Facultatea de Matematica si Informatica
Universitatea din Bucuresti

Programare orientata pe obiecte

- suport de curs -

Dobrovat Anca - Madalina

An universitar 2019 – 2020
Semestrul I
Seria 25

Curs 4 & 5

23 - 30/10/2019

Agenda cursului

1.Recapitularea discutiilor din cursul anterior.
2. Supraîncărcarea funcțiilor.
3. Copy constructor
4. Supraîncărcarea operatorilor.
    - Supraîncărcarea operatorilor cu funcții friend.
    - Supraîncărcarea operatorilor cu funcții membru.

5. Conversia datelor în C++.
5.1 Conversii între diferite tipuri de obiecte (operatorul cast, operatorul= și constructor de copiere).
5.2 Membrii constanți ai unei clase in C++.
5.3 Modificatorul const, obiecte constante, pointeri constanți la obiecte și pointeri la obiecte constante.

1. Recapitulare curs 3

- functii prieten – se garanteaza accesul la membrii privati/protected;

- clase prieten – toate functiile **membre** din clasa prietena au acces la membrii; privati sau protejati;

- functii inline;

- pointerul “this” (*Cand este apelata o functie membru, i se paseaza automat un argument implicit = un pointer catre obiectul care a generat apelarea (obiectul care a invocat functia)* ==> ***this***.)

- constructorii si destructorul unei clase; constructorul de copiere

1. Recapitulare curs 3

Membrii statici ai unei clase

- date membre:
  - nestatice (distincte pentru fiecare obiect);
  - statice (unice pentru toate obiectele clasei, exista o singura copie pentru toate obiectele).

- cuvant cheie “static”
- create, initializate si accesate – independent de obiectele clasei.
- alocarea si initializarea – in afara clasei.

- functiile statice:
  - efectueaza operatii asupra intregii clase;
  - nu au cuvantul cheie “this”;
  - se pot referi doar la membrii statici.

- referirea membrilor statici:
  - clasa :: membru;
  - obiect. Membru (identic cu nestatic).

1. Recapitulare curs 3

Membrii statici ai unei clase

Folosirea variabilor statice de instanta elimina necesitatea variabilelor globale.

Folosirea variabilelor globale aproape intotdeauna violeaza principiul encapsularii datelor, si deci nu este in concordanta cu OOP

1. Recapitulare curs 3

Membrii statici ai unei clase

Exemplu:

```
class Z
{
    static int x;
    int y;
public:
    Z()   {       x++;    }
    ~Z()   {       x--;    }
    int get_x(){return x;}
    static void afis_x(){cout<<x<<" "<<y<<endl;}
};
int Z::x;
```

```
int main()
{
    //cout<<Z::x<<endl;
    Z A;
    cout<<A.get_x()<<endl;
    //cout<<Z::x<<endl;
    Z B;
    cout<<B.get_x()<<endl;
    cout<<A.get_x()<<endl;
    Z::afis_x();
    //cout<<Z::x<<endl;
    //cout<<A.x<<endl;
    return 0;
}
```

1. Recapitulare curs 3

Membrii statici ai unei clase

Spuneţi dacă programul de mai jos este corect. În caz afirmativ, spuneţi ce afisează, în caz negativ spuneţi de ce nu este corect.

```
# include <iostream.h>
class A {
      int x;
      const int y;
      public: A (int i, int j) : x(i), y(j) { }
      static int f (int z, int v) {return x + z + v;} };
int main() {
      A ob (5,-8);
      cout<<ob.f(-9,8);
      return 0;
}
```

R: Nu este corect pt ca functia statica f utilizeaza variabila nestatica x

1. Recapitulare curs 3

Membrii statici ai unei clase

Spuneţi dacă programul de mai jos este corect. În caz afirmativ, spuneţi ce afisează, în caz negativ spuneţi de ce nu este corect.

```
#include <iostream.h>
class A
{ static int x;
  public: A(int i=0) {x=i; }
  int get_x() { return x; }
  int& set_x(int i) { x=i;}
  A operator=(A a1) { set_x(a1.get_x()); return a1;}
};
int main()
{ A a(212), b;
  cout<<(b=a).get_x();
  return 0;
}
```

R: Nu este corect pentru ca variabila statica x nu e alocata si initializata.

1. Recapitulare curs 3

Membrii statici ai unei clase

Spuneţi dacă programul de mai jos este corect. În caz afirmativ, spuneţi ce afisează, în caz negativ spuneţi de ce nu este corect.

```
#include<iostream.h>
class cls {
      static int i;
      int j;

      public:
      cls(int x=7) { j=x; }
      static int imp(int k){ cls a; return i+k+a.j; } };

int cls::i;

int main()
      { int k=5;
      cout<<cls::imp(k);
return 0;
}
```

R: Corect pt ca functia statica f utilizeaza variabila statica I, parametrul k si obiectul local a.

2. Supraîncărcarea funcțiilor

Supraincarcarea functiilor si a operatorilor (overloading) ofera o baza importanta pentru polimorfismul la compilare.

Supraincarcarea functiilor = procesul de folosire a aceluiasi nume pentru 2 sau mai multe functii.

Fiecare redefinire trebuie sa foloseasca sau tipuri diferite sau numar diferiti de parametri.

Compilatorul alege in functie de aceste diferente.

2. Supraîncărcarea funcțiilor

Observatii:

1) Nu pot fi supraincarcate doua functii care difera **doar prin tipul returnat** => eroare la compilare

Exemplu:
```
int f(int x);
float f(int x);
```

2) Atentie la declaratii de functii care par uneori ca difera, dar nu este asa:

Exemplu:
```
void f(int *p);
void f(int p[ ]);
```

Exemplu:
```
void f(int x);
void f(int& x);
```

2. Supraîncărcarea funcțiilor

Ambiguitati:

- compilatorul nu este capabil sa aleaga intre 2 sau mai multe functii supraincarcate.

- cauza principala a ambiguitatii = conversia automata a tipului in C++.

Exemple:
1) int myfunc(double d); // ...
cout << myfunc('c'); // not an error, conversion applied

2) eroare determinata de apel, nu de supraincarcarea efectiva
float myfunc(float I) {return I;}
double myfunc(double I) {return -i;}

Apel: cout<<myfunc(10.5); // neambiguu, apeleaza double
cout<<myfunc(10); //ambiguitate

Obs: valorile reale (in virgula mobila) constante, in C++, sunt considerate by default "double", daca nu sunt mentionate explicit "float".

2. Supraîncărcarea funcțiilor

Ambiguitati:

Exemple:
3)
```
char myfunc(unsigned char ch) {return ch-1;}
char myfunc(char ch){return ch+1;}
```

Apel:
```
cout << myfunc('c'); // this calls myfunc(char)
cout << myfunc(88) << " "; // ambiguous
```

***Obs: "unsigned char" si "char" nu sunt implicit ambigue, dar apelul myfunc(88) e ambiguu*.**

2. Supraîncărcarea funcțiilor

Ambiguitati:

Exemple: - folosirea argumentelor implicite in supraincarcare
4)

```
#include <iostream>
using namespace std;
int myfunc(int i);
int myfunc(int i, int j=1);
int main()
{
cout << myfunc(4, 5) << " "; // unambiguous
cout << myfunc(10); // ambiguous
return 0;
}
int myfunc(int I) { return i; }
int myfunc(int i, int j) { return i*j; }
```

2. Supraîncărcarea funcțiilor

Supraîncărcarea constructorilor

In afara de rolul special de initializare, constructorii nu difera de alte functii, deci au si proprietatile de supraincarcare.

Scop:
1) pentru o mai mare flexibilitate a programului;

2) pentru a permite crearea atat a obiectelor initializate cat si a celor neinitializate;

3) pentru a defini “copy constructor”

3. Copy constructor

Transmiterea obiectelor catre functii:

- daca este prin valoare, atunci se copiaza starea unui obiect in alt obiect bit cu bit (bitwise copy);
- probleme apar daca obiectul trebuie sa aloce memorie datelor sale membre;
- ***apelam constructorul cand cream obiectul și apelam de 2 ori destructorul.***

Mai concret:

- la apel de funcție nu se apeleaza constructorul “normal” ci se creeaza o copie a parametrului efectiv folosind constructorul de copiere;
- un asemenea constructor defineste cum se copiaza un obiect;
- se poate defini explicit de catre programator, dar exista implicit în C++ automat;
- trebuie rescris in situatiile în care se utilizeaza alocarea dinamica pentru datele membre.

3. Copy constructor

Cazuri de utilizare:

Initializare explicita:
MyClass B = A;
MyClass B (A);

Apel de functie cu obiect ca parametru:
    void f(MyClass X) {…}

Apel de functie cu obiect ca variabila de intoarcere:
    MyClass f() {MyClass obiect; … return obiect;}.
    MyClass x = f();

Copierea se poate face și prin operatorul = (*detalii mai târziu*).

3. Copy constructor

Sintaxa:

```
classname (const classname &ob) {
// body of constructor
}
```

- ob este obiectul din care se copiaza;
- pot exista mai multi parametri, dar:
  - trebuie sa definim valori implicite pentru ei;
  - obiectul din care se copiaza trebuie declarat primul.

Obs: constructorul de copiere este folosit doar la initializari;

```
classname a,b;
b = a;
```

- nu este initializare, este copiere de stare.

3. Copy constructor

Functii care intorc obiecte

Un obiect temporar este creat automat pentru a tine informatiile din obiectul de întors.

Dupa ce valoarea a fost intoarsa, acest obiect este distrus.

Probleme cu memoria dinamica: solutie polimorfism pe = si pe constructorul de copiere.

Copierea prin operatorul =

trebuie sa fie de acelasi tip (aceeasi clasa)

3. Copy constructor

Functii care intorc obiecte

Exemplu:

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

class myclass {
   int i;
public:
   void set_i(int n) { i=n; }
   int get_i() { return i; }
};

myclass f() // return object of type myclass
{
   myclass x;
   x.set_i(1);
   return x;
}

int main()
{
   myclass o;
   o = f();
   cout << o.get_i() << "\n";
   return 0;
}
```

4. Supraincarcarea operatorilor

- una din cele mai importante caracteristici ale C++;
- majoritatea operatorilor pot fi supraincarcati, similar ca la functii;

Obs: NU SE POT SUPRAINCARCA
- " . " (acces la membru);
- ".*" (acces la membru prin pointer);
- "::" (rezolutie de scop);
- "?:" (operatorul ternar).

- se face definind ***o functie operator;***
- 2 modalitati: - ca functie membra a clasei sau ca functie prietena a clasei.

Obs: NU SE POATE MODIFICA PRIN SUPRAINCARCARE:
- pluralitatea operatorului (numarul de argumente);
- precedenta si asociativitatea operatorului.

Obs: NU SE POT DA VALORI IMPLICITE (exceptie operatorul " ( ) ").
Obs: Operatorii, cu exceptia "=", se mostenesc in clasele derivate.
RECOMANDARE: pastrati intelesul operatorilor respectivi.

4. Supraincarcarea operatorilor

Crearea unei functii membre:

- ***sintaxa generala***:

```
ret-type class-name::operator#(arg-list)
{
    // operations
}
```

Unde:

: operatorul supraincarcat (+ - * / ++ -- = , etc.);

ret-type, in general, este tipul clasei;

pentru operatori unari arg-list este vida, pentru binari arg-list contine un element.

4. Supraincarcarea operatorilor

Ca functii membre

Obs. Operatorii:

" = " (atribuire);

" [ ] " (indexare);

" ( ) " (apel de functie);

" → " (acces membru de tip pointer);

Pot fi definiti DOAR CU FUNCTII MEMBRE NESTATICE.

4. Supraincarcarea operatorilor

Ca functii membre

Exemplu – supraincarcare operator +

```
class loc {
      int longitude, latitude;
public:
      loc() {    }
      loc(int lg, int lt) {
            longitude = lg;
            latitude = lt;
      }
void show() {
cout << longitude << " ";
cout << latitude << "\n";  }
loc operator+(loc op2);
};
```

```
// Overload + for loc.
loc loc::operator+(loc op2) {
      loc temp;
      temp.longitude = op2.longitude + longitude;
      temp.latitude = op2.latitude + latitude;
      return temp;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30);
ob1.show(); // displays 10 20
ob2.show(); // displays 5 30
ob1 = ob1 + ob2;
ob1.show(); // displays 15 50
return 0;
}
```

4. Supraincarcarea operatorilor

Ca functii membre

Exemplu – supraincarcare operator + (Observatii)

+ este operator binar, dar functia **loc operator+(loc op2);** are un singur parametru, pentru ca celalalt este retinut in “this”;

- obiectul din stanga face apelul la functia operator;

ob1 = ob1 + ob2; e posibila pentru ca se intoarce acelasi tip de date in operator;

- posibil: (ob1 + ob2).show();

- se genereaza un obiect temporar (constructor de copiere).

4. Supraincarcarea operatorilor

Ca functii membre

Exemple – supraincarcare operatori -, =, ++

```
class loc {
int longitude, latitude;
public:
loc() { } // needed to construct temporaries
loc(int lg, int lt) { longitude = lg; latitude = lt; }
void show() { cout << longitude << " "; cout << latitude << "\n"; }

loc operator+(loc op2);
loc operator-(loc op2);
loc operator=(loc op2);
loc operator++();

};
```

4. Supraincarcarea operatorilor

Ca functii membre

Exemple – supraincarcare operatori -, =, ++ (prefixat)

```
loc loc::operator+(loc op2) {
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp; }

loc loc::operator-(loc op2) {
loc temp; // atentie la ordinea operanzilor
temp.longitude = longitude - op2.longitude;
temp.latitude = latitude - op2.latitude;
return temp;}
```

```
loc loc::operator=(loc op2)
{
      longitude = op2.longitude;
      latitude = op2.latitude;
return *this; // obiectul care apeleaza
}

loc loc::operator++()
{
      longitude++;
      latitude++;
return *this;
}
```

4. Supraincarcarea operatorilor

Ca functii membre

Obs:

- in operatorul – e importanta ordinea operanzilor;
- operatorii = si ++ returneaza *this pentru ca lucreaza pe variabilele de instanta;
- se pot face atribuiri multiple;

Supraincarcarea operatorilor de incrementare prefixat si postfixat

Sintaxa:

```
// Prefix increment
type operator++( ) {
// body of prefix operator
}
```

```
// Postfix increment
type operator++(int x) {
// body of postfix operator
}
```

Obs: Analog - -

- pentru postfix: definim un parametru int “dummy”

4. Supraincarcarea operatorilor

Ca functii membre

Supraincarcarea operatorilor prescurtati +=, /= , etc.

Sintaxa:

```
loc loc::operator+=(loc op2)
{
      longitude = op2.longitude + longitude;
      latitude = op2.latitude + latitude;
return *this;
}
```

Obs: se combina operatorul de atribuire “=” cu alt operator.

4. Supraincarcarea operatorilor

Ca functii friend

- functia friend are acces la membrii privati si protejati ai clasei;
- nu are pointerul “this”;
- e nevoie de declararea tuturor operanzilor;
- primul parametru este operandul din stanga, al doilea parametru este operandul din dreapta.

Diferente supraincarcarea prin membri sau prieteni

- in general, daca nu exista diferente, e indicat sa se utilizeze supraincarcarea prin functii membre;
- daca conteaza ordinea/pozitia operanzilor, atunci trebuie sa se utilizeze functii friend;

Expl: obiect + int diferit de int + obiect => supraincarcarea a 2 functii operator+ cu un parametru obiect si un parametru int.

Numarul de parametri dintr-o functie operator implementata ca functie friend este cu 1 mai mare decat numarul de parametri dintr-un functie operator implementata ca functie membru (datorita lui ***“this”***).

4. Supraincarcarea operatorilor

Ca functii friend

Supraincarcarea operatorului +

```
class loc {
int longitude, latitude;
public:
loc() { } // needed to construct temporaries
loc(int lg, int lt) { longitude = lg; latitude = lt; }
void show() { cout << longitude << " "; cout << latitude << "\n"; }

friend loc operator+(loc op1, loc op2); // now a friend
};

loc operator+(loc op1, loc op2) {
loc temp;
      temp.longitude = op1.longitude + op2.longitude;
      temp.latitude = op1.latitude + op2.latitude;
return temp;
}
```

4. Supraincarcarea operatorilor

Ca functii friend

Supraincarcarea operatorilor unari

- pentru ++, -- folosim referinta pentru a transmite operandul pentru ca trebuie sa se modifice si nu avem pointerul this;
- apel prin valoare: primim o copie a obiectului si nu putem modifica operandul (ci doar copia);

```
class loc {
int longitude, latitude;
public:
…..
friend loc operator++(loc &op);
friend loc operator--(loc &op);
};
```

```
loc operator++(loc &op) {
      op.longitude++;
      op.latitude++;
return op;
}

loc operator--(loc &op) {
      op.longitude--;
      op.latitude--;
return op;
}
```

4. Supraincarcarea operatorilor

Ca functii friend

Supraincarcarea operatorului + pentru a putea executa ob + ob, int + ob = flexibilitate

```
class loc {
int longitude, latitude;
public:
...
friend loc operator+(loc op1, loc op2);
friend loc operator+(int op1, loc op2);
};
loc operator+(loc op1, loc op2) {
loc temp;
      temp.longitude = op1.longitude + op2.longitude;
      temp.latitude = op1.latitude + op2.latitude;
return temp; }
```

```
loc operator+(int op1, loc op2) {
loc temp;
      temp.longitude = op1 + op2.longitude;
      temp.latitude = op1 + op2.latitude;
return temp; }
```

4. Supraincarcarea operatorilor

Supraincarcarea operatorului de conversie (cast)

Exemplu:

```
int main()
{
    double x = 1.23;
    int i = (int)x;   // conversie catre int ---> parte intreaga
    cout << "i = " << i << endl;
}
```

4. Supraincarcarea operatorilor

Supraincarcarea operatorului de conversie (cast)

Sintaxa: **operator tip()**

Exemplu - conversie catre un tip predefinit

```
class Test
{
    int v[2];
    public:
    Test(){v[0] = 100; v[1] = 200;}
    operator int();
};

Test::operator int ()
{
    return v[0]+v[1];
}
```

```
int main()
{
    Test obiect;
    int x = obiect;
    cout<<x;
    return 0;
}
```

4. Supraincarcarea operatorilor

Supraincarcarea operatorului de conversie (cast)

Exemplu - conversie catre un tip definit de utilizator

```
class Test {
    public:
    Test(){}
};

class Test2 {
    public:
    operator Test();
};

Test2::operator Test ()
{
    return Test();
}
```

```
   void afisare(Test t)
{cout<<"dupa conversie";}

int main()
{
     Test2 x;
    afisare(x);;
    return 0;
}
```

4. Supraincarcarea operatorilor

Supraincarcarea unor operatori speciali: “ [ ] “

- nu poate fi supraincarcat ca functie friend;

- este considerat un operator binar, sub forma operator [ ] ( );

- obiect [5] <=> obiect.operator [ ] (5);

- sintaxa:

```
type class-name::operator[ ](int i)
{
      // . . .
}
```

4. Supraincarcarea operatorilor

Supraincarcarea unor operatori speciali: " [ ] "

```
class atype {
      int a[3];
public:
      atype(int i, int j, int k) {    a[0] = i; a[1] = j;  a[2] = k;}

int operator[ ] (int i) { return a[i]; }

};

int main() {
atype ob(1, 2, 3);
cout << ob[1]; // displays 2
return 0;
}
```

4. Supraincarcarea operatorilor

Supraincarcarea unor operatori speciali: " [ ] "

Daca trebuie sa fie la stanga unei atribuiri, atunci operatorul trebuie sa intoarca referinta.

```
class atype {
      int a[3];
public:
      atype(int i, int j, int k) {    a[0] = i; a[1] = j;  a[2] = k;}
int& operator[ ] (int i) { return a[i]; }
};

int main()
{
      atype ob(1, 2, 3);
      cout << ob[1]; // displays 2
      cout << " ";
      ob[1] = 25; // [ ] on left of =
      cout << ob[1]; // now displays 25
return 0;
}
```

4. Supraincarcarea operatorilor

Supraincarcarea unor operatori speciali: “ ( ) “

- nu se creaza un nou mod de apel de functie ci se defineste un mod de a chema functii cu numar variabil de parametri.

Exemplu:

```
double operator( )(int a, float f, char *s);
O(10, 23.34, "hi");
```

echivalent cu O.operator( )(10, 23.34, "hi");

4. Supraincarcarea operatorilor

Supraincarcarea unor operatori speciali: “ ( ) “
- functie membra, nestatica

```
class loc {
int longitude, latitude;
public:
…..
loc operator( )(int i, int j);
};

// Overload ( ) for loc.
loc loc::operator()(int i, int j)
{
      longitude = i;
      latitude = j;
      return *this;
}
//Apel loc ob1(10, 20);
ob1(7,8); // se executa de sine statator
```

4. Supraincarcarea operatorilor

Supraincarcarea unor operatori speciali: “ → “

- functie membra, nestatica

- operator unar

- obiect->element // obiectul genereaza apelul

- element trebuie sa fie accesibil

- intoarce un pointer catre un obiect din clasa

4. Supraincarcarea operatorilor

Supraincarcarea unor operatori speciali: “ → “

```
class myclass {
public:
   int i;
myclass *operator->() {return this;}
};

int main()
{
   myclass ob;
   ob->i = 10; // same as ob.i
   cout << ob.i << " " << ob->i;
   return 0;
}
```

4. Supraincarcarea operatorilor

Supraincarcarea unor operatori speciali: “ → “

- functie membra, nestatica
- operator unar
- obiect->element // obiectul genereaza apelul
- element trebuie sa fie accesibil
- intoarce un pointer catre un obiect din clasa

```
class myclass {
public:
   int i;
myclass *operator->() {return this;}
};

int main()
{
   myclass ob;
   ob->i = 10; // same as ob.i
   cout << ob.i << " " << ob->i;
   return 0;
}
```

4. Supraincarcarea operatorilor

Supraincarcarea altor operatori NEW si DELETE

- supraincarcare op. de folosire memorie in mod dinamic pentru cazuri speciale
- size_t: predefinit
- pentru new: constructorul este chemat automat
- pentru delete: destructorul este chemat automat
- supraincarcare la nivel de clasa sau globala

4. Supraincarcarea operatorilor

Supraincarcarea altor operatori NEW si DELETE

```
void *operator new(size_t size)
{
/* Perform allocation.Throw bad_alloc on failure. Constructor called automatically.*/
return pointer_to_memory;
}

// Delete an object.
void operator delete(void *p)
{
/* Free memory pointed to by p.
Destructor called automatically. */
}
```

4. Supraincarcarea operatorilor

Supraincarcarea altor operatori NEW si DELETE

```
class loc {
  int longitude, latitude;
public:
   loc() {}
   loc(int lg, int lt) {longitude = lg; latitude = lt;}
   void show() { cout << longitude << " ";
                 cout << latitude << "\n";}
   void *operator new(size_t size);
   void operator delete(void *p);
};
```

```
// new overloaded relative to loc.
void *loc::operator new(size_t size){
   void *p;
   cout << "In overloaded new.\n";
   p = malloc(size);
   if(!p) { bad_alloc ba; throw ba; }
return p;
}
```

4. Supraincarcarea operatorilor

Supraincarcarea altor operatori NEW si DELETE

```
class loc {
  int longitude, latitude;
public:
   loc() {}
   loc(int lg, int lt) {longitude = lg; latitude = lt;}
   void show() { cout << longitude << " ";
                 cout << latitude << "\n";}
   void *operator new(size_t size);
   void operator delete(void *p);
};
```

```
// delete overloaded relative to loc.
void loc::operator delete(void *p){
   cout << "In overloaded delete.\n";
   free(p);
}
```

4. Supraincarcarea operatorilor

Supraincarcarea altor operatori NEW si DELETE

```
class loc {
  int longitude, latitude;
public:
   loc() {}
   loc(int lg, int lt) {longitude = lg; latitude = lt;}
   void show() { cout << longitude << " ";
                 cout << latitude << "\n";}
   void *operator new(size_t size);
   void operator delete(void *p);
};
```

```
int main(){
   loc *p1, *p2;
   try {p1 = new loc (10, 20);
   } catch (bad_alloc xa) {
         cout << "Allocation error for p1.\n";
         return 1;}
  try {p2 = new loc (-10, -20);
  } catch (bad_alloc xa) {
         cout << "Allocation error for p2.\n";
         return 1;}
  p1->show();
  p2->show();
  delete p1;
  delete p2;
  return 0;
}
```

- In overloaded new.
- In overloaded new.
- 10 20
- -10 -20
- In overloaded delete.
- In overloaded delete.

4. Supraincarcarea operatorilor

Supraincarcarea altor operatori NEW si DELETE

- daca new sau delete sunt folositi pentru alt tip de date in program, versiunile originale sunt folosite

- se poate face overload pe new si delete la nivel global

  – se declara in afara oricarei clase

  – pentru new/delete definiti si global si in clasa, cel din clasa e folosit pentru elemente de tipul clasei, si in rest e folosit cel redefinit global

4. Supraincarcarea operatorilor

Cum alegem modul de supraincarcare a operatorilor?

Sursa: Rob Murray, C++ Strategies & Tactics, Addison-Wesley, 1993, page 47

Operator	Recomandare de implementare
Toti operatorii unari	Membru
=,[ ], ( ), ->, ->*	TREBUIE SA FIE MEMBRE
+=, -=, *=, /=, %=, ^=, &=, |=, >>=, <<=	Membru
Toti operatorii binari	Ne - membru

5. Pointeri catre obiecte

- putem avea Pointeri catre obiecte la fel cum putem avea pointeri catre alte tipuri de variabile.
- retin adresa unui obiect
- acceseaza campurile obiectului folosind operatorul “->”

Expl:

```
class cl {
      int i;
public:
      cl(int j) { i=j; }
      int get_i() { return i; }
};
int main() {
cl ob(88);
cl *p  = &ob; // get address of ob
cout << p->get_i();
return 0; }
```

5. Pointeri catre obiecte

- aritmetica pointerilor - relativa la tipul de baza al acestora

Expl:

```
class cl {
      int i;
public:
      cl(int j) { i=j; }
      int get_i() { return i; }
};
int main() {
cl ob[2] = {10,20} ;
cl *p  = ob; p++;
cout << p->get_i(); // afiseaza 20
return 0; }
```

5. Pointeri catre obiecte

Pointeri de verificare a tipului in C++

In C++ se poate atribui un pointer altui pointer, doar daca tipurile lor de baza sunt compatibile.

Exemplu:

```
int *p;
float *q;
p=q; //eroare, nepotrivire de tipuri
```

Eliminare eroare: - schimbarea de tip (type casting) dar nu se mai aplica verificarile de tip automate facute de C++

5. Pointeri catre obiecte

Pointeri catre tipuri derivate

In general, un pointer de un anumit tip nu poate indica spre un obiect de tip diferit. Exceptie - clasele derivate, pentru ca functioneaza ca si clasa de baza plus alte detalii.

Utilizati in polimorfism la executie (functii virtuale).

```
class Baza{ public: int x;};
class Derivata: public Baza {public: int y;};

int main()
{
  Baza *b;
  Derivata d;
  b = &d;
  cout<<b->x; // acces la membrul importat din baza
  cout<<b->y; // nu are acces pentru ca membrul este propriu derivatei
    return 0;
}
```

5. Pointeri catre obiecte

Pointeri catre tipuri derivate

Rezolvare - conversie de tip

```
class Baza{ public: int x;};
class Derivata: public Baza {public: int y;};

int main()
{
  Baza *b;
  Derivata d;
  b = &d;
  cout<<b->x; // acces la membrul importat din baza
  cout<<((Derivata*)b)->y;
    return 0;
}
```

5. Pointeri catre obiecte

Pointeri catre tipuri derivate

Un pointer catre clasa de baza, chiar daca indica spre derivata, la incrementare cauta un alt obiect de tip baza

```
class Baza{ public: int x;  } };
class Derivata: public Baza { };

int main()
{
  Baza *b;
  Derivata d[2];
  b = d;
  d[0].x = 10; d[1].x = 20;
  cout<<b->x; // afiseaza 10
     b++;
  cout<<b->x; // nu afiseaza 20 ci o valoare gresita
  return 0;
}
```

5. Pointeri catre obiecte

Pointeri catre membrii clasei

Sunt folositi rar.

Au asociat, **pe lângă tipul datei sau funcţiei, şi tipul clasă respectiv**.

Un pointer către un membru al unei clase nu se asociază obiectelor clasei respective, **ci clasei**, el conţinând ca informaţie nu adresa membrului ci **deplasarea lui în cadrul clasei**.

Sintaxa:

```
tip nume_clasa::*pointer_membru;
```

Atribuirea:

```
pointer_membru=&nume_clasa::membru;
```

5. Pointeri catre obiecte

Pointeri catre membrii clasei

Operatorii destinaţi accesului la un membru prin pointeri sunt:

- operatorul .* dacă se specifică un obiect al clasei (in loc de “.”)
- operatorul ->* dacă se specifică un pointer de obiect (in loc de “->”)

Sintaxa folosită pentru referirea membrului este:

obiect.*pointer_membru

sau

pointer_obiect->*pointer membru

În cazul pointerilor la funcţii membre nu este obligatorie utilizarea operatorului &, deci declaraţiile:
p_func=&pozitie::deplasare; şi **p_func=pozitie::deplasare;**
sunt echivalente.

5. Pointeri catre obiecte

Pointeri catre membrii clasei

```
class cl { public:
      cl(int i) { val=i; }
      int val;
      int double_val() { return val+val; } };

int main() {
int cl::*data; // data member pointer
int (cl::*func)(); // function member pointer
cl ob1(1), ob2(2); // create objects
data = &cl::val; // get offset of val
func = &cl::double_val; // get offset of double_val()

cout << "Here are values: "; cout << ob1.*data << " " << ob2.*data << "\n";
cout << "Here they are doubled: ";
cout << (ob1.*func)() << " "<< (ob2.*func)() << "\n";
return 0; }
```

5. Referinte la obiecte

- in functii: transmiterea prin referinta nu creaza noi obiecte temporare
- se lucreaza direct pe obiectul transmis
- copy-constructorul si destructorul nu mai sunt apelate.

Atentie: obiectul referit poate fi modificat, deci, ideal, referinte constante.

- Intoarcerea din functie a unei referinte.
- permite atribuiri catre apel de functie

5. Referinte la obiecte

Referinte catre clasele derivate

- putem avea referinte definite catre clasa de baza si apelata functia cu un obiect din clasa derivata;
- exact la la pointeri

```
class Baza{ public: int x;
void functie() {cout<<"Baza";}};

class Derivata: public Baza {public: int y;};

int main()
{
  Derivata d;
  Baza &b = d;
  b.functie();
  return 0;
}
```

6. Membrii constanți ai unei clase in C++

"The concept of constant (expressed by the const keyword) was created to allow the programmer to draw a line between what changes and what doesn't." (Bruce Eckel)

#define MAX 100

- procesor, nu ocupa memorie, poate fi introdusa intr-un header pentru ca e aceeasi pentru tot programul
- nu face verificare de tip, poate introduce probleme.

const int MAX = 100;

6. Membrii constanți ai unei clase in C++

```
/// Using const for safety

#include <iostream>
using namespace std;
      const int i = 100; // Typical constant
      const int j = i + 10; // Value from const expr
      long address = (long)&j; // Forces storage
      char buf[j + 10]; // Still a const expression

int main()
{
    cout << "type a character & CR:";
      const char c = cin.get(); // Can't change
      const char c2 = c + 'a';
    cout << c2;
// ...
}
```

6. Membrii constanți ai unei clase in C++

Pointerii si const

The const specifier binds to the thing it is "closest to."

Pointer la valori const

const int * u; sau **int const * u;** // u este un pointer care indica catre o constanta intreaga

Pointeri constanti

int d = 1;

int* const w = &d; // w este o constanta de tip pointer care retine doar adresa lui d.

Pointeri constanti la valori constante

int d = 1;

const int* const x = &d; // (1)

int const* const x2 = &d; // (2)

6. Membrii constanți ai unei clase in C++

Pointerii si const

Atribuirea si verificarea tipului

```
int d = 1;
const int e = 2;
```

- permisa asignarea unei adrese a unui non const la un pointer const
- NU este permisa asignarea unei adrese a unei constante unui pointer neconst, intrucat compilatorul “intelege” ca s-ar putea modifica valoarea respectiva printr-un pointer.

```
int* u = &d; // OK -- d not const

//! int* v = &e; // Illegal -- e const

int* w = (int*)&e; // Legal but bad practice
```

6. Membrii constanți ai unei clase in C++

Argumentele functiilor si returnarea valorilor

Transmiterea prin parametru constant

```
void f1(const int i) {
i++; // Illegal -- compile-time error
}
```

- se “promite” nemodificarea valorii transmise
- argument transmis fara referinta, deci se face o copie, se incearca incrementarea copiei, deci valoarea transmisa nu se modifica

Mai elegant

```
void f2(int ic) {
    const int& i = ic;
    i++; // Illegal -- compile-time error
}
```

6. Membrii constanți ai unei clase in C++

Argumentele functiilor si returnarea valorilor

Returnarea unei valori constante

- nu se vad diferente daca se aplica tipurilor predefinite

```
// Returning consts by value
      // has no meaning for built-in types
int f3() { return 1; }
const int f4() { return 1; }

int main() {
 const int j = f3(); // Works fine
 int k = f4(); // But this works fine too!
}
```

- recomandata la returnarea tipurilor definite de utilizator; daca se returneaza cu “const”, atunci nu poate fi “l-value”

6. Membrii constanți ai unei clase in C++

Argumentele functiilor si returnarea valorilor

```
class X {  int i;
public:
 X(int ii = 0){ i = ii; }
 void modify(){ i++; }
};

X f5() { return X();}
const X f6() { return X();}
void f7(X& x) { x.modify(); /* Pass by non-const reference */}

int main() {
 f5() = X(1); // OK -- non-const return value
 f5().modify(); // OK
// Causes compile-time errors:
//! f7(f5());
//! f6() = X(1);
//! f6().modify();
//! f7(f6());}
```

Perspective

Cursul 6:

1 Tratarea excepțiilor in C++.

2. Proiectarea descendenta a claselor. Mostenirea in C++.
- Controlul accesului la clasa de bază.
- Constructori, destructori şi moştenire.
- Redefinirea membrilor unei clase de bază într-o clasa derivată.
-. Declaraţii de acces.