Cursul 5 - CLASE în C++, continuare

- Copiere şi atribuire
- Obiecte membru, ordinea de apel a constructorilor
- Date membru calificate **`const`**, membri referință, inițializare
- Membri calificați **`static`**
- Metode ce întorc referințe la membri din secțiunea **`private`**
- Spațiu de nume, domeniu de vizibilitate
- Pointeri la membri
- Operatorii [] şi ()

CLASE: Copiere şi asignare

- Atenție la clasele ce conțin pointeri! Folosirea constructorului de copiere implicit poate conduce la erori. Soluția: implementarea unui constructor de copiere utilizator
  - Constructor de copiere

```
X::X(const X& x){
// Implementare copiere
}
```

  - Supraîncărcare operator de atribuire:

```
X::X& operator=(const X& x){
// Implementare asignare
}
```

```
#include <iostream.h>
class TipTablou{
    const char* s;
        //
};
class Tablou{
public:
    Tablou(int s=10){
        p = new TipTablou[marime = s];
        cout <<++i<< " Tablou()\n";}
    ~Tablou() {
        delete[] p; cout <<i--<<  " ~Tablou()\n";}
    Tablou(const Tablou& );  // constructor de copiere
    Tablou& operator=(const Tablou&); // operator atribuire
private:
    TipTablou* p;
    int marime;
    static int i;
};
```

```
Tablou::Tablou(const Tablou& t){
    p = new TipTablou[marime = t.marime];
    for (int k = 0; k < marime ; k++)
        p[k] = t.p[k];
    cout << "Copiere ob: "<< i << endl;
}

Tablou& Tablou::operator=(const Tablou& t){
    if(this != &t){ // evitare asignare t = t
        delete[] p;
        p = new TipTablou[marime = t.marime];
        for (int k = 0; k < marime ; k++)
                p[k] = t.p[k];
    }
    cout << "s-a asignat ob: " << i << endl;
    return *this;
}
```

CLASE: Copiere şi asignare

- Constructorul de copiere se apelează
  - La declararea unui obiect cu inițializare
  - La evaluarea expresiilor pentru crearea de obiecte temporare
  - La transmiterea parametrilor prin valoare
  - La returnarea din funcții prin valoare

```
#include <iostream>
using namespace std;
class Clasa1{
    int i;
public:
    Clasa1() : i(13) {}
    Clasa1(const Clasa1 &a) : i(a.i+1) {cout << " copie ";}
    int get_i() {return i;}
};
Clasa1 f(Clasa1 a){
    cout << a.get_i() << "\n";
    return a;
}
int main(){
    Clasa1 a;
    Clasa1 a1=a;// Apel copiere
    cout << a1.get_i() << endl;
    a1 = f(a1);// Apel copiere de 2 ori
    cout << a1.get_i() << endl;
    return 0;
}
/* copie 14 copie 15 copie 16  */
```

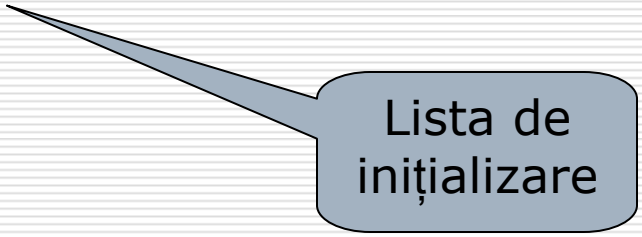
CLASE – Obiecte membru

- Argumentele pentru constructorii membrilor sunt date în lista de inițializare:

  Clasa::Clasa(par) : *Obiect1(par), Obiect2(par)…*{…}

```
class Club{
public:
     Club(const string& n, Data df);
private:
     string nume;
     Tablou membri;
     Tablou membri_fondatori;
     Data data_fond;
};
Club::Club(const string& n, Data df)
       :nume(n), membri(), membri_fond(),data_fond(df)
{
     // Implementare constructor

}    // apelurile fara parametri pot lipsi
```



- Ordinea de apel a constructorilor : ordinea declarațiilor în clasă

CLASE –membri **const**, membri referință

- Inițializarea membrilor:
  - obiecte ale claselor fără constructor implicit,
  - calificați **const**,
  - referință

  se face numai prin liste de inițializare:

```
class X{
public:
    X(int ii, const string& n, Data d, Club& c)
            :i(ii), c(n, d), pc(c) {}
private:
    const int i; Club c; Club& pc;
};
class Punct{
public:
    Punct(int vx = 0, int vy = 0):x(vx), y(vy) {};
private:
    int x; int y;
};
```

CLASE –membri **const**

```
// FILE: ConstMember.cpp,  Membri const intr-o clasa
class X {
  const int i ;
public:
  X(int ii);
  int f() const;
  int g() ;
};

X::X(int ii) : i(ii) {}
int X::f() const { return i;}
int X::g()       { return i++; }

void main() {
  X x1(10);
  const X x2(20);
  x1.f(); // OK
  x2.f(); // OK
  x1.g(); // OK
  x2.g(); // KO ! g nu-i const!
}
```

CLASE –membri **mutable**

- Date membru calificate **mutable**: pot fi modificate de metode **const**

```
class Z {
  int i;
  mutable int j;// j poate fi modificat si de metode const
public:
  Z();
  void f() const;
};
Z::Z() : i(0), j(0) {}
void Z::f() const {
    //i++; // Eroare -- functie membru const
    j++; // OK: j este mutable
}

void main() {
  const Z zz;
  zz.f(); // f schimba atributul j al obiectului zz
}
```

CLASE –membri calificați **`static`**

- Dată membru calificată **`static`**: este independentă de orice instantă a clasei. Toate obiectele din clasa respectivă partajează aceeasi zonă de memorie declarată **`static`**

- O dată membru **`static`** poate fi privită ca fiind globală în domeniul de vizibilitate al clasei: este accesibilă de către toate instantele clasei

- Se pot număra instantele unei clase cu un membru declarat **`static`**

- Definiția unui membru **`static`** trebuie să apară, o singură dată, undeva în afara clasei dar în acelasi fişier, fără a mai folosi calificatorul **`static`**:

```
class A{
    //...
    static tip v;
    //...
};
tip A::v = expresie_constanta;
```

CLASE –membri calificați **static**

```
int x = 100;
class CStatic {
//…
  static int x;
  static int y;
};
int CStatic::x = 1;
int CStatic::y = ::x + 1;
```

CLASE –membri calificați **static**

```
class X{
public:
    X(int x=0):i(x){};
private:
    int i;
};
class Stat {
public:
    //…
private:
  static X x, Tablou1[];
  static const X y, Tablou2[];
};

X Stat::x(100);

X Stat::Tablou1[] = {X(1), X(2), X(3), X(4)};

const X Stat::y(100);

const X Stat::Tablou2[] = {X(11), X(22), X(33), X(44), X(55)};
```

Numărarea instanțelor create

```
class A {
public:
    A() { nr_instante++; // ... }
    ~A();
    int get_nr_inst()
    { return nr_instante; }
    //...
private:
    static int nr_instante;
    double x;
};
int A::nr_instante = 0;
Void main(){
    A a;
    A b;
    cout << a.get_nr_inst();  // afiseaza 2
    cout << b.get_nr_inst();  // afiseaza 2
}
```

CLASE –membri calificați **`static`**

- O funcție membru declarată **`static`**
  - funcție asociată clasei: se poate apela şi prin **`X::f()`**
  - nu poate utiliza pointerul **`this`**
  - poate accesa doar membrii statici ai clasei
  - o funcție calificată static poate fi apelată şi ca mesaj către obiect; nu este indicat
  - Se poate folosi mecanismul pentru a construi o clasă ce are un unic obiect ca membru static şi nu pot fi instanțiate alte obiecte

CLASE –membri calificați **static**

```
class X {
  int i;
  static int j;
public:
  X(int ii = 0) : i(ii) {
    j = i; // j poate fi accesat de metode nestatice
  }
  int val() const { return i; }
  static int incr() {
    i++; // Eroare: incr() este static, i nu este static
    return ++j; // OK, j este static
  }
  static int f() {
    val(); // Eroare: f() este static iar val() nu
    return incr(); // OK, incr() este static
  }
};
```

O clasă responsabilă cu efectuarea de operații

```
class Operatii {
public:
    static int add(int, int);
    static int mult(int, int);
    static int getAddCounter();
    static int getMultCounter();
private:
    static int addCounter;
    static int multCounter;
};
//  implementarea operatiei de adunare
int Operatii::add(int x, int y){
    addCounter++;
    return x+y;
}
//  utilizare
int a, b;
a = Operatii::add(2, 3);
b = Operatii::mult(4, 6);
```

Data curentă

```
class Data {
public:
   //. . .
   static Data azi();
private:
   //. . .
   static int an_curent();
   //. . .
};
//...
cout << Data::azi(); // OK
cout << Data::an_curent(); // nu-i OK!
```

O clasă cu obiect unic

```
class Unic {
public:
  static Unic* instanta() { return &e; }
  int get_i() const { return i; }
private:
  static Unic e;
  int i;
  Unic(int ii) : i(ii) {}// anuleaza crearea de ob
  Unic(const Unic&); // anuleaza copierea
};
Unic Unic::e(99);
int main() {
  Unic x(1); // Eroare nu se pot crea alte obiecte
  // Se poate accesa unicul obiect:
  cout << Unic::instanta()->get_i() << endl;
  return 0;
}
```

Metode ce întorc referințe la membri **`private`**

- Atenție la astfel de metode!!

```
#include <iostream.h>
class A{
    public:
    A(int x = 0): el(x){}
    int& get_el() {return el;} // intoarce referinta la el
private:
    int el;
};
main(){
    A a(5); int& b = a.get_el(); // b este alias pentru a.el
    cout << (a.get_el())<< ' ' ;
    b= b+5; // se modifica a.el prin intermediul lui b
    cout << a.get_el() << endl;
}
// 5 11
```

CLASE – Spațiul de nume

- **`class, struct, enum, union`** crează un spațiu distinct – ***spațiu de nume*** – căruia aparțin toate numele folosite acolo;
- Clasele definite în alte clase pot conține membri statici:

```
class C{class B{static int i;};};
int C::B::i=99;
```

- Clasele locale (clase definite în funcții) nu pot conține membri statici:

```
void f(){
   class Local{
   public :
    static int i; // Eroare: nu se poate defini i!
   } x;
}
```

Spațiu de nume - exemplu

```
#include <iostream.h>
int a;  double b;
class A{
public:
    A(int aa = 1) :a(aa){}
    void print(){cout << "A::a= " << a << endl;}
    class B{
    public:
            B(int aa = 2) : a(aa){}
            void print(){cout << "A::B::a= " << a << endl;}
    private:
            int a;
    };
    friend B; // daca B are acces direct la  A::a, A::b
    static double get_b(){ return b;}
private:
    int a;
    static double b;
};
```

Spațiu de nume - exemplu

```
double A::b = ::b;
void main(){
    a=5; b = 5.5;
    A a;
    a.print();
    //B b; // Eroare: 'B': undeclared identifier
    A::B b;
    b.print();
    cout << "A::b = " << a.get_b()<<endl;
    cout << "::a = " << ::a << " ::b= " << ::b ;
}
/*
A::a= 1
A::B::a= 2
A::b = 0
::a = 5 ::b= 5.5
*/
```

CLASE – Spațiul de nume (cont)

- Numele funcțiilor globale, variabilele globale, numele claselor fac parte dintr-un spațiu de nume global ; probleme grave uneori. Soluții:
  - Crearea de nume lungi, complicate
  - Divizarea spațiului global prin crearea de **`namespace`**
- Creare : **`namespace <`*`Nume_Namesp`*`>{ <`*`Declaratii`*`> }`**
- Diferențele față de **`class, struct, union, enum:`**
  - Se poate defini doar global, dar se pot defini **`namespace`** imbricate
  - Definiția namespace nu se termină prin ‘;’
  - O definiție **`namespace`** poate continua pe mai multe fişiere header

CLASE – Spațiul de nume (cont)

- Un **namespace** poate fi redenumit:

```
namespace BibliotecaDeProgramePersonaleAleLuiIon{}
namespace  Ion = BibliotecaDeProgramePersonaleAleLuiIon
```

- Nu pot fi create instanțe ale unui **namespace**

- Utilizare namespace-urilor deja declarate:
  - Folosirea operatorului de rezoluție:

    class <*NumeSp*>::C{};
  - Folosirea directivei **using:**

    using namespace <*NumeSp*>;
  - Folosirea declarației **using:**

    using <*NumeSp*>::<*Nume_membru*>;

CLASE – Spațiul de nume (Exemple)

```
namespace Int {
  enum sign { pos, neg };
  class Integer {
    public:
     Integer(int ii = 0):i(ii),s(i >= 0 ? pos : neg){}
     sign getSign() const { return s; }
     void setSign(sign sgn) { s = sgn; }
    };
}
namespace Math {
  using namespace Int;
  Integer a, b;
  Integer divide(Integer, Integer);
  // ...
}
```

CLASE – Spațiul de nume (Exemple)

```
namespace U {inline void f() {} inline void g() {}}
namespace V {inline void f() {} inline void g() {}}
void h() {
  using namespace U; // Directiva using
  using V::f; // Declaratia using
  f(); // Apel V::f();
  U::f();
}
namespace Q {using U::f;using V::g;}
void m() {
  using namespace Q;
  f(); // Apel U::f();
  g(); // Apel V::g();
}
int main() {
    h();
    V::f();
    m();
}
```

Pointeri la date membru Operatorii .* şi ->*

- Declarație:

  tip Nume_clasa::*****_point_membru_;

  tip Nume_clasa::*****_point_membru_ **=**

  &*Nume_clasa*::*Nume_membru*;

- Utilizare

  Nume_clasa obiect, *****_point_obiect_;

  obiect.*****_point_membru_ = …

  point_obiect **->***_point_membru_ = …

Pointeri la metode

- Declarație

  tip (*Nume_clasa*::**point_func*) (*parametri*) ;

  tip (*Nume_clasa*::**point_func*) (*parametri*) =

  &*Nume_clasa*::*Nume_metoda*;

- Utilizare:

  (obiect.**point_func*) (*parametri*) ;

  (*point_obiect* ->**point_func*) (*parametri*) ;

- Un pointer la o funcție membru trebuie să se potrivească în trei elemente:
  - numărul şi tipurile argumentelor
  - tipul valorii returnate
  - tipul clasei a cărei membru este

Exemplu

```
//Pointeri la functii membru
class C{
public:
    void f(int n=5) const {cout << "apel C::f(" << n <<")" << endl;}
    void g(int n) const {cout << "apel C::g(" << n <<")" << endl;}
    void h(int n) const {cout << "apel C::h(" << n <<")" << endl;}
    void i(int n) const {cout << "apel C::i(" << n <<")" << endl;}
};
void main(){
    C c;
    C* pc = &c;
    void (C::*p_metoda)(int=0) const = &C::h;
    (c.*p_metoda)(7);
    (pc->*p_metoda)();
    p_metoda = &C::f;
    (pc->*p_metoda)(8);
    (c.*p_metoda)();
    c.f();
}

// apel C::h(7) apel C::h(0) apel C::f(8) apel C::f(0) apel C::f(5)
```

Atenție la declararea pointerilor

```
class Ex{
public:
   int fct(int a, int b){ return a+b;}
};
//typedef int (Ex::*TF)(int,int);
typedef int (*TF)(int,int);
void main(){
   Ex ex;
   TF f = Ex::fct; // eroare: f nu e pointer
                   // al clasei Ex
   cout << (ex.*f)(2,3) << "\n";
}
```

Supraîncarcare operatori: `operator@`

- Nu se pot supraincarca operatorii:
  - `::` scope resolution
  - `.` acces la membru
  - `.*` dereferențiere membru
  - `?:` **`sizeof()`**
- Operatorii binari se definesc fie ca funcții membru nestatic cu un argument sau funcții nemembru cu 2 argumente

Supraîncarcare operatori: **`operator@`**

- Operatorii unari se definesc fie ca funcții membru nestatic fără argumente (cu un argument **`int`** pentru postfix) sau funcții nemembru cu 1 argument (2 argumente postfix);

- **`operator=`**, **`operator[]`**, **`operator()`** şi **`operator->`** trebuie definiți ca funcții membru nestatic; asta asigură că primul operand este *lvalue*

Supraîncarcare operatori

- **operator<<** şi **operator>>** nu pot fi definite ca metode; în **std** sunt definiți aceşti operatori şi redefinirea ca metode ar însemna modificarea claselor din std ceea ce este imposibil**;**
- **operator<<** şi **operator>>** nu pot fi definite ca **friend** dacă se foloseşte **std;**

- Pentru un operator binar **x@y** unde x este de tip X iar y de tip Y sistemul procedează astfel:
  - Dacă X este o clasă se determină dacă X sau clasa sa de bază defineşte **operator@** ca membru; dacă da se foloseşte acest operator
  - Dacă nu, se caută declarația lui **operator@** în contextul ce conține **x@y** apoi în **namespace**-ul lui X, în **namespace** -ul lui Y şi se încearcă aplicarea celui ce se potriveşte mai bine. Condiția este ca cel puțin un operand să fie de tip(clasa) utilizator.

operator<< şi operator>>

```
#include<iostream>
using namespace std;
class Punct{
   int x, y;
public :
    Punct(int xx=0, int yy=0): x(xx), y(yy) { }
    int getx() const{return x;}
    int gety() const{return y;}
};
ostream& operator<<( ostream& os,const Punct& p){
  return os <<'('<<p.getx()<<","<<p.gety()<<')'<< endl;
}
istream& operator>>(istream& in, Punct& p){
    int x,y;
    in >> x >> y;
    p = Punct(x,y);
    return in;
}
```

Supraîncarcare operator []

```
#include <iostream>
using namespace std;
class Cstring{
    char* rep;
    int lung;
public:
    Cstring(char* s = "", int l = 0) ;
    Cstring(const Cstring&);
    char& operator[](int);
    Cstring& operator=(const Cstring&);
    int get_lung() { return lung;}
};
char& Cstring::operator[](int i){return *(rep+i);}
Cstring::Cstring(char* s, int l):rep(s),lung((l==0)?strlen(s): l)
{
    cout << "Sirul : '"<< rep;
    cout << "' are lungimea : " << lung << endl;
}
```

Supraîncarcare operator []

```
int main(){
    Cstring p1("Popescu Ion"), p2("Ionescu Paul");
    cout << " \nSirul p1 folosind operator[] : “;
    cout << endl;
    for ( int i = 0; i < p1.get_lung(); i++)
       cout << p1[i];
    p1 = p2;
    cout << " \nNoul sir p1 folosind operator[]: “;
    cout << endl;
    for ( i = 0; i < p1.get_lung(); i++)
       cout << p1[i];
    return 0;
}
```

Operator () – Obiecte funcții

```
class Matrice{
public:
   Matrice(int, int);
   int& operator()(int, int);
   int get_s1(){return size1;}
   int get_s2(){return size2;}
private:
   int a[100];
   int size1,size2;
};
```

Supraîncarcare operator ()

```
int& Matrice::operator()(int i, int j){
  if(i < 0 || i >= size1){
  cerr << "Primul indice gresit:" << i;
  cerr << endl;
  return a[0];
  }
  if(j < 0 || j >= size2){
  cerr << "Al doilea indice gresit:" << j;
  cerr << endl;
  return a[0];
  }
  return a[i*size2 + j];
}
```

Supraîncarcare operator ()

```
int main(){
Matrice a(3, 4);
int i, j;
for (i = 0; i < a.get_s1(); i++)
   for(j =0; j < a.get_s2(); j++)
          a(i, j) = 2*i + j;
for (i = 0; i < a.get_s1(); i++) {
   for(j =0; j < a.get_s2(); j++)
          cout <<  a(i, j) << "  ";
   cout << endl;
}
cout << a(1, 2) << endl;
cout << a(4, 2) << endl;
cout << a(2, 8) << endl;
return 0;
}
```

Supraîncarcare operator ()

```
/*
0  1  2  3
2  3  4  5
4  5  6  7
4
Primul indice in afara domeniului:4
0
Al doilea indice in afara domeniului:8
0
*/
```