## Supraîncărcarea operatorilor (continuare)

## Supraîncărcarea operatorului de atribuire

- Operatorul de atribuire se poate supraîncărca numai prin funcţii membru
- Pentru a asigura un comportament similar operatorului de atribuire ce operează asupra tipurilor de date standard, metoda ce supraîncarcă operatorul de atribuire trebuie sa returneze o referinţă către clasa căreia îi apaţine
- O atenţie specială trebuie acordată situaţiilor în care clasa menevrează zone de memorie alocate dinamic, caz în care trebuie asigurată operarea corectă inclusive în cazul autoatribuirii
- Pentru operatorii de tip *op*= este necesară supraîncărcarea explicită, nefiind suficientă supraîncărcarea separată pentru *op* şi respectiv =
- Forma generală de supraîncărcare a operatorului de atribuire este :

```
NumeClasa&  NumeClasa:: operator =  (const NumeClasa & operand)
{
        ……
        return *this;
}
```

Exemple:

a. Supraîncărcarea operatorului de atribuire pentru clasa Complex :

```
Complex& Complex::operator = (const Complex & z)
 {
        real= z.real;
        imag=z.imag;
        return *this;
  }
```

b. Supraîncărcarea operatorului de atribuire pentru clasa Sir.

```
Sir& Sir::operator=(const Sir& sursa)
{
   if(this != &sursa)
   {
     delete []psir;
     lung=sursa.lung;
     psir=new char[lung+1];
     strcpy(psir, sursa.psir);
   }
   return *this;
}
```

- În general, pentru o clasă NumeClasa, operatorul generic *op*= se supraîncarcă astfel :

```
NumeClasa&  NumeClasa:: operator op=  (const NumeClasa & operand)
{
        ……
        return *this;
}
```

Exemplu : Supraîncărcarea operatorului += pentru clasa Complex :

```
Complex& Complex::operator += (const Complex & z)
 {
        real=real+z.real;
        imag=imag+z.imag;
        return *this;
  }
```

## Supraîncărcarea operatorilor binari de relaţie

- Forma generală de supraîncărcare a operatorilor de relaţie este :

```
bool NumeClasa:: operator op_rel (const NumeClasa & operand) const
{
        ……
        return expresie_relatie;
}
```

Exemplu : Supraîncărcarea operatorului < pentru clasa Copac.

```
bool Copac::operator <(const Copac& c) const
{
        return inalt<c.inalt;
}
```

Tema : Proiectaţi clasa Copac astfel încât că se poate face si comparaţii de forma intreg < obiect din clasa Copac !

## Supraîncărcarea operatorilor unari de incrementare/decrementare

- În general, pentru o clasă NumeClasa, operatorul generic, în notaţie prefixată *op_inc/dec_pre* se supraîncarcă astfel :

```
NumeClasa&  NumeClasa:: operator op_inc/dec_pre ()
{
        ……
        return *this;
}
```

- În general pentru o clasă NumeClasa, operatorul generic, în notaţie postfixată *op_inc/dec_post* se supraîncarcă astfel :

```
NumeClasa  NumeClasa:: operator op_inc/dec_post (int)
// Se introduce un parametru fictiv de tip int . Chiar şi în antetul funcției acest parametru nu are nume.
// Compilatorul foloseşte această informație pentru a face distincție între cele două notații
{
        NumeClasa temp(*this); // se crează o copie a stării curente a obiectului
        // incrementarea/decrementarea ce afectează efectiv obiectul care a apelat operatorul
        ……
        return temp; // se returnează obiectul (o copie a acestuia mai precis!) cu starea de dinaintea
                        //  incrementării/decrementării
}
```

Exemple de supraîncărcare a operatorului de incrementare pentru clasa Copac :

```
Copac& Copac::operator++ () // incrementare în notație prefixată
{
   inalt++;
   return *this;
}

Copac Copac::operator++ (int)  // incrementare în notație postfixată
{
   Copac temp(*this);
   inalt++;
   return temp;
}
```

## Supraîncărcarea operatorului de indexare []

- În general, operatorul de indexare [] este necesar pentru a accesa date de tip tablou cu tipul de bază *tip* , inclusiv pentru instanţieri de obiecte constante, se supraîncarcă cu prototipuri de forma :

  tip operator[](***size_t***) ***const***;
  tip& operator[](***size_t***);

***size_t*** corespunde unui tip de date întreg returnat de operatorul ***sizeof*** şi este definit în fişierul header ***cstring*** ca fiind tip intreg ***unsigned***.

Observaţie:

Pentru depanarea programelor este utilă în anumite situaţii folosirea macro-ului ***assert***

void assert (int expression);

Daca expresia argument *expression* are valoarea zero (i.e. ***false***), se afişează un mesaj pe dispozitivul standard de eroare şi se apelează funcţia ***abort*** , încheindu-se execuţia programului.

Tipul mesajului afişat depinde de implementarea compilatorului. Un format uzual este:

Assertion failed: *expression*, file *filename*, line *line number*

Acest macro este dezactivat dacă înaintea includerii fişierului header assert.h este definit un macro cu numele NDEBUG. Aceasta permite programatorului să includă oricâte apeluri ale macroului assert în codul sursă code pe durata depanării programului, dezactivându-le apoi pentru varianta finală prin includerea liniei de mai jos la începutul codului, înaintea includerii fişierului header assert.h:

#define NDEBUG

Exemplu. Să se construiască o clasă pentru operarea cu vectori cu elemente de tip intreg, pentru care să se supraîncarce operatorul de indexare:

```
#include<iostream>
#include <assert.h>

using namespace std;

class VectInt
{
    int   *pVect;
    int   m_Size;

public:

    VectInt(int n = 10);
    ~VectInt()
    {
        delete []pVect;
    }
    int& operator[](int i)
    {
        assert (i >= 0 && i < m_Size);
        return pVect[i];
    }
};
```

```
VectInt::VectInt(int n)
{
        m_Size=n;
        assert(n > 0);
        pVect = new int[m_Size];
}

int main(void)
{
    VectInt v(5);
    int i;
    for(i=0;i<5;i++)
        v[i]=10*i;
    for(i=0;i<5;i++)
        cout<<v[i]<<'\t';
    cout<<endl;
    return 0;
}
```

## Membri statici

## Date membru statice

- Până în prezent, fiecare obiect al unei clase işi avea propriile date membru
- Există aplicaţii în care este eficient să existe date membru comune tuturor obiectelor clasei, cu alte cuvinte să existe o singură copie a datei membru respective pentru toate obiectele clasei. Astefel de date membru se numesc **date membru statice.**
- Datele membru statice au următoarele caracteristici :

    - Se declară folosind cuvântul rezervat ***static***
    - Pot fi ***private/protected/public***
    - O dată membru statică este creată înaintea instanţierii oricărui obiect al clasei respective
    - În definiţia unei clase, în cazul datelor membru statice avem de a face efectiv cu o declaraţie a membrului static, în sensul că NU SE FACE REZERVARE DE MEMORIE pentru data respectivă. Rezervarea de memorie se face separat în domeniul fişier (global), chiar şi pentru membri statici privati/protejaţi, care insă pot fi accesaţi astfel doar la definiţia datei
    - O dată membru statică se poate referi fie prin intermediul unui obiect al clasei, fie independent

## Funcţii membru statice

- Funcţiilor membru statice nu li se transmite pointerul ***this*** şi de aceea ele nu pot accesa date membru nestatice ; ele operează doar asupra datelor membru statice
- Ele pot fi apelate fie prin intermediul unui obiect al clasei, fie independent

Reamintim faptul că funcţiile membru ale unei clase au următoarele proprietăţi:

1. Au acces la membri protejaţi şi privaţi ai clasei
2. Sunt în domeniul clasei
3. Li se transmite pointerul ***this***

Funcţiile membru statice au numai proprietăţile 1 şi 2.

Funcţiile friend au numai proprietatea 1.

Exemple de utilizare a datelor membru şi funcţiilor statice

```
#include <iostream>
using namespace std;
class Clasa
{
private:
        static int   a; // data membru statica privata
        int    b;
public:
        static int c;  // data membru statica publica
        static void Init(int=10);  //functie membru statica
        void Set(int i, int j){a=i;b=j;}
        friend ostream& operator<<(ostream&, const Clasa&);
};
ostream& operator<<(ostream& out, const Clasa& o)
{
    out<<" a static ="<<o.a<<" b nestatic ="<<o.b<<endl;
    return out;
}
void Clasa::Init(int x)
{
      a=x;
//b=99; object(or object pointer) required to access  non-static data member
}

int Clasa::a = 88;     // definitie data membru statica
int Clasa::c;          // definitie data membru statica

int main()
{
     Clasa::c=100;
     // fara definitie apare eroare de link : Clasa::c is an undefined
     // reference
     //Clasa::a=99;  access to private member is not allowed

     cout<<"c="<<Clasa::c<<endl;
     Clasa x,y;
     cout<<"x="<<x;
     x.Set(1,1);
     cout<<"x="<<x;
     y.Set(2,2);
     cout<<"y="<<y;
     cout<<"x="<<x;
     Clasa::Init(100);
     cout<<"y="<<y;
     cout<<"x="<<x;
     return 0;
}
```

**REZULTATE**

```
c=100
x= a static =88 b nestatic =-858993460
x= a static =1 b nestatic =1
y= a static =2 b nestatic =2
x= a static =2 b nestatic =1
y= a static =100 b nestatic =2
x= a static =100 b nestatic =1
```

```cpp
#include <iostream>
#include <string.h>
using namespace std;
class Angajat
{
    char *nume;
    static int nr_ang;  // declaratie
   public:
    Angajat(const char *);
    Angajat(const Angajat &c);
    ~Angajat();
    const char *getNume() const;
    static int getNr_ang();
};
int Angajat::getNr_ang()
{
    return nr_ang;
}
Angajat::Angajat(const char *s)
{
    nume=new char[strlen(s)+1];
    strcpy(nume,s);
    ++nr_ang;
}
const char *Angajat::getNume()
const
{
    return nume;
}

Angajat::Angajat(const Angajat& c)
{
    nume=
       new char[strlen(c.nume)+1];
    strcpy(nume,c.nume);
    ++nr_ang;
}

Angajat::~Angajat()
{
    delete  []nume;
    --nr_ang;
    cout<<nr_ang<<endl;
}

// definitie
int Angajat:: nr_ang=0;
```

```cpp
int main(void)
{

cout<<Angajat::getNr_ang()<<endl;
const Angajat ob("Sef");

cout<<Angajat::getNr_ang()<<endl;
Angajat *pAng1=
          new Angajat("Ionescu");

cout<<Angajat::getNr_ang()<<endl;
Angajat *pAng2=
          new Angajat("Popescu");

cout<<Angajat::getNr_ang()<<endl;
Angajat a(*pAng1);

cout<<Angajat::getNr_ang()<<endl;
delete pAng1;
delete pAng2;

cout<<Angajat::getNr_ang()<<endl;
return 0;
} // REZULTATE  0 1 2 3 4 3 2 2 1 0
```