C5: Supradefinirea operatorilor; Agregarea

Cuprins:

- Supradefinirea operatorului de indexare: []
- Supradefinirea operatorilor de insertie/extragere “<<”, ”>>”
- Supradefinirea operatorilor new, delete, new[], delete[]
- Polimorfism - definitie
- Agregarea (compunerea)

C5: Supradefinirea operatorilor; Agregarea

Considerati clasa vector:

```
class vector
{   int n;
    int *buf;
public:
    vector(int n=0)
    {
       this->n=n;
       if (n<=0) buf=NULL;
              else buf=new int[n];
     }

     vector (const vector& v)
     {
        buf = NULL;
       *this=v;
     }

int getDim()    {
return n;}
```

```
vector& operator=(const vector& v)
{
   if(this!=&v)  {
            if (buf!=NULL) delete [] buf;
            if (v.n<=0) buf=NULL;
              else{
                   buf=new int[v.n];
                   n=v.n;
                   for (int i=0; i<n;i++)
                          buf[i]=v.buf[i];
                }
       }
    return *this;
}

~vector(){
     if (buf!=NULL) delete [] buf;
}

};
```

C5: Supradefinirea operatorilor; Agregarea

Operatorul [ ]

- este denumit operator de indexare ;
- functia **operator []()** – se implementeaza ca functie membra a clasei ;
- apelare: **v[i]** ⇔ acces pentru modificarea sau recuperarea valorii elementului de pe pozitia i a sirului/vectorului x;
- **v[i] ⇔ v.operator [](i)**

Exemplu:

```
class vector
{           int n;
            int* buf;
    public:
            //….alte metode

            int& operator [](int i);
            //vreau sa aflu sau sa modific valoarea unui element din buf
};
```

```
int& vector::operator[](int i)
//intorc o referinta ca sa pot sa si modific elementul i
{
        assert (i >= 0 && i < this->n);        // verificare index; #include <assert.h>
        return this -> buf[i];                    //sau  direct:  return buf[i];

}

//Utilizare
vector v(3); //creez un obiect vector -  prin apel constructor cu un parametru-dimensiunea
          // in constructor setez dim cu 3 si aloc spatiu pentru buf (3 intregi)

for (int i=0;i<v.getdim();i++)
        v[i]=i;        //elementul de pe pozitia i din buf obiectului v - v[i] -  ia valoarea i.
// ce se intampla daca nu intorceam din functie o referinta? Testati!

for (int i=0;i<v.getdim();i++)
    cout<<v[i]<<" ";   //afisare element de pe pozitia i din buf lui v
```

C5: Supradefinirea operatorilor; Agregarea

Supradefinirea operatorilor de insertie/extragere “<<”, ”>>” :

- necesari cand dorim sa afisam/citim obiecte de tipuri de date definite de noi

```
class vector
{               int n;
                int* buf;
    public:
//
};
```

```
//doresc sa afisez un obiect de tip vector v1;

vector v1(3);
v1[0]=v1[1]=v2[2]=1;      //il populez cu valori  folosind operatorul de indexare

cout << v1;
// efect:
                        vectorul:
                        Nr_elem: 3
                        Elem din buf: 1 1 1
```

=> supradefinire operator <<

Operatorul << este deja supradefinit pentru tipurile de date de baza:

```
int a=4;
float f=4.4;
double d=1.1;
cout << a << f << d << “   ”;
```

Mod de functionare:

Operatorul de afisare este **binar**:

- primul operand – este dispozitivul de iesire (de tip *ostream*);
- al doilea operand – este obiectul ce urmeaza sa fie afisat.

```
cout << v1;          ⇔          operator << (cout, v1);
//cout – obiect de tip ostream; definit in namespace-ul std;

//dar la fel de bine as putea sa afisez intr-un fisier care este tot un stream de
//iesire - o sa revenim

//un operand e de tip ostream =>  trebuie inclusa biblioteca iostream
```

In functionarea cu mai multi parametri se aplica asociativitatea **de la stanga la dreapta**.

cout << a << b; ⇔ ((cout << a) << b);

⇔ operator << (operator<<(cout, a), b);

Operatorul << trebuie sa returneze dispozitivul de iesire (cout); acesta este de tipul stream de iesire, mai exact: **ostream** => trebuie inclusa biblioteca iostream

Prototipul functiei:

friend ostream & operator << (ostream &, const vector &);

Se apeleaza la varianta “functie friend” deoarece primul operand nu este de tipul clasei!!!

Nu se modifica obiectul afisat, dar se modifica dispozitivul/fisierul pe/in care se face afisarea/salvarea datelor.

```
// implementarea in interiorul unui fisier sursa; am declarat functia ca friend in header-ul
// clasei vector: friend ostream & operator<<(ostream & , const vector & );


#include <iostream>
using namespace std;

ostream& operator<< (ostream& dev, const vector& v)
{
        dev << "Vector:" <<endl;
        dev << "Nr Elem:" << v.n << endl;
        if (v.buf!=NULL)
          for(int i = 0; i < v.n; i++)
              dev << v.buf[i]<< "   ";
        dev << endl;
        return dev;
}

// Utilizare:
vector v2(3); // apel constructor cu parametru dimensiunea bufferului
vector v3(v2);  //apel constructor de copiere
cout << v2 << v3;
```

supradefinire operator >>

```
// am dori:
vector v1;
cin >>v1;

// efect:   Vectorul:
                              Dati nr_elem:
                              Dati elem:
```

Prototipul functiei:

friend istream & operator >> (istream &, vector &);

Se apeleaza tot la varianta “functie friend” deoarece primul operand nu este de tipul clasei – ci e un flux de intrare - istream (trebuie inclusa biblioteca iostream).

Vreau sa citesc date in al doilea parametru (deci il transmit ca referinta neconstanta).

```
//in headerul clasei vector am declarat functia ca prietena:
//friend istream& operator>>(istream&, vector &)
// implementarea functiei friend in interiorul unui fisier sursa:

istream & operator >> (istream & dev, vector & v){
            cout << "Vectorul:"<<endl;
            cout << "Dim. max:" << v.n << endl;
            dev>>v.dim;
            if (v.buf!=NULL) delete [] buf;
            if (v.n>0){
                        buf=new int[v.n];
                        for(int i = 0; i < v.n; i++)
                                    dev >> v.buf[i];
            } else v.buf=NULL;
            return dev;
}
// apel:
vector v3, v4; // apel constructor fara parametri
cin >> v3>>v4;
```

Exemplu: (clasa telefon)

```
#include <cstdlib>
#include <iostream>
using namespace std;

class telefon
{
        char cod_interurban [5];
        char nr_telefon [8];
  public:
       friend ostream & operator << (ostream &, const telefon &);
       friend istream & operator >> (istream &, telefon &);
}; //Mai am alte functii la dispozitie in afara de cele doua definite?

ostream & operator << (ostream & dvo, const telefon & t)
{
       dvo << "\n Nr. Telefon: \n“ << "Cod Interurban:" <<
                                  t.cod_interurban<< "Numar Telefon" << t.nr_telefon;
return dvo;
}
```

```
istream & operator >> (istream & devi, telefon & t)
{
        devi >>  t.cod_interurban;
        devi >>  t. nr_telefon;
        return devi;
}

// utilizare:

int main()
{
        telefon tf1;

        cout << "Introd. Nr. De telefon in formatul xxxx_xxxxxxxx \n";
        cin >> tf1;
        cout << "Date inregistrate: " << tf1;

       telefon tf2(tf1), tf3;
       tf3=tf2;
       cout<<tf2<<tf3;

      return 0;
}
```

C5: Supradefinirea operatorilor; Agregarea

Supradefinirea operatorilor new, new[], delete, delete[] :

tip_obj * obj = new tip_obj(param1,..,paramn); // sau: tip_obj * obj = new tip_obj;

new:

- aloca suficient spatiu pentru obj de tipul tip_obj
- apeleaza constructorului cu/fara parametrii pe a crea obiectul in acel spatiu
- intoarce un pointer catre adresa obiectului creat
- converteste automat acest pointer la tipul tip_obj*

void* operator new(size_t);

Operator new primeste ca parametru dimensiunea obiectului pentru care se vrea alocat spatiu (tipul size_t) si returneaza un pointer (void*) catre zona de memorie.

size_t este un tip de date capabil sa reprezinte dimensiunea oricarui obiect in biti; este tipul returnat de functia sizeof (definit in biblioteca <new> => trebuie inclusa).

delete obj;

delete - apeleaza destructorul pentru obj de tipul tip_obj si apoi elibereaza memoria

void operator delete(void*) ;

- Primeste ca parametru un pointer void* si returneaza void.

---

vecObj = new tip_obj[cate];

void* operator new[](size_t);

– primeste ca parametru cate obiecte de tipul respectiv ar trebui create; apeleaza constr.

---

delete [] vecObj;

void operator delete[](void*);

- primeste ca parametru adresa de inceput a vectorului; [] cauta cate elemente are vectorul si apeleaza de atatea ori destructorul pentru tip_obj.

De ce as supradefini acesti operatori?

In cazul in care se ramane fara spatiu de memorie o sa avem niste comportamente inexplicite ale aplicatiei realizate.

As vrea sa primesc mesaje clare - care sa explice ce s-a intamplat –> pot sa fac asta supradefinind acesti operatori.

```
void* orice_tip::operator new(size_t x) throw (bad_alloc)
 {  void* m=malloc(x);
   cout<<"aloc spatiu cu operatorul new supradefinit";
   if (m==NULL){
            cout<<"Nu mai am spatiu liber";
            throw bad_alloc();   //#include new
            }
   return m;
 }

 void* orice_tip ::operator new[](size_t x) throw (bad_alloc)
 {
     void*m=malloc(x*sizeof(orice_tip));
     cout<<" aloc spatiu cu operatorul new[] supradefinit ";
     //m=NULL;// testati ce se intampla – simulez lipsa de spatiu
     if (m==NULL) {
                cout<<"Nu mai am spatiu liber";
                throw bad_alloc();   //#include new
                }
     return m;
 }
```

Exceptii : o functie/un program poate sa “arunce” o exceptie in cazul unei situatii nedorite (ex. nu mai exista memorie libera).

Exista exceptii predefinite, de exemplu, bad_alloc - aruncata de new si new[].

```
void orice_tip::operator delete(void*   m)
{
      free(m);
      cout<<“eliberez cu delete";
}



void orice_tip ::operator delete[](void*  m)
{
      free(m);
      cout<<“eliberez cu delete[]";
}
```

//se folosesc ca operatorii new si delete deja implementati
//ascund implementarea operatorilor standard new si delete pentru tipul pentru
//care au fost supradefiniti

C5: Supradefinirea operatorilor; Agregarea

Exceptii

O exceptie C++ este un raspuns la o circumstanta exceptionala care apare in timpul functionarii unui program (ex: impartire cu 0; lipsa memorie libera; depasirea domeniului unui vector din std, etc).

Exceptiile pun la dispozitie un mod de transfer al controlului dintr-o zona de program in alta.

Tratarea exceptiilor (exception handling) foloseste 3 cuvinte cheie: **try, catch** si **throw**.

throw: o secventa de cod/functie arunca o exceptie cand apare o problema

try: un bloc **try** identifica o zona de cod unde este posibil sa apara exceptia. Blocul try este urmat de minim un bloc catch.

catch: un program prinde o exceptie folosind un manipulator de exceptii -“exception handler” in zona de cod unde vrem sa prindem si sa tratam problema. Cuvantul cheie **catch** indica prinderea si tratarea problemei.

```
void  f(){
…..
throw “exception”;
}
```

Daca un bloc de cod/functie arunca o exceptie, aceasta e prinsa folosind o combinatie **try** - **catch**:

```
try {

// cod ce poate sa produca o exceptie cum ar fi f()
f();
//dar pot sa mai fie si alte zone de cod ce pot sa arunce exceptii

}catch( ExceptionType e1 ) {

 // cod care trateaza exceptia e1; (afisare mesaj, executie functii, iese din aplicatie)

}catch( ExceptionType e2 ) {

// cod care trateaza exceptia e2;

}catch( ExceptionType eN )

// cod care trateaza exceptia eN;
}
```

throw:
Exceptiile pot sa fie aruncate oriunde folosind cuvantul cheie **throw**.
Operandul functiei throw determina tipul exceptiei.

try - catch:
Blocul **catch** il urmeaza pe cel de **try** si prinde o exceptie pe care o rezolva intr-un fel.
Poate fi specificat care este tipul exceptiei care se doreste tratata, iar acesta e determinat de declaratia exceptiei care apare in paranteze dupa catch

```
try {
 // cod ce poate arunca o exceptie
}catch( ExceptionType e ) {
 // codul care trateaza exceptia de tipul ExceptionType cu numele e
}
```

Daca se doreste specificarea faptului ca in blocul catch se trateaza orice tip de exceptie atunci in paranteze se pune ... :

```
try {
// cod ce poate sa arunce o exceptie
}catch(...) {
 // cod pentru tratarea exceptiei
 }
```

```
#include <iostream>
 using namespace std;

double imp(int a, int b)
{ if( b == 0 ) throw "Impartire la 0!";  //aruncarea unei exceptii in cazul unei impartiri prin 0
                                         // tipul exceptiei este const char*
return (double)a/b;
}

int main ()
{ int x = 50; int y = 0; double z = 0;
try {
 z = imp(x, y);
cout << z<< endl;
}catch (const char* msg) {
 cout << msg << endl; // afisez un mesajul, dupa care se continua executia
}
try {
 z = imp(x, y+1);
cout << z << endl;
}catch (const char* msg) {
 cout << msg << endl;
}
return 0; }
```

AVANTAJ:

NU opresc executia aplicatie din cauza unei exceptii, ci tratez problema si merg mai departe!

Mecanism foarte util!

Exceptii standard C++ : *nu intra in materia pentru examen, dar necesare in dezvoltarea de aplicatii robuste

Exceptii	Descriere
std::exception	O exceptie //clasa parinte a tuturor exceptiilor standard C++
std::bad_alloc	Poate fi aruncata de **new**.
std::bad_cast	Poate fi aruncata de **dynamic_cast**.
std::bad_exception	Folositoare pentru a arunca exceptii neasteptate
std::bad_typeid	Poate fi aruncata de **typeid**.
std::logic_error	O exceptie care poate fi teoretic detectata prin inspectarea codului.
std::domain_error	O exceptie aruncata cand se foloseste un domeniu matematic incorect
std::invalid_argument	O exceptie aruncata datorita argumentelor invalide.
std::length_error	O exceptie aruncata cand se creaza un std::string prea lung
std::out_of_range	Poate fi aruncata de operatorul[] din std::vector si std::bitset<>::operator[]().
std::runtime_error	O exceptie care nu poate fi teoretic detectata prin inspectarea codului.
std::overflow_error	Aruncata in caz de depasiri matematice.
std::range_error	Aruncata la incercarea de stocare a unei valori care e in afara domeniului.
std::underflow_error	Aruncata in caz de depasiri matematice ale limitelor inferioare.

C5: Supradefinirea operatorilor; Agregarea

Observatii:

1. Este recomandat ca operatorii sa fie supradefiniti astfel:

Operatori	**supradefiniti ca:**
toti operatorii unari	functii membre
=, []	**trebuie** sa fie functii membre
+=, -=,*=, …	functii membre
toti ceilati operatori binari	functii friend

2. Daca pentru un tip nou de date nu supradefinim operatorul =, acesta va fi generat in mod automat (atentie: daca avem atribute de tip pointer carora urmeaza sa le alocam spatiu, este recomandat sa il implementam)

C5: Supradefinirea operatorilor; Agregarea

Observatii legate de tipul argumentelor si tipul returnat in cazul supradefinirii operatorilor

- teoretic, argumentele pot sa fie transmise si returnate oricum in/din functiile operator supradefinite, dar exista o serie de “reguli de buna conduita” care ar trebui urmate:

1. - daca se doreste **utilizarea unui parametru/argument fara modificarea acestuia**, atunci el ar trebui transmis in functie ca **referinta constanta (astfel voi putea sa transmit ca parametri si obiecte temporare)**: ex: complex c, a(2,2); c=**a+complex(1,1)**;

- operatiile aritmetice (+,-,etc) si cele logice(==,>,etc) nu vor modifica valorile parametrilor

- in acest caz, daca operatorul este implementat ca functie membra va fi o **functie membra constanta** (nu modifica atributele obiectelor care apeleaza functia).

- numai operatorii de atribuire(=,+=, etc) au dreptul de a schimba operandul/ /argumentul din stanga **(lvalue)**.

Acesta va fi **transmis prin referinta** deoarece se va modifica. ex: **a+=b**;

2. - tipul returnat depinde de intelesul operatorului

- daca efectul operatorului este acela de a **produce o noua valoare**, va trebui sa fie creat/generat un nou obiect si **returnat prin valoare**.

Ex: operatorul + genereaza un nou obiect care contine suma celor doi parametri.

- acest nou obiect este returnat prin **valoare ca si constanta (returning by value as const)**, astfel incat rezultatul sa nu poata fi modificat in cadrul unei operatii in care el ar fi operandul din stanga(lvalue):

```
//ganditi-va la tipurile de date de baza; nu avem voie sa facem asa ceva
int a=2,b=3,c=6;
a+b=d; //ERROR non-lvalue in assignment
```

De ce nu? S-ar modifica un obiect temporar – iar modificarea s-ar pierde:

```
complex a(2,2),b(2,2);
(a+b).modifica(7,7);//modificarea asupra obiectului returnat de + se pierde.
```

!Pot apela numai functii constante – ex: afisare, etc care garanteaza ca nu vor face modificari pe rezultatul nesalvat undeva anume.

3. - toti **operatorii de atribuire** trebuie sa **modifice operandul din stanga(lvalue)**

- pentru a permite ca rezultatul atribuirii sa fie folosit intr-o expresie in lant: **a=b=c,** este asteptat ca operatorul sa intoarca o referinta catre operandul de tip lvalue pe care tocmai l-a modificat.

Ar trebui ca aceasta referinta sa fie constanta sau nu?

Uneori se doreste realizarea unei noi modificari a obiectului care tocmai a fost modificat prin atribuire:

```
int x,y=2; (x=y)++;

complex a , b(2,2);
(a=b)+=complex(1,1) ;
```

iar modificarea sa trebuie pastrata in a.

In acest caz, tipul returnat de toti operatorii ce implica atribuire ar fi indicat sa fie o **referinta neconstanta** catre lvalue.

4. - pentru operatorii logici toata lumea se asteapta sa primeasca, in cel mai rau caz, un rezultat de tip **int** si in cel mai bun caz un rezultat de tip **bool**.

5. - **operatorii de incrementare si decrementare atat in versiune prefixata cat si in versiune postfixata** modifica obiectul, deci **nu pot sa fie functii constante**.

- versiunea **prefixata returneaza valoarea obiectului dupa modificare**: return *this;

- **versiunea postfixata intoarce valoarea inainte de modificare**, ceea ce inseamna crearea unui **nou obiect**, deci va intoarce un **obiect prin valoare**.

Rezultatul intors ar trebui sa fie const sau nu?

Daca nu este const si avem ceva de genul:

(++a).funct(), funct() va opera asupra lui **a (o referinta)**

dar in cazul **(a++).funct()**, va opera asupra unui **obiect temporar** returnat de varianta postfixata a operatorului. Obiectele temporare sunt constante, ceea ce va fi sesizat de compilator (nu pot chema decat functii const).

Cea mai buna solutie e ca **versiunea prefixata sa returneaza non const si cea postfixata const**.

C5: Supradefinirea operatorilor; Agregarea

Polimorfism (in afara contextului de mostenire) = poli -mai multe; morf - forma

- **abilitatea unei metode cu un anumit nume sa aiba comportamente diferite, in functie de parametrii de intrare** (Ad hoc polymorphism)

Am avut de-a face cu asa ceva pana acum?

Da, in cazul supradefinirii functiilor si operatorilor:

```
int a=3,b=5;
cout<<(a+b); //parametrii de intrare in functia operator+  si operator<< sunt de tip int
```

respectiv :

```
complex a(3,3),b(5,5);
cout<<(a+b); //parametrii de intrare in functia operator+  si operator<< sunt de tip complex
```

Ex: Presupunem ca avem o clasa complex cu “operatii aritmetice” cu o metoda aduna (sau operator+) care primeste ca argumente doua numere oarecare (reale sau complexe).

Daca polimorfismul nu ar fi luat in calcul, atunci ar trebui implementate functii cu nume diferit pentru toate cele trei cazuri:

- nr real + nr complex;
- nr complex + nr real ;
- nr complex + nr complex;

iar utilizatorul clasei ar trebui sa tina minte toate cele trei nume de functii pentru a face o adunare.

Cum?

Ca functii friend pentru ca primul operand nu e mereu de tipul clasei.

Cu ajutorul polimorfismului, cele trei functii distincte pot avea **acelasi nume,** iar:

- programatorul care utilizeaza clasa complex nu trebuie sa retina decat o singura denumire
- compilatorul alege varianta corecta in functie de ordinea, numarul si tipul parametrilor de intrare furnizati.

Prin **polimorfism** se realizeaza o interfata de comunicare a obiectului cu exteriorul mult mai unitara si flexibila: metoda “adunare” are un singur nume si nu trei, eventual se supradefineste operatorul+ in cele 3 cazuri.

Dezvoltatorii de cod pot sa se concentreze mai bine asupra modului cum sa foloseasca obiectele, fiind scutiti de detalii de implementare.

C5: Supradefinirea operatorilor; Agregarea

Reutilizarea codului in dezvoltarea de aplicatii orientate pe obiect

Cea mai mare promisiune facuta de programarea orientata pe obiect a fost cea a reutilizarii codului (code reuse).

Problema se poate pune in termeni de reutilizarea codului: in cadrul aceluiasi proiect sau inter proiecte.

Promisiunea e cu siguranta respectata in ce priveste primul context.

(Ati mai reutilizat cod pana acum?)

Importanta: Eliminarea codul duplicat – conduce la:

- **eficienta** in termeni de durata de implementare a unui proiect
- usurinta in ce priveste lizibilitatea codului->**modularizare**; arhitectura robusta

~~Copy-Paste~~ Be smart!

Mecanismele prin care putem reutiliza codul scris pentru o clasa pentru dezvoltarea unui nou tip de date (alta clasa):

- Compozitia/Agregara (Composition)
- Derivarea/Mostenirea (Inheritance)

C5: Supradefinirea operatorilor; Agregarea

Agregarea /compozitia – o metoda de reutilizare a codului

O clasa contine unul/mai multe atribut/(e) de tipul altei/altor clase.

Ex:

```
class Angajat
{ Adresa a;
  NumeComplet nc;
  int salariu;
//metode
};
```

```
class Companie
{ Adresa a;
  char* numeFirma;
  Angajat *ang;
 //metode
};
```

```
class Catalog_Materie
{ NumeComplet *stud;
  int *nota;
  int nr_studenti;
};
```

Puteam sa definesc altfel aceste clase?

Adresa si NumeComplet sunt 2 clase, de sine statatoare:

```
class Adresa
{ char*oras,*strada;
  int nr;
  char codpostal[7] ;
};
```

```
class NumeComplet
{ char* nume, *prenume;
  char initiala;
//…
};
```

Teoretic, deoarece clasele Adresa si NumeComplet existau deja, aveau implementate toate functiile necesare accesarii, modificarii si prelucrarii atributelor: constructori fara, cu parametri si de copiere, destructor, operator=, operator<<, operator>>, metode de tip set/get atribut cu verificarea consistentei datelor .

Acestea pot fi reutilizate in clasa Angajat, Companie, Catalog_Materie. **Cum?**

Agregarea

- **Agregarea este** procedeul prin care se creaza un nou tip de date (o noua clasa) folosind tipuri de date (clase) deja existente
- **Relatia de agregare – o relatie intre clase de tipul “has a” , “has many”**

```
class A{
  //  lista atribute;
  //  lista metode;
};
```

```
class B{
    A a;
   // alte atribute si metode
};
```

```
class C{
    A *vec;
    B b;
   // alte atribute si metode
};
```

- clasa B are un atribut de tipul clasei A
- clasa B nu are acces direct la atributele din A, dar poate sa utilizeze functiile acesteia pentru a realiza operatiile de interes
- se protejaza incapsularea datelor din A
- se reutilizeaza codul deja implementat (speram ca bine si complet) pentru A.

Observatie: Putem avea si alte situatii: clasa C are mai multe atribute de tip A: A*vec; precum si un atribut de tip B b ;

```
#include <iostream>
using namespace std;
class Pagina {
   private: char* text; //un text de o anumita lungime l;
          int l;  //poate sa lipseasca - e redundant
          int nr_pag;     //numarul paginii
   public:  Pagina(){
                  text=NULL;l=0;nr_pag=0; //necesar la crearea vectorului de pagini din Carte
             }

             Pagina(char* c, int nr) {  //necesar la populatrea cu pagini cu nr si continut a cartii
                   if (c==NULL) text =NULL;
                   else {  text=new char[(l=strlen(c)) +1];
                           strcpy(text,c);
                           nr_pag=nr;}
              }

             Pagina& operator=(const Pagina &p) {  //in addPag se foloseste atribuirea de
                  if(this!=&p){                                  //obiecte de tip Pagina
                    if (text!=NULL) {
                              delete [] text; text=NULL;
                    }
                    if (p.text!=NULL) {   text=new char[(l=p.l)+1];
                                          strcpy(text,p.text);
                                          nr_pag=p.nr_pag;
                    }     return *this;
                }
```

Tema: Implementati destructorul si operator>>

```
Pagina(const Pagina &p){
                    text = NULL;
                    *this=p;
} // constructor de copiere


friend ostream& operator<<(ostream &dev,const Pagina &p){ // se foloseste in
                        dev<<"Pagina: "<<p.nr_pag<<endl;                    //op<< din Carte
                         if (p.text!=NULL)
                                dev<<"Continut: "<<p.text<<endl<<endl;
                         return dev;
 }

 int get_nr_pag() const {
            return nr_pag;
}

 int get_nr_char() const {
            return l;
}
};
```

Ce mai trebuia obligatoriu implementat pentru aceasta clasa? Constructor copiere, operator=, destructor!

```
#include “Pagina.h”
class Carte {
   private:
         char *titlu;
         Pagina *pag;
         int dim; //cate pagini
   public:  Carte(int d, char* t) //inainte de exec. cod din constr. se aloca sp. pt atrib. Cat?
         {     pag = new Pagina[dim=d];     //apel constructor fara param din Pagina
               titlu=new char[strlen(t)+1];
               strcpy(titlu,t);
          }
          void addPag(const Pagina &p)
          {
                pag[p.get_nr_pag()]=p;
          } //apel operator= din Pagina

          friend ostream& operator<<(ostream &dev,const Carte &p){
               dev<<"Titlul: "<<p.titlu<<endl;
               dev<<"Pagini: "<<endl;
               for (int i=0;i<p.dim;i++)
                   dev<<p.pag[i];      //apel operator<< din Pagina
               return dev;
          }
          int getDim() {return dim;}
}; //in Carte nu am acces la atributele din Pagina pentru ca sunt private=> lucrez cu metode
```

```
int main()
{
   Carte c(20,"Felix");
   for (int i=0;i<c.getDim();i++)
        c.addPag(Pagina("text",i));//initializarea obiectelor din container
   cout<<c;

   return 1;
}
```

As putea sa dezvolt aplicatia folosind compozitia

```
class Biblioteca{
        Carte *buf;
        int nr_cart;
        char *nume;
        Adresa adr;
//metode / functii friend
    - constructori, destructor
    - operator =, <<
    - sortare buf dupa titlu, etc
};
```

Ce ar trebui implementat in Carte? Tema – implementati.

Tipuri de agregare:

- ***puternica (strong)*** – daca nu exista obiectele continute prin agregare, existenta obiectului container inceteaza

Ex: class Carte {
private: Pagina *pag;
public://…};
//daca o Carte nu continea decat atributul Pagini *pag; si nu as fi avut nicio pagina atunci //obiectul nu exista.

- ***slaba (weak)*** – obiectul-container poate exista si in absenta obiectelor agregate

Ex: o biblioteca poate exista si fara carti

Initializarea obiectelor continute intr-o clasa agregata poate fi facuta in 3 momente de timp distincte:

-la **definirea** obiectului (inaintea constructorului: folosind fie o valoare initiala, fie blocuri de initializare)

-in cadrul **constructorului**

-chiar **inainte de folosire** (acest mecanism se numeste *lazy initialization*) ~ ca in exemplul anterior (am facut alocare de spatiu pentru Pagina din vectorul de pagini, dar nu am dat valori atributelor din Pagina)

```
#include <iostream>
using namespace std;

class Adresa {
   private:
          char* str;
          int nr;

   public:  Adresa(){str=NULL; nr=0;} //necesar la crearea unui obiect Persoana

             Adresa(char* c, int n):nr(n) //aloc spatiu pt nr si il initializez in acelasi timp
          //se apeleaza pseudoconstructorul pentru nr de tip int inainte de a intra in constructor;
          //pseudoconstructorii se folosesc pentru crearea si initializarea tipurilor de date de baza
            {      cout<<"const cu param Adresa";
                   if (c!=NULL) {str=new char[(strlen(c))+1];
                                 strcpy(str,c);}
                   else str=NULL;
             } //necesar in constructorul din Persoana

          Adresa(const Adresa&p) // constructor de copiere pentru Adresa
          {    cout<<"const de copiere din Adresa";
               str=new char[strlen(p.str)+1];
               strcpy(str,p.str);
               nr=p.nr;
          } //necesar in constructorul din Persoana
```

<=>

```
str=NULL;
*this=p;
```

```
 Adresa& operator=(const Adresa &p) {
//in Persoana se foloseste atribuirea de obiecte de tip Adresa
      if(this!=&p)
      {
            if (str!=NULL) delete [] str;
            if (p.str!=NULL){
               str=new char[strlen(p.str)+1];
               strcpy(str,p.str);
            } else str=NULL;
            nr=p.nr;
      }
      return *this;
 }

  friend ostream& operator<<(ostream &dev,const Adresa &p){
// in operatorul de << din Persoana se foloseste operatorul de << din Adresa
        if (str!=NULL) dev<<"Strada: "<<p.str<<endl;
        dev<<"Nr. : "<<p.nr<<endl<<endl;
        return dev;
  }

  ~Adresa(){
            if  (str!=NULL) delete []str;
  }
};
```

```
class Persoana {
   private:
              Adresa adr;
              char *nume;

   public:
         Persoana(const Adresa &d, char* n):adr(d)
//apel constructor de copiere din Adresa pentru a crea atributul adr ca si copie a
//parametrului d;   inainte de a intra in constructorul Persoana
         {      cout<<“in constr Persoana vs 1”;
                nume= new char[strlen(n)+1];
                strcpy(nume,n);
          }
//sau
//          Persoana(const Adresa &a, char* n)//se apeleaza automat constr. f. param din Adresa care
//          {     cout<<“in constr Persoana vs 2”;                  //aloca spatiu pt un obj de tip Adresa
//                adr=a; //apel operator=
//                nume= new char[strlen(n)+1];
//                strcpy(nume,n);
//          }

//          Persoana(char* s, int nr, char* n):adr(s,nr) // creez adr cu constructorul cu parametrii
//          {     cout<<“in constr Persoana vs 3”;
//                nume= new char[strlen(n)+1];
//                strcpy(nume,n);
//          }
```

```
//sau
        Persoana(char* s, int nr, char* n)  //s-a alocat deja spatiu pentru adr;
        {       Adresa aux(s,nr);
                adr= aux;                           //apel operator= din Adresa
                nume= new char[strlen(n)+1];
                strcpy(nume,n);
        }

        friend ostream& operator<<(ostream &dev,const Persoana &p){
                dev<<"Nume: "<<p.nume<<endl;
                dev<<"Adresa: "<<endl;
                dev<<p.adr<<endl;               //apel operator<< din Adresa
                return dev;
        }
    };
int main()
{
   Persoana c("blv. Ceva",20,"Ana Maria");
   cout<<c;
   Adresa a("str",10);
   Persoana c1(a,"Vasile");
   cout<<c1;
   return 1;
}
```

operatorul= generat automat pentru clasa Persoana il apeleaza automat pe cel implementat de noi in adresa. Dar constructorul de copiere/destructorul? Testati!

Tema:In toate metodele din Persoana verificati ca numele sa nu fie nealocat sau NULL. Implementati operatorul=, constr. de copiere si destr.

```
#include <iostream>
using namespace std;

class A{
        public:
        A(){
           cout<<"constr fara param"<<endl;
        }

        A(const A& a){
           cout<<"constr copiere"<<endl;
        }

        A& operator=(const A& a){
           cout<<"op="<<endl;
           return *this;
        }

        ~A(){
           cout<<"destr"<<endl;

        }
};
```

```
class B{
        A a;
};

int main() {
        B b;
        B c(b);
        c=b;
        return 0;
}
```

Testati!

C5: Supradefinirea operatorilor; Agregarea

"Teme"

- Rulati aplicatia de mai sus
- Testati constructorii din Persoana si ordinea de apel constr Persoana – constr Adresa
- Comentati pe rand constructorii si metodele din Adresa si observati ce se intampla.
- Ce mesaje de eroare obtineti? De ce?
- Creati un vector de persoane si afisati toate persoanele de pe strada "Virtutii".

Problema

Ganditi-va la un exemplu de agregare intre 2/n clase.
Adica relatii de tipul:

ceva are o /mai multe **altceva**

Scrieti interfetele acestor clase , precizand care e necesitatea metodelor pe care le-ati inclus.

C5: Supradefinirea operatorilor; Agregarea

Optimizare la returnarea unui rezultat – C++98

Cand un obiect este creat pentru a fi intors din functie prin valoare scriem de multe ori ceva de tipul:

```
return complex(a.re+b.re, a.im+b.im);
```

Aceasta instructiune seamana cu un apel constuctor, dar nu este chiar asa.

Este sintaxa pentru crearea unui obiect temporar : **creaza un obiect temporar ca rezultat al functiei.**

Ce se intampla daca scriem:

```
complex tmp(a.re+b.re, a.im+b.im);
return tmp;
```

In acest caz se intampla 3 lucruri:

- se creaza obiectul tmp prin apelul functiei constructor cu parametri
- la return - constructorul de copiere copiaza obiectul tmp in locatia in care e returnat obiectul
- este chemat destructorul pentru tmp;

Ce se intampla atunci in cazul:

```
return complex(a.re+b.re, a.im+b.im);
```

In cazul “returnarea unui obiect temporar” codul functioneaza diferit:

- cand compilatorul vede aceasta instructiune stie ca nu mai exista alta intrebuintare a obiectului pe care trebuie sa il returneze

- asa ca va construi obiectul direct in locatia in care acesta trebuie returnat.

- va folosi un singur constructor (nu si constructorul de copiere);
- nu este initiat niciun apel destructor, deoarece nu se creaza un obiect local.

Tema bonus – 0.5p – deadline 30.11

In standardul C++ 11, 14, 17 apar niste functii noi generate automat:

Move constructor
Move assignment

si un nou tip de date:

r-value reference

Documentati aceste noi functii si tip nou de date – necesitate, utilizare, in ce standard au aparut.

Realizati exemple legate de implementarea lor si momentul apelarii.

Alte elemente noi:
Smart pointers - documentati– necesitate, utilizare, in ce standard au aparut. Realizati un exemplu de utilizare.