Conceptul de clasă - Plan

- Membri, controlul accesului, constructori, membri statici, funcții membru constante, funcții inline.
- Setul de operații pentru definirea unui tip utilizator: constructori, accesori, manipulatori, operatori, copiere, asignare, excepții
- Obiecte: construire și distrugere, copiere, new/delete, obiecte ca date membru, inițializarea obiectelor, membri constanți, tablouri de obiecte, obiecte locale, obiecte nelocale, obiecte temporare.

- Obiecte membru, ordinea de apel a constructorilor
- Date membru calificate **`const`**, membri referință, inițializare
- Metode ce întorc referințe la membri din secțiunea **`private`**
- Spațiu de nume, domeniu de vizibilitate
- Pointeri la membri
- Prieteni
- Supraîncărcare operatori
  - Operatorii +=, +,++
  - Operatorii <<, >>
  - Operatorii [] şi ()
- Conversii

CLASE

- O clasă în C++ se caracterizează prin:
  - Un nume - identificator
  - O colecție de *date membru* –prin aceste date este reprezentat un obiect. Datele nestatice sunt create pentru fiecare obiect (instanță a clasei)
  - O colecție de *funcții membru* – tratamente (metode) ce se aplică obiectelor. Un tratament se execută de către un obiect ca răspuns la primirea unui mesaj
  - Eventual, o colecție de prieteni: funcții şi/sau clase prieten

CLASE

- Specificarea unei clase se face prin numele rezervat **`class`** sau **`struct`**

- Specificarea este cuprinsă, în general, într-un fişier header ce va fi partajat de fişierele ce implementează clasa şi de cele ce utilizează clasa

- Numele clasei este numele unui tip utilizator: se pot declara variabile care au acest tip

- Componentele unei clase sunt repartizate în trei domenii de protecție: public, protected, private

CLASE - Declarație

- Declarație cu **`class`**:

`class` <*Nume_clasa*> **{***Corp_clasa***};**

```
class <Nume_clasa> {
        //Declarații membri privați (nerecomandat)
public:
        // Declarații funcții (metode) (eventual date)
protected:
        // Declarații date (eventual metode)
private:
        // Declarații date (eventual metode)
};
```

CLASE - Declarație

- Declarație clasă cu **`struct`**:

```
struct <Nume_clasa> {Corp_clasa};

struct <Nume_clasa> {
        // Declarații membri public (nerecomandat)
public:
        // Declarații funcții (eventual date)
protected:
        // Declarații date (eventual metode)
private:
        // Declarații date (eventual metode)
};
```

CLASE - Declarație

- Fiecare domeniu de protecție poate fi, fizic, împărțit în mai multe regiuni. Ordinea regiunilor poate fi arbitrară
- Domeniul **`public`**:
  - Componenta este vizibilă oriunde
  - Variabilele ce definesc starea unui obiect nu trebuie să fie publice (principiul încapsulării)
  - În general, metodele sunt în domeniul public
- Domeniul **`protected`**:
  - Componenta este vizibilă în funcțiile clasei, în funcțiile şi clasele prieten şi în clasele derivate
- Domeniul **`private`**:
  - Componenta este vizibilă doar în funcțiile clasei şi în funcțiile şi clasele prieten

Exemplu

```
#define MAX_STACK 10
class Stack
{
  public:
     Stack(int = MAX_STACK);
     ~Stack();
     void push(char);
     void pop();
     char top();
     bool is_empty();
     void display();
  private:
     char *arr;
     int top_index, nMax;
};
```

Exemplu

```
//data.h
#include <iostream>

using std::ostream;
//using namespace std;

class Data {
 public:
  Data(int oZi = ziCurenta(), int oLuna = lunaCurenta(),
   int unAn = anCurent());
  ~Data();
  void pesteNZile(int n);
  void pesteNLuni(int n);
  void pesteNAni(int n);
  Data operator ++(int); // postfix
  Data& operator ++();    // prefix
  void setZi(int zi_noua);
  void setLuna(int );
  void setAn(int an_nou);
```

Exemplu – continuare

```
  int getZi() const;
  int getLuna() const;
  int getAn() const;
  friend ostream& operator << (ostream&, const Data&);
  static Data azi();
 private:
  int zi, luna, an;
  static int ziCurenta();
  static int lunaCurenta();
  static int anCurent();
  bool esteAnBisect();
  int NrZileLuna();
  int NrZileAn();
};
```

OBIECTE - Declarație

- Declarație obiecte:

  <Nume_clasa> <Nume_Obiect>,… ;

  <Nume_clasa> <Nume_Obiect>(<Parametri_actuali>);

- La declararea unui obiect:
  - Se alocă memorie pentru datele membru
  - Se execută o metodă constructor
- La ieşirea dintr-un bloc
  - Se apelează destructorul (este unic) pentru fiecare din obiectele locale, ce încetează să mai existe

OBIECTE - Declarație

```
Data d1; // Apel constructor d1.Data()
Data d2(26,4,2010);
// Apel constr. Data(int,int,int)
Data d3[5];// Apel constructor Data() de 5 ori
```

- Obiecte dinamice:

```
Data *pd = new Data; // apel constructor pd->Data()
Data *t = new Data[5];// apel de 5 ori t->Data()
delete pd;  // se apeleaza pd->~Data()
delete [] t; // se apeleaza t ->~Data() de 5 ori
```

Exemplu

```
#include <iostream>
using namespace std;
class C
{
   public:
        C(){cout << "C() ";}
        ~C(){cout << "~C() ";}
   private:
        int i;
};
int main(){
   C c1;
   C* p1= new C;
   {
        C c[2];
        C* p2 = new C;
        delete p2;
   }
   C c[2];
   delete p1;
   return 0;
}
//C() C() C() C() C() ~C() ~C() ~C() C() C() ~C() ~C() ~C() ~C()
```

CLASE - Implementare

- Funcții membru(Metode) - Implementare:
  - În cadrul clasei – implementarea se face, în general, inline (sistemul decide acest lucru)
  - În afara clasei (precedat de **`inline`** pentru implementare inline):

<tip> *<Nume_clasa>* **::** *<Nume_funcție>*(*<Lista_parametri>*)

{

// Corp funcție

}

CLASE - Implementare

```
void Data::pesteNLuni(int n)
{
  int ll = luna + n;
  if (ll > 12)
    {
      luna = ll % 12;
      if (luna == 0) luna = 12;
      an += (ll - luna)/12;
    }
  else
    luna = ll;
}

inline void Data::setAn(int an_nou){an = an_nou;}
```

CLASE - Mesaje

- Apel(Invocare) funcție membru (Mesaj către obiect):

  <Nume_obiect>`.` *<Nume_functie>*(*Lista_par_act*)

  <Nume_point_obiect> `->` *<Nume_functie>*(*Lista_par_act*)

  (`*`*<Nume_point_obiect>*)`.` *<Nume_functie>*(*Lista_par_act*)

```
Data* pd, azi;
azi.setAn(2010);
azi.setLuna(4);
azi.setZi(26);
pd = &azi;
pd->get_zi();
(*pd).get_zi();
```

CLASE – Funcții membru

- Funcții manager: constructori, destructor

  `Data(int, int, int), Data(int, int), Data(int), Data(), Data(const char*)`

  `~Data()`

- Funcții de acces (examinare obiect); aceste funcții nu trebuie să modifice starea obiectului

  `Data::getZi(), Data::setAn()`

- Implementori:

  `void Data::adunaZi()`

- Operatori :

  `Data& Data::operator++()`

CLASE – Funcții membru

- Funcții ajutătoare:

```
bool Data::esteAnBisect(){
    return (an%400==0)||((an%4==0)&&(an%100!=0));
}
```

- Funcții membru **const** (nu modifică starea obiectului):

```
inline int Data::getZi() const {return zi;}
int Data::getAn() const {return an++;}//Eroare!
```

CLASE – Metode **const**

- O funcție membru **const** poate fi apelată atât pentru obiecte **const** cât şi pentru obiecte non - **const**;
- O funcție membru non - **const** nu poate fi apelată pentru obiecte **const**!
- Exemplu

```
void f(Data& d, const Data& cd){
  int i = d.getAn();
  d.adunaAn(2);
  int j = cd.getAn();
  cd.adunaAn(1); // Eroare: cd este obiect const!
  //…
}
```

- **const** face parte din tipul metodei, la implementarea în afara clasei trebuie precizat acest lucru:

```
inline int Data::getAnn() const{
   return an;
}
```

CLASE – Membri **`static`**

- O variabilă membru dar nu este parte a obiectelor (instanțe ale clasei) se numește variabilă membru static
  - O singură copie a unei variabile membru static este partajată de toate obiectele; există câte o copie din variabilele membru nonstatic pentru fiecare instanță(obiect) a clasei

```
static Data default_date;
static int nr_instante;
```

  - Definiţia unui membru **`static`** trebuie să apară, o singură dată, undeva în afara clasei dar în același fișier:

```
int Data::nr_instante = 0;
Data Data::default_date(10,1,1906)
```

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 **`static`**

- O funcție ce trebuie să acceseze membrii clasei dar nu e nevoie să fie invocată pentru un anume obiect este numită funcție membru **`static`**
  - funcție asociată clasei: se poate apela prin **`X::f()`**
  - o funcție calificată static poate fi apelată şi ca mesaj către obiect; nu este indicat
  - poate accesa doar membrii statici ai clasei

```
static void set_default(int, int, int);
```

  - O funcție statică trebuie definită undeva:

```
void Data::set_default(int dd, int mm, int yy)
{
   default_date = Data(dd,mm,yy);
}
```

CLASE –membri **static**

```
class X {
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
  }
private:
  int i;
  static int j;
};
```

O clasă cu obiect unic (singleton)

```
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;
}
```

(Auto)referințe

- Pentru modificarea unei date prin adăugare de ani, luni, zile s-au folosit funcțiile:

```
void pesteNZile(int n);
…
```

- Uneori este mai util să returneze referințe la Data:

```
Data& pesteNZile(int n);
Data& pesteNLuni(int n);
Data& pesteNAni(int n);
```

pentru a putea scrie ceva de genul:

```
azi.pesteNAni(5).pesteNLuni(3).pesteNZile(20).
```

CLASE - Pointerul implicit **`this`**

- Pentru fiecare funcție membru există o singură instanță care este accesibilă fiecărui obiect

- Pointerul **`this`** – pointer implicit accesibil doar în funcțiile membru nonstatic ale unei clase (structuri); pointează la obiectul pentru care este apelată funcția

```
int getLuna() const {return this->luna;}
int getLuna(Data* d )const {return d->luna;}

Data azi(26,4,2010);
cout << azi.getLuna() << endl;
cout << azi.getLuna(&azi) << endl;
```

`(*this)` – pentru (returnarea) obiectul(ui) curent într-o funcție membru

CLASE - Pointerul implicit **this**

- Exemple:

```
void Data::setLuna( int m ) {
   luna = m;
   //this->luna = m;
   //(*this).luna = m;
}
Data& Data::pesteNAni(int n) {
   if(zi==29 && luna==2 && !esteAnBisect(an+n){
       zi = 1; luna = 3;
   }
   an += n;
   return (*this);
}
```

Exemplu: Pointerul implicit **this**

```
#include <iostream>
using namespace std;
class ExThis{
public:
   ExThis() :Y(0.0){
        cout << "Obiect " << X++  ;
        cout << " : this = " << this << "\n";

   }
   ~ExThis(){
        cout << "Se distruge:" << --X << " this = " << this << "\n";
         }
private:
   static int X;
   double Y;
};

int ExThis::X = 0;
void main(){
      cout << "main: Se declara tabloul Tab[5] de tip ExThis:\n" ;
      ExThis Tab[5];
      cout << "Sfarsit main: Se distrug obiectele din tablou:\n";
}
```

Exemplu- Pointerul implicit **this**

```
/*

main: Se declara tabloul Tab[5] de tip ExThis:
Obiect 0 : this = 0x0012FF4C
Obiect 1 : this = 0x0012FF54
Obiect 2 : this = 0x0012FF5C
Obiect 3 : this = 0x0012FF64
Obiect 4 : this = 0x0012FF6C
 Sfarsit main: Se distrug obiectele din tablou:
Se distruge:4 this = 0x0012FF6C
Se distruge:3 this = 0x0012FF64
Se distruge:2 this = 0x0012FF5C
Se distruge:1 this = 0x0012FF54
Se distruge:0 this = 0x0012FF4C

*/
```

Constructorii unei clase

- Așa cum a fost proiectată clasa Data se poate declara:

```
Data d(45,17,21);
```

- Constructorul ar trebui să fie responsabil de validarea datelor:

```
class Data
{
public:
    enum Luna{ian = 1, feb, mar, apr, mai, iun, iul, aug, sep, oct, nov,
    dec};
    class Data_gresita { };
    Data(int zz=0, Luna ll =Luna(0), int aa=0); // 0 = data implicita
    int getZi() const;
    Luna getLuna() const;
    int getAn() const;
    static void set_default(int, Luna, int);
    //..
private:
    int zi, luna, an;
    static Data default_date;
};
Data Data::default_date(10,1,1906);
```

CONSTRUCTORUL

```
Data::Data(int zz, Luna ll, int aa)
{
    if(aa == 0) aa = default_date.getAn();
    if(ll == 0) ll = default_date.getLuna();
    if(zz == 0) zz = default_date.getZi();
    int max;
    switch(ll)
    {
    case feb:
         max = 28 + anBisect(aa);
         break;
    case apr:case iun:case sep:case nov:
         max = 30;
         break;
    case ian:case mar:case mai:case iul:case aug:case dec:
         max = 31;
         break;
    default:
         throw Data_gresita();
    }
    if (zz < 1 || max < zz) throw Data_gresita();
    an = aa;
    luna = ll;
    zi = zz;
}
```

CONSTRUCTORI

- Constructor implicit: poate fi apelat fără parametri

```
Data(int z = 0, int l = Luna(0), int a = 0);
```

- Dacă o clasă nu are constructori, se apelează unul implicit (furnizat de sistemul C++);
- Dacă o clasă are constructori, ar trebui să aibă şi unul implicit ;

- O clasă ce are membri **const** sau referințe trebuie să aibă constructori

```
class X{const int a; const int& b;};
X x; // Eroare, constructorul implicit nu poate
     // initializa a si b
```

CONSTRUCTORI

- Constructor şi operatorul new:

```
Data* d = new Data(26, 4, 2010);// d – obiect dinamic
class Tablou{
public:
    Tablou(int n=10){p = new char[sz = n];}
    ~Tablou(){delete[] p;}
private:
    const char* p; int sz;
};
```

- Dacă o clasă are membri pointeri(este necesară alocare dinamică) trebuie definit explicit un constructor şi destructorul clasei.

- Obiectele membru într-o clasă trebuiesc construite la crearea obiectului compus: alegerea constructorului se face în lista de iniţializare a constructorului clasei compuse

CONSTRUCTORI

```
class Y{
public:
  Y(){y = 0;};
  Y(int n){y = n;};
private:
  int y;
};
class X{
public:
  //X(int n):y1(Y::Y()), y2(Y::Y(n)),x(n){}
  X(int n):y1(), y2(n),x(n){}
private:
  int x;
  Y y1,y2;
};
void main(){
X x(2);
}
```

DESTRUCTOR

- Distruge un obiect în maniera în care a fost construit
- Are sintaxa: **~<Nume_clasa>(){… }**
- Apelat implicit pentru o variabilă(obiect) din clasa automatic la părăsirea domeniului de vizibilitate
- Apelat implicit când un obiect din memoria heap, creat cu operatorul **new**, este şters cu operatorul **delete**

```
int main(){
  Tablou* p = new Tablou;
  Tablou* q = new Tablou;
  //…
  delete p;
  delete q;//delete p;
}
```

Creare / distrugere obiecte

- Obiect din clasa automatic (obiect local): este creat de fiecare dată când declararea sa este întâlnită la execuția programului și distrus când programul părăsește blocul în care este declarat
- Obiect în memoria heap: este creat cu operatorul **`new`** şi distrus cu operatorul **`delete`**;
- Obiect membru nestatic (ca dată membru a altei clase): este creat/distrus când un obiect al cărui membru este, este creat/distrus
- Obiect element al unui tablou: este creat/distrus când tabloul a cărui element este, este creat/distrus

Creare / distrugere obiecte

- Obiect local static: este creat când se întâlneşte prima dată declarația sa la execuția programului şi este distrus la terminarea programului;
- Obiect global, obiect în namespace, obiect membru static: este creat o singură dată la începutul execuției programului şi distrus la terminarea programului
- Obiect temporar: este creat ca parte a evaluării unei expresii şi distrus la sfârşitul evaluării în întregime a expresiei în care apare

CLASE: Copiere şi atribuire

- Pentru orice clasă, se pot iniţializa obiecte la declarare şi se poate folosi operatorul de atribuire

- Iniţializarea şi atribuirea implicită înseamnă copiere membru cu membru a unui obiect în celălalt:

```
Punct p1;  //apel constructor implicit
Punct p2 = p1;//initializare prin copiere:
              //p2 nu exista, se construeste
Punct p3;
p3 = p2;    //atribuire prin copiere:
            //p3 exista si se modifica
```

CLASE: Copiere şi atribuire

- Cazul în care nu avem alocare dinamică:

```
Data d1(26,4,2010);
Data d2 = d1; //echiv. cu Data d2(d1);
```

- La o declarație cu inițializare **`X x1(x2)`** este apelat automat un constructor, numit constructor de copiere: **`X::X(const X&)`**

- Pentru clasa Data acest constructor este echivalent cu:

```
Data::Data(const Data& d)
{
   zi = d.zi;
   luna = d.luna
   an = d.an
}
```

CLASE: Copiere şi atribuire

- 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 atribuire
}
```

```
// constructorul de copiere pentru Data
// datele membru zi, luna, an sunt dinamice

Data::Data(const Data& oData)
{
   zi = new int(*oData.zi);
   luna = new int(*oData.luna);
   an = new int(*oData.an);
}

// operatorul de atribuire
Data& Data::operator=(const Data &oData)
{
   if(this != &oData) // evitare atribuire x = x
   {   delete zi;
       delete luna;
       delete an;
       zi = new int(*oData.zi);
       luna = new int(*oData.luna);
       an = new int(*oData.an);
   }
   return *this;
}
```

CLASE: Copiere şi atribuire

- Constructorul de copiere se apelează(implicit):
  - 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

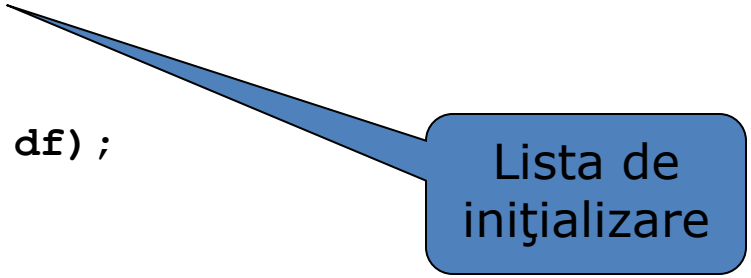
CLASE – Obiecte membru

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

Clasa::Clasa(par) : *Obiect1(par1), Obiect2(par2)…*{…}

```
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), data_fond(df), membri_fond(), membri()
{
   // 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, membri calificați **const** sau membri 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;
};
```

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

- Atenție la astfel de metode!!

```
class A{
   public:
   A(int x = 0): el(x){}
   int& get_elem() {return elem;} // referinta la elem
private:
   int elem;
};
main(){
   A a(5); int& b = a.get_elem(); // b alias pentru a.elem
   cout << (a.get_elem())<< ' ' ;
   b= b + 5; // se modifica a.elem prin intermediul lui b
   cout << a.get_elem() << endl;
}
// 5 11
```

CLASE – Spațiul de nume

- **`class, struct, enum, union`** creează 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;
}
```

CLASE – Spațiul de nume (cont)

- Numele funcțiilor globale, variabilele globale, numele claselor fac parte dintr-un spațiu de nume global ; apar 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 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, când se instanțiază, trebuie să se potrivească cu funcția respectivă 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";
}
```

PRIETENII UNEI CLASE

- O declarație de funcție membru înseamnă:
  - Funcția poate accesa partea privată a clasei;
  - Funcția este în domeniul de vizibilitate al clasei;
  - Funcția poate fi invocată de un obiect (are acces la pointerul **`this`**)

- Declararea unei funcții **`friend`**, îi conferă doar prima proprietate
- Se declară atunci când este necesar accesul la membrii privați

```
Class Vector{}; class Matrice{}
Vector operator*(constMatrice& m, const Vector& v){
  Vector r;
  // Implementare r = m*v, Trebuie sa aiba acces la
  // reprezentarile claselor Vector si Matrice
  return r;
}
```

PRIETENII UNEI CLASE

- O clasă poate avea prieteni:
  - Alte clase (structuri)
  - Funcții
  - Operatori
- Adesea operatorii << şi >> sunt declarați **`friend`**

```
class Data {
/…
friend ostream& operator <<(ostream&, const Data&);
//…
};
ostream& operator << (ostream& o, const Data& d) {
  o << d.zi << ' ' << d.luna << ' ' << d.an;
  return o;
}
```

Exemplu

```
// Functii si clase friend

struct X;
// Declaratie necesara pentru definitia lui Y

struct Y {
  void f(X*);
};

struct X { // Definitia clasei X
public:
  void init();
  friend void g(X*, int);      // Functie globala friend
  friend void Y::f(X*);        // Functie din Y friend
  friend struct Z;             // Structura(clasa) friend
  friend void h();             // Functie globala friend
private:
  int i;
};
```

Supraîncărcare operatori: `operator@`

- Nu se pot supraîncărca operatorii:

  `::` scope resolution

  `.` acces la membru

  `.*` dereferențiere membru

  `?:` `sizeof()` `typeid`

- Nu pot fi introduși noi operatori
- Operatorii binari se definesc fie ca funcții membru nestatic cu un argument sau ca funcții nemembru cu 2 argumente
- Interpretarea expresiei **`a@b`**:

  `a.operator@(b)` respectiv **`operator@(a,b)`**

Supraîncărcare operatori: **`operator@`**

- Operatorii unari se definesc fie ca funcții membru nestatic fără argumente (cu un argument **`int`** pentru operator postfix) sau ca funcții nemembru cu 1 argument (2 argumente pentru operator postfix);
  - **`@a`** înseamnă **`a.operator@()`** respectiv **`operator@(a)`**
  - **`a@`** înseamnă **`a.operator@(int)`** respectiv **`operator@(a, int)`**
- **`operator=, operator[], operator()`** şi **`operator->`** trebuie definiți ca funcții membru nestatic; asta asigură că primul operand este *lvalue*

- Recomandare: Minimizarea numărului funcțiilor ce au acces direct la reprezentare:
  - Operatorii ce modifică valoarea primului argument, ca membri (ex. +=)
  - Operatorii ce produc noi valori, ca funcții externe (ex. +, –,)

```
class complex{
  public:
    complex& operator+=(const complex a);
    //…
  private:
    double re, im;
};
inline complex& complex:: operator+=(const complex a){
  re += a.re; im += a.im; return *this;
}
complex operator+(complex a, complex b){
  complex s = a;
  return s+= b;
}
```

Exemplul: + supraîncărcat

```
struct X{
     void operator+(int);
     void operator+(double);
     X(int);
     int n;
     ...
  };
  void operator+(X, X);
  void operator+(double, X);
  void f(X a){
     a+99;            // a.operator+(99)
     a+5.55;          // a.operator+(5.55)
     33 + a;  // ::operator+(X(33), a)
     1.1 + a; // ::operator+(1.1, a)
     a + a            // :: operator+(a, a)
  }
```

Exemplu: &

```
struct Y{
  public:
     Y* operator&();
          // adresa, operator unar
     Y operator&(Y);
          // operatorul logic &, binar
     Y operator++(int); // ++ postfix, unar
     Y operator&(Y, Y);
          // Eroare, prea multe argumente
     Y operator/();
          // Eroare, prea putine argumente
};
```

Exemplu: ++

```
struct Clock{
    Clock tic_tac();
    Clock operator++();
    Clock operator++(int);
    int ora, min, ap; // ap=0 pentru AM, 1 pentru PM
};
Clock Clock::tic_tac(){
    ++min;
    if(min == 60){
        ora++;
        min = 0;
    }
    if(ora == 13)
        ora = 1;
    if(ora == 12 && min == 0)
        ap = !ap;
    return *this;
}
```

Exemplu: ++ (cont)

```
Clock Clock::operator++(){
   return tic_tac();
}

Clock Clock::operator++(int n){
   Clock c = *this;
   tic_tac();
   return c;
}
```

operator<< şi operator>>

```
#include<iostream>
using namespace std;
class Punct{
public :
  Punct(int xx=0, int yy=0): x(xx), y(yy) { }
  int getx() const{return x;}
  int gety() const{return y;}
private:
  int x, 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;
}
```

Supraîncărcare **operator[]**

```
#include <iostream>
using namespace std;
class Cstring{
public:
   Cstring(char* s = "", int l = 0) ;
   Cstring(const Cstring&);
   char& operator[](int);
   Cstring& operator=(const Cstring&);
   int get_lung() { return lung;}
private:
   char* rep;
   int 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;
}
//…
```

Supraîncărcare 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:
  enum { max_Size = 20};
  Matrice(int s1, int s2):size1(s1), size2(s2){}
  int& operator()(int, int);
  int get_s1(){return size1;}
  int get_s2(){return size2;}
private:
  int a[max_Size];
  int size1,size2;
};
```

Supraîncărcare `operator()`

```
int& Matrice::operator()(int i, int j){
if(i < 0 || i >= size1)
      throw "Primul indice gresit\n";
if(j < 0 || j >= size2)
      throw "Al doilea indice gresit\n";
return a[i*size2 + j];
}
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;
}
try{
      cout << a(1, 2) << endl;
      cout << a(2, 8) << endl;
}
catch(char* s){cout << s ;}
return 0;
}
```

Supraîncărcare **operator()**

```
/*
0  1  2  3
2  3  4  5
4  5  6  7
4
Al doilea indice gresit
*/
```

Înțeles predefinit pentru operatori

- =, & (adresa) și , au înțeles predefinit pentru orice clasă
- Acest înțeles poate fi anulat prin plasarea lor în secțiunea private

```
class X {
private:
        void operator=(const X&);
        void operator&();
        void operator,(const X&);
        //…
};
```

- Poate fi stabilit alt înțeles prin supraîncărcare

CLASE: Constructor de conversie

- Constructorul de conversie defineşte o conversie(ce se aplică implicit) de la un tip (de baza) la un tip utilizator

X::X(*tip_de_baza* m)

```
punct(int i) : x(i), y(0) {} // int -> punct
data(int d):zi(d), luna(luna_curenta()),
  an(anul_curent()) {} // int -> data
complex(double r) : re(r), im(0){} // double -> complex

void f(){
  complex z = 2; // z = complex(2)
  3 + z; // complex(3) + z;
  z += 3; // z += complex(3);
  3.operator+=(z); // eroare
  3 += z; // eroare
}
```

CLASE: Constructor de conversie

- Constructorul de conversie poate suplini definițiile unor operatori; pentru clasa complex nu-i necesar a defini:

```
complex operator+(double, complex);
complex operator+(complex, double);
```

- Un constructor de conversie nu poate defini:
  - O conversie implicită de la un tip utilizator la un tip de bază
  - O conversie de la o clasă nouă la o clasă definită anterior, fără a modifica declarațiile vechii clase
  - Soluția: Operatorul de conversie

Constructor explicit

- Un constructor cu un singur argument definește o conversie implicită; uneori acest lucru poate produce erori:

```
complex z = 2 // ok
String s = 'a' ; // nu-i ok daca exista
  //constructorul String(int n)
```

- Conversia implicită poate fi stopată dacă se declară constructorul explicit

```
class String{
    //…
    explicit String(int n);
    String(const char* p);
    //…
};
```

Constructor explicit

```
class An{
    int y;
public:
    explicit An(int i):y(i){}
    operator int() const {return y;}
};
class Data{
public:
    Data(int d, Luna m, An y);
    //…
};
Data d10(7, mar, 1977);// eroare
Data d11(7, mar, An(1977));//ok
```

CLASE: Operatorul de conversie

- este o funcție membru care definește o conversie de la tipul X la tipul T:

```
X::operator T()const;
```

Exemplu:

```
class Clock{
public:
  Clock(int = 12, int = 0, int = 0);
  Clock tic_tac();
  Clock operator++();
  Clock operator++(int);
  operator int()const;  // Conversie Clock --> int
                        // Ora 8:22 AM devine 822
                        // Ora 8.22 PM devine 2022
private:
  int ora;
  int min;
  int ap;  // 0 pentru AM, 1 pentru PM
};
```

CLASE: Operatorul de conversie

```
class fractie{
public:
  fractie(int n=0, int  m=1):numarator(n),
  numitor(m) { }
  operator int(){
        return numarator/numitor;
  }  //conversie fracţie -> int
  operator double(){
        return double(numarator)/numitor;
  }  //conversie fracţie -> double
private:
  int numarator, numitor;
};

fractie x(1,2);
double u = x;
int i = int(x);
```

CLASE: Conversie

- Ambiguitate constructor de conversie/operator de conversie:

```
class Apple {
public:
  operator Orange() const; // Apple -> Orange
};
class Orange {
public:
  Orange(Apple); // Apple -> Orange
};
void f(Orange) {}
int main() {
  Apple a;
  f(a);
}
//error C2664:'f' : cannot convert parameter 1 from
//'class Apple' to 'class Orange'
```

CLASE: Conversie

- Ambiguitate operatori conversie/supraîncărcare funcții:

```
class Orange {};
class Pear {};
class Apple {
public:
  operator Orange() const; // conversie Apple -> Orange
  operator Pear() const;   // conversie Apple -> Pear
};
// Supraincarcare eat():
void eat(Orange);
void eat(Pear);

int main() {
  Apple c;
  eat(c); // Error: Apple -> Orange or Apple -> Pear ???
}
```