arata();//   evident se va afisa 3  7
  ob2.arata();//     se va afisa tot 3  7

  return 0;
}
```

```
10 20
15 25
11 21
12 22
12 22
3 7
3 7
```

Crearea operatorilor de incrementare şi decrementare cu prefix şi sufix

- **Instrucţiunile :**
  - **o++;**
  - **++o;**
  - erau identice în versiuni vechi de C++.
- Dar prin supraîncărcare se pot defini versiuni diferite ale incrementării sau decrementării, care să realizeze diferenţierea dorită:

```
// incrementare cu prefix
tip operator++(){
// corpul operatorului cu prefix
}
// incrementare cu sufix
tip operator++(int x){
// corpul operatorului cu sufix
}
```

```
// decrementare cu prefix
tip operator--(){
// corpul operatorului cu prefix
}
// decrementare cu sufix
tip operator--(int x){
// corpul operatorului cu prefix
}
```

Supraîncărcarea operatorilor prescurtaţi

- **Oricare dintre operatorii "prescurtaţi" din C++ (+=, -= etc.) pot fi supraîncărcaţi de asemenea.**

- **De exemplu += relativ la loc:**

```
loc loc::operator+=(loc op2){
    longitud = op2. Longitud + longitud;
    latitud = op2.latitud + latitud;
    return *this;
}
```

Observaţie: la supraîncărcare unuia dintre aceşti operatori, de fapt se combină o atribuire cu o operaţie de alt tip.

Restricţii la supraîncărcarea operatorilor

- nu se poate modifica precedenţa unui operator
- nu se poate modifica numărul operanzilor preluaţi de un operator. (totuşi, poate fi ignorat unul)
- funcţiile operator nu pot avea argumente implicite
- următorii operatori nu pot fi supraîncărcaţi: **. :: .* ?**

Observaţie : Cu excepţia operatorului = , funcţiile operator sunt moştenite de orice clasă derivată. Dar o clasă derivată este liberă să supraîncarce, relativ la ea însăşi, orice operator (inclusiv cei supraîncărcaţi de clasa de bază).

Supraîncărcarea operatorilor folosind o funcție friend

- Se poate supraîncărca un operator relativ la o clasă folosind o funcţie friend.
- Deoarece un prieten nu este un membru al clasei, nu are un pointer de tip this; de aceea, unei funcţii supraîncărcate de tip friend operator i se transmit explicit operanzii:
- Unul care supraîncarcă un operator binar are doi parametri (operandul din stânga este pasat în primul parametru, iar cel din dreapta în cel de-al doilea parametru)
- Unul care supraîncarcă un operator unar are un parametru.

b Exemplu Program3

```
class loc {
   int longitud, latitud;
public:
   loc() {} //necesara constructia temporara
   loc(int lg, int lt) {
   longitud = lg;
   latitud = lt;
    }
void arata() {
  cout << longitud << " ";
  cout << latitud << "\n";
}
friend loc operator+(loc op1, loc op2); //acum este
                                        // prieten
loc operator-(loc op2);
loc operator=(loc op2);
loc operator++();
};
```

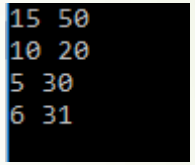
Supraîncărcarea operatorilor folosind o funcție friend –continuare 1

```
// acum, + este supraincarcat folosind functia prieten
loc operator+(loc op1, loc op2) {
  loc temp;
  temp.longitud = op1.longitud + op2.longitud;
  temp.latitud = op1.latitud + op2.latitud;
  return temp;
}

loc loc::operator-(loc op2) {
  loc temp;
  //observati ordinea operanzilor
  temp.longitud = longitud - op2.longitud;
  temp.latitud = latitud - op2.latitud;
  return temp;
}
```

```
loc loc::operator=(loc op2) {
  longitud = op2.longitud;
  latitud = op2.latitud;
  return *this; // returneaza obiectul care a generat
                //  apelarea
}

loc loc::operator++() {
  longitud++;
  latitud++;
  return *this;
}
```

Supraîncărcarea operatorilor folosind o funcție friend –continuare 2

```
int main() {
  loc ob1(10, 20), ob2(5, 30);
  ob1 = ob1 + ob2;
  ob1.arata();
  ob1 = ob1 - ob2;
  ob1.arata();
  ob1 = ob2;
  ob1.arata();
  ++ob1;
  ob1.arata();
  return 0;
}
```



Supraîncărcarea operatorilor folosind o funcție friend – continuare 2

b **restricţii care se aplică funcţiilor de tip friend operator :**

- **nu pot fi supraîncărcaţi operatorii =, (), [] sau -> folosind aceste funcţii**
- **când supraîncărcaţi operatorii de incrementare sau decrementare utilizând funcţii friend, trebuie să folosiţi un parametru de referinţă (vezi următorul paragraf).**

Folosirea unui friend pentru a supraîncărca ++ sau --

- ++ sau -- presupun modificarea operanzilor
- dacă surpraîncărcaţi aceste operaţii folosind un friend, atunci operandul este transmis ca parametru prin valoare (deci o funcţie de tip friend operator nu are cum să modifice operandul - deoarece acestei funcţii nu i se transmite pointerul this ca operand, ci doar o copie a acestuia, nici o modificare adusă parametrului nu afectează operandul care generează apelarea)
- puteţi corecta acest lucru specificînd parametrul funcţiei friend operator ca referinţă .
- Exemplu Program4

```
class loc {
  int longitud, latitud;
public:
  loc() {}
  loc(int lg, int lt) {
  longitud = lg;
  latitud = lt;
}
void arata() {
  cout << longitud << " ";
  cout << latitud << "\n";
}
```

```
loc operator=(loc op2);
friend loc operator++(loc &op);
friend loc operator--(loc &op);
};

loc loc::operator=(loc op2) {
  longitud = op2.longitud;
  latitud = op2.latitud;
  return *this;  // returneaza obiectul care a generat
                 // apelarea
}
```

Folosirea unui friend pentru a supraîncărca ++ sau -- - continuare 1

```
//acum un prieten   - foloseste referinta
loc operator++(loc &op) {
  op.longitud++;
  op.latitud++;
  return op;
}

//il face prieten pe op -- - foloseste referinta

loc operator--(loc &op) {
  op.longitud--;
  op.latitud--;
  return op;
}
```

```
int main() {

  loc ob1(10, 20), ob2;

  ob1.arata();
  ++ob1;
  ob1.arata(); //afiseaza 11 21

  ob2 = ++ob1;
  ob1.arata(); //afiseaza 12 22

  --ob2;
  ob2.arata(); //afiseaza 11 21
  return 0;
}
```

```
10 20
11 21
12 22
11 21
```

Realizarea flexibilității cu funcțiile friend operator

Situaţie :

ob+100 //valida

100+ob //invalida

(necesitatea de a poziţiona în unele aplicaţii mereu obiectul în stânga este câteodată o povară care poate determina eşecuri)

- **Soluţia** : supraîncărcarea folosind o funcţie **friend**, nu o funcţie membru (în acest caz, funcţiei operator îi sunt transmise explicit ambele argumente - pentru a permite atât **obiect+întreg** cât şi **întreg+obiect**, prin supraîncărcare de două ori - o versiune pentru fiecare situaţie)
- Exemplu **Program5**

```
class loc {
  int longitud, latitud;
public:
  loc() {}
  loc(int lg, int lt) {
  longitud = lg;
  latitud = lt;
}
```

```
void arata() {
  cout << longitud << " ";
  cout << latitud << "\n";
}

loc operator+(loc op2);
friend loc operator+(loc op1, int op2);
friend loc operator+(int op1, loc op2);
};
```