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 3

16/10/2019

Agenda cursului

1. Recapitularea discutiilor din cursul anterior.

2. Conceptele de clasa și obiect.

2.1 Structura unei clase.

2.2 Metode de acces la membrii unei clase. Functii si clase prieten.

2.3 Constructorii și destructorul unei clase. Constructorul de copiere

2.4 Membrii statici ai unei clase.

1. Recapitulare curs 2 - Principiile programarii orientate pe obiecte

Principalele concepte care stau la baza POO sunt:

Abstractizarea

Incapsularea

Modularitatea

Ierarhizarea.

1. Recapitulare curs 2 - Principiile programarii orientate pe obiecte

Polimorfism

La compilare

-supraincarcare de functii

-supraincarcarea de operatori

-sabloane

La executie

-functii virtuale.

2. Conceptele de clasa si obiect

2.1. Structura unei clase

Clasa

- formeaza baza programarii orientate pe obiecte;
- folosita pentru a defini natura unui obiect;
- unitatea de baza pentru incapsulare;
- cuvant cheie “class”;
- similare cu “struct”;
- specificatorul de acces default = “private”;
- se poate utiliza “protected” pentru mostenire.

2. Conceptele de clasa si obiect

2.1. Structura unei clase

Sintaxa:

```
class nume_clasa {
        date si functii private
specificator de acces:
       date si functii;
specificator de acces:
       date si functii;
----------
specificator de acces:
       date si functii;
} lista de obiecte;
```

Obs:

- lista de obiecte este optionala;
- specificator de acces:
  - private
  - protected
  - public;
- se poate trece de la public la privat si iarasi la public;
- o clasa nu poate contine membri obiecte de tipul clasei, sau extern, register sau auto;
- o clasa poate contine membri pointeri catre obiecte de tipul clasei.

2. Conceptele de clasa si obiect

2.1. Structura unei clase

Exemplu (*sursa H. Schildt*):

```
class employee {
      char name[80]; // private by default
public:
      void putname(char *n); // these are public
      void getname(char *n);
private:
      double wage; // now, private again
public:
      void putwage(double w); // back to public
      double getwage();
};
```

- mai utilizata:

```
class employee {
      char name[80];
      double wage;
public:
void putname(char *n);
void getname(char *n);
void putwage(double w);
double getwage();
};
```

2. Conceptele de clasa si obiect

2.1. Structura unei clase

Struct vs Class

- struct are default membri ca public iar class ca private;
- pentru compatibilitate cu cod vechi;

Union vs Class

- union are default membri ca public iar class ca private;
- union nu participa la mostenire, class și struct da;
- consecinta: nu poate avea functii virtuale;
- nu exista: variabile de instanta statice, referinte și obiecte care fac overload pe operatorul =;
- obiecte cu (con/de)structor definiti nu pot fi membri in union.

2. Conceptele de clasa si obiect

2.1. Structura unei clase

```
#include <iostream>
using namespace std;
union swap_byte
{
    void swap();
    void set_byte(unsigned short i);
    void show_word();
    unsigned short u;
    unsigned char c[2];
};
void swap_byte::swap()
{
    unsigned char t;
    t = c[0];
    c[0] = c[1];
    c[1] = t;
}
```

Exemplu – union (*sursa H. Schildt*):

```
void swap_byte::show_word()
{
    cout << u;
}
void swap_byte::set_byte(unsigned short i)
{
    u = i;
}

int main()
{
    swap_byte b;
    b.set_byte(49034);
    b.swap();
    b.show_word();
    return 0;
}
```

R: 35519

2. Conceptele de clasa si obiect

2.1. Structura unei clase

Union anonime vs Class

- nu au nume pentru tip;
- nu se pot declara obiecte de tipul respectiv;
- folosesc aceeasi locatie de memorie;
- variabilele din union: accesibile ca si cum ar fi declarate in blocul respectiv
- nu pot avea funcții;
- nu poate avea private sau protected;
- union-uri anonime globale trebuiesc precizate ca statice.

2. Conceptele de clasa si obiect

2.1. Structura unei clase

Exemplu – union anonime(*sursa H. Schildt*):

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    // define anonymous union
    union {
    long l;
    double d;
    char s[4];
    } ;

// now, reference union elements directly
l = 100000;
cout << l << " ";
d = 123.2342;
cout << d << " ";
strcpy(s, "hi");
cout << s;
return 0;
}
```

2. Conceptele de clasa si obiect

2.2 Metode de acces la membrii unei clase. Functii si clase prieten.

Functii prieten

- cuvânt cheie : friend;

- permite accesarea campurilor protected și private din alta clasa;

- utilizare și în supraincarcarea operatorilor, pentru unele functii de I/O, si portiuni interconectate.

2. Conceptele de clasa si obiect

2.2 Metode de acces la membrii unei clase. Functii si clase prieten.

Expl. - Functii prieten

```
class Test
{
  int x; // privat
  public:
  void afis() {cout<<x<<endl;}
};

void functie (Test ob)
{
  cout<<"Obiect: "<<ob.x<<endl; // in
acest context
}

int main()
{
    Test a;
    functie(a);
    return 0;
}
```

```
class Test
{
  int x;
  public:
  void afis() {cout<<x<<endl;}
  friend void functie (Test);
};

void functie (Test ob)
{
  cout<<"Obiect: "<<ob.x<<endl;
}

int main()
{
    Test a;
    functie(a);

    return 0;
}
```

2. Conceptele de clasa si obiect

2.2 Metode de acces la membrii unei clase. Functii si clase prieten.

Exemplu (*sursa H. Schildt*):

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

const int IDLE = 0;
const int INUSE = 1;
class C2; // forward declaration

class C1 {
   int status; //IDLE or INUSE
public:
   void set_status(int state);
   friend int idle(C1 a, C2 b);
};

class C2 {
    int status;  //IDLE or INUSE
public:
    void set_status(int state);
    friend int idle(C1 a, C2 b);
};
```

```
void C1::set_status(int state) { status = state; }

void C2::set_status(int state) { status = state; }

int idle(C1 a, C2 b) {
  if(a.status || b.status) return 0;
  else return 1; }

int main()
{
  C1 x;    C2 y;
  x.set_status(IDLE);
  y.set_status(IDLE);
  if(idle(x, y)) cout << "Screen can be used.\n";
  else cout << "In use.\n";
  x.set_status(INUSE);
  if(idle(x, y)) cout << "Screen can be used.\n";
  else cout << "In use.\n";
  return 0;
}
```

2. Conceptele de clasa si obiect

2.2 Metode de acces la membrii unei clase. Functii si clase prieten.

Exemplu functii prieten din alte clase (*sursa H. Schildt*):

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

const int IDLE = 0;
const int INUSE = 1;
class C2; // forward declaration

class C1 {
    int status; //IDLE or INUSE
public:
    void set_status(int state);
   int idle(C2 b);
};

class C2 {
     int status;  //IDLE or INUSE
public:
     void set_status(int state);
     friend int C1::idle(C2 b);
};
```

```
void C1::set_status(int state) { status = state; }

void C2::set_status(int state) { status = state; }

 int C1::idle(C2 b)
 {
   if (this->status || b.status) return 0;
   else return 1;
 }

int main()
{
   C1 x;    C2 y;
   x.set_status(IDLE);
   y.set_status(IDLE);
   if(x.idle(y)) cout << "Screen can be used.\n";
   else cout << "In use.\n";
   x.set_status(INUSE);
   if(x.idle(y)) cout << "Screen can be used.\n";
   else cout << "In use.\n";
   return 0;
```

2. Conceptele de clasa si obiect

2.2 Metode de acces la membrii unei clase. Functii si clase prieten.

Clase prieten

- daca avem o clasa prieten, toate functiile membre ale clasei prieten au acces la membrii privati ai clasei

Exemple diverse:
- din proiecte;

```
Class Nod
{     int val;
      nod * urm;
public:
…….....
      friend class Lista;
};

Class Lista
{
      Public: // toate functiile
};
```

2. Conceptele de clasa si obiect

2.2 Metode de acces la membrii unei clase. Functii si clase prieten.

Functii inline

- utilizate în C++;
- doar o recomandare, nu imperativ, compilatorul poate alege cum trateaza;
- pentru functiile recursive, compilatorul, în general,nu o face;
- nu sunt apelate efectiv ci codul lor se dezvolta în interior, când e nevoie de ele;
- cuvânt cheie “inline” (in cazul explicit);
- se creeaza coduri eficiente;
- apelul de funcție ia timp,deci indicat inline;
- dezavantaj – cod mare prin duplicare;

Recomandare:
- este bine sa introducem inline doar funcții foarte mici;
- dezvoltare inline doar pentru functiile cu impact mare asupra performantelor programului.

2. Conceptele de clasa si obiect

2.2 Metode de acces la membrii unei clase. Functii si clase prieten.

Exemple - Functii inline (*sursa H. Schildt*):

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

 inline int max(int a, int b)
{
   return a>b ? a : b;
}

int main()
{
   cout << max(10, 20);
   cout << " " << max(99, 88);
   return 0;
}
```

Dpv compilator:

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

int main()
{
   cout << (10>20 ? 10 : 20);
   cout << " " << (99>88 ? 99 : 88);
   return 0;
}
```

2. Conceptele de clasa si obiect

2.2 Metode de acces la membrii unei clase. Functii si clase prieten.

Exemple - Functii inline membre ale unei clase (*sursa H. Schildt*):

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

class myclass {
   int a, b;
public:
   void init(int i, int j);
   void show();
};

// Create an inline function.
inline void myclass::init(int i, int j)
{    a = i;  b = j; }

// Create another inline function.
inline void myclass::show()
{ cout << a << " " << b << "\n"; }
```

```
int main()
{
   myclass x;
   x.init(10, 20);
   x.show();
   return 0;
}
```

2. Conceptele de clasa si obiect

2.2 Metode de acces la membrii unei clase. Functii si clase prieten.

Exemple - Functii inline definite implicit in clase (*sursa H. Schildt*):

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

class myclass {
   int a, b;
public: //automatic inline
   void init(int i, int j) {    a = i;  b = j; }
void show()
      { cout << a << " " << b << "\n"; }
};
 int main()
 {
   myclass x;
   x.init(10, 20);
   x.show();
   return 0;
 }
```

2. Conceptele de clasa si obiect

2.3 Constructorii si destructorul unei clase (*reminder curs 2*).

- initializare automata
- efectueaza operatii prealabile utilizarii obiectelor create.

Compilatorul aloca și eliberează memorie datelor membre.

Caracteristici speciale:
- numele = numele clasei (~ numele clasei pentru destructori);
- la declarare nu se specifica tipul returnat;
- nu pot fi mosteniti, dar pot fi apelati de clasele derivate;
- nu se pot utiliza pointeri către functiile constructor / destructor;
- constructorii pot avea parametri (inclusiv impliciti) și se pot supradefini.

Destructorul este unic și fără parametri.

Constructorii de copiere (discuții mai ample mai tarziu)
- creaza un obiect preluand valorile corespunzatoare altui obiect;
- exista implicit (copiaza bit-cu-bit, deci trebuie redefinit la date alocate dinamic).

2. Conceptele de clasa si obiect

2.3 Constructorii si destructorul unei clase.

Orice clasa, are by default:

- un constructor de initializare
- un constructor de copiere
- un destructor
- un operator de atribuire.

Constructorii parametrizati:

- argumente la constructori
- putem defini mai multe variante cu mai multe numere si tipuri de parametri

overload de constructori (mai multe variante, cu numar mai mare și tipuri de parametri).

2. Conceptele de clasa si obiect

2.3 Constructorii si destructorul unei clase.

```
class A
{
      int x;
      float y;
      string z;
};

int main()
{
      A a;    // apel constructor de initializare - fara parametri
      A b = a; // apel constructor de copiere
      A e(a); // apel constructor de copiere
      A c;    // apel constructor de initializare
      c = a;  //operatorul de atribuire (=)
}
```

2. Conceptele de clasa si obiect

2.3 Constructorii si destructorul unei clase.

Exemplu

```
class A
{
    int *v;
public:
    A(){v = new int[10]; cout<<"C";}
    ~A(){delete[ ]v;  cout<<"D";}
    void afis(){cout<<v[3];}
};

void afisare(A ob) { }

int main()
{
    A o1;
    afisare(o1);
    o1.afis();
}
```

2. Conceptele de clasa si obiect

2.3 Constructorii si destructorul unei clase.

Exemplu - Constructori parametrizati

```
class A
{
    int x;
    float y;
    string z;
public:
    A(){x = -45;}
    A(int x) {this->x = x; this->y = 5.67; z = "Seria 25";}
    // this -> camp e echivalent cu camp simplu: this ->z echivalent cu z
    A(int x, float y) {this->x = x; this->y = y; z = "Seria 25";}
    A(int x, float y, string z) {this->x = x; this->y = y; z = z;}

int main()
{
A a;
A b(34);
return 0;
}
```

2. Conceptele de clasa si obiect

2.3 Constructorii si destructorul unei clase.

Exemplu - Constructori parametrizati

```
class A
{
    int x;
    float y;
    string z;
public:

A(int x = -45, float y = 5.67, string z = "Seria 25") // valori implicite pt param
    {this->x = x; this->y = y; z = z;}
};

int main()
{
A a;
A b(34);
return 0;
}
```

2. Conceptele de clasa si obiect

2.3 Constructorii si destructorul unei clase.

Functiile constructor cu un parametru – caz special (sursa H. Schildt)

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

class X {
   int a;
public:
   X(int j) { a = j; }
   int geta() { return a; }
};

int main()
{
   X ob = 99; // passes 99 to j
   cout << ob.geta(); // outputs 99
   return 0;
}
```

Se creeaza o conversie explicita de date!

2. Conceptele de clasa si obiect

2.3 Constructorii si destructorul unei clase.

Tablouri de obiecte

Daca o clasa are constructori parametrizati, putem initializa diferit fiecare obiect din vector.

```
class X{int a,b,c; … };
X v[3] = {X(10,20) , X (1,2,3), X(0) };
```

Daca constr are un singur parametru, atunci se poate specifica direct valoarea.

```
class X {
 int i;
public:
X(int j) { i = j;}
};

X v[3] = {10,15,20 };
```

2. Conceptele de clasa si obiect

2.3 Constructorii si destructorul unei clase.

Exemplu – Ordine apel

```
class A
{
   int x;
public:
   A(int x = 0)   {
      x = x;
      cout<<"Constructor "<<x<<endl;   }
   ~A()   {
      cout<<"Destructor "<<endl;   }
   A(const A&o)   {
      x = o.x;
      cout<<"Constructor de copiere"<<endl;   }
   void f_cu_referinta(A& ob3)   {
      A ob4(456);   }
   void f_fara_referinta(A ob6)   {
      A ob7(123);   }
} ob;
```

```
int main()
{
   A ob1(20), ob2(55);
   ob2.f_cu_referinta(ob1);
   ob1.f_fara_referinta(ob3);
   A ob5;
return 0;
}
```

2. Conceptele de clasa si obiect

2.3 Constructorii si destructorul unei clase.

Exemplu – Ordine apel

1) In acelasi domeniu de vizibilitate, constructorii se apeleaza in ordinea declararii obiectelor, iar destructorii in sens invers.
2) Variabilele globale se declara inaintea celor locale, deci constructorii lor se declara primii.
3) Daca o functie are ca parametru un obiect, care nu e transmis cu referinta atunci, se activeaza constructorul de copiere si,implicit, la iesirea din functie, obiectul copie se distruge, deci se apeleaza destructor

```
/// Ordine:
/// 1. Constructor ob;
/// 2. Constructor ob1;
/// 3. Constructor ob2;
/// 4. Constructor ob4; - ob3 e referinta/alias-ul ob1, nu se creeaza obiect nou
/// 5. Destructor ob4;
/// 6. Constructor copiere ob6
/// 7. Constructor ob7
/// 8. Destructor ob7
/// 9. Destructor ob6
/// 10. Constructor ob5
/// 11. Destructor ob5
/// 12. Destructor ob2
/// 13. Destructor ob1
/// 14. Destructor ob
```

2. Conceptele de clasa si obiect

2.3 Constructorii si destructorul unei clase.

Exemplu – Ordine apel

```
class A {
public:
  A(){cout<<"Constr A"<<endl;}
};

class B {
public:
  B() {cout<<"Constr B"<<endl;}
private:
   A ob;
};

int main()
{
  B ob2;
  /// Apel constructor obiect A si apoi constructorul propriu
}
```

2. Conceptele de clasa si obiect

2.3 Constructorii si destructorul unei clase.

```
class A {
    int x;
public:
    A(int x = 7){this->x = x; cout<<"Const "<<x<<endl;}
    void set_x(int x){this->x = x;}
    int get_x(){ return x;}
    ~A(){cout<<"Dest "<<x<<endl;}
};

void afisare(A ob) {
    ob.set_x(10);
    cout<<ob.get_x()<<endl;  }

int main ( ) {
    A o1;
    cout<<o1.get_x()<<endl;
    afisare(o1);
    return 0;
}
```

Exemplu – Ce se afiseaza?

```
Const 7 // obiect o1
7 // o1.get_x()
10 // in functie ob.get_x()
Dest 10 // ob
Dest 7 // o1
```

2. Conceptele de clasa si obiect

2.3 Constructorii si destructorul unei clase. Constructorul de copiere

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.

2. Conceptele de clasa si obiect

2.3 Constructorii si destructorul unei clase. Constructorul de copiere

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*).

2. Conceptele de clasa si obiect

2.3 Constructorii si destructorul unei clase. Constructorul de copiere

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.

2. Conceptele de clasa si obiect

2.3 Constructorii si destructorul unei clase. Constructorul de copiere

Exemplu de necesitate redefinire constructor de copiere

```
class array {
      int *p;
      int size;
public:
      array(int sz) { p = new int[sz]; size = sz; }
      ~array() { delete [ ] p; }
      // copy constructor
      array(const array &);
};

array::array(const array &a) {
      size = a.size;
      p = new int[a.size];
      for(int i=0; i<a.size; i++) p[i] = a.p[i];
}
```

2. Conceptele de clasa si obiect

2.3 Constructorii si destructorul unei clase. Constructorul de copiere

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)

2. Conceptele de clasa si obiect

2.3 Constructorii si destructorul unei clase. Constructorul de copiere

Exemplu:

Functii care intorc obiecte

```
#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;
}
```

2. Conceptele de clasa si obiect

Pointeri catre obiecte. 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**.

- forma prescurtata = fara this

```
class clss2
{
    int x;
public:
    clss2(int i=0)     {
        this->x=i;
    }
    int get_x()     {
        return this->x;
    }
};

main()
{
    clss2 ob(10);
      cout<<ob.get_x(); // this->x = ob.x
}
```

2. Conceptele de clasa si obiect

2.4 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).

2. Conceptele de clasa si obiect

2.4 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

2. Conceptele de clasa si obiect

2.4 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;
}
```

2. Conceptele de clasa si obiect

2.4 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;
}
```

2. Conceptele de clasa si obiect

2.4 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;
}
```

2. Conceptele de clasa si obiect

2.4 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;
}
```

2. Conceptele de clasa si obiect

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)

2. Conceptele de clasa si obiect

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;
}
```

Perspective

Cursul 4:

4. Supraîncărcarea funcțiilor și operatorilor în C++.
4.1 Clase și funcții friend.
4.2 Supraîncărcarea funcțiilor.
4.3 Supraîncărcarea operatorilor cu funcții friend.
4.4 Supraîncărcarea operatorilor cu funcții membru.
4.5 Observații.