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 2

09/10/2019

Agenda cursului

1. Recapitularea discutiilor din cursul anterior.
2. Completări aduse de C++ faţă de C.
3. Principiile programarii orientate pe obiecte.
4. Conceptele de clasa și obiect.

1. Recapitularea discutiilor din cursul anterior

1.1 Regulamente UB si FMI

1.2 Obiectivele disciplinei

Obiectivul general al disciplinei:
Formarea unei imagini generale, preliminare, despre programarea orientată pe obiecte (POO).

Obiective specifice:
1. Înțelegerea fundamentelor paradigmei programarii orientate pe obiecte;
2. Înțelegerea conceptelor de clasă, interfață, moștenire, polimorfism;
3. Familiarizarea cu şabloanele de proiectare;
4. Dezvoltarea de aplicații de complexitate medie respectând principiile de dezvoltare ale POO;
5. Deprinderea cu noile facilităţi oferite de limbajul C++.

1. Recapitularea discutiilor din cursul anterior

1.3 Programa cursului

1. Principiile programarii orientate pe obiecte
2. Proiectarea ascendenta a claselor. Incapsularea datelor in C++
3. Supraincarcarea functiilor si operatorilor in C++
4. Proiectarea descendenta a claselor. Mostenirea in C++
5. Constructori si destructori in C++
6. Modificatori de protectie in C++. Conversia datelor in C++
7. Mostenirea multipla si virtuala in C++
8. Membrii constanti si statici ai unei clase in C++
9. Parametrizarea datelor. Sabloane in C++. Clase generice
10. Parametrizarea metodelor (polimorfism). Functii virtuale in C++. Clase abstracte
11. Controlul tipului in timpul rularii programului in C++
12. Tratarea exceptiilor in C++
13. Recapitulare, concluzii
14. Tratarea subiectelor de examen

1.4 Bibliografia cursului

1. Recapitularea discutiilor din cursul anterior

1.5 Regulament de notare si evaluare

Curs si laborator: fiecare cu 2 ore pe săptămâna.
Seminar: 1 ora pe saptamana.

Nota laborator = media celor 3 lucrari practice (proiecte);
Nota test practic (colocviu);
Nota test scris;
Toate cele 3 probe de evaluare trebuie sa aiba note >=5.

Nota finala se calculeaza ca medie ponderata intre notele obtinute la cele 3 evaluari, ponderile cu care cele 3 note intra in medie fiind:

25% - nota pe lucrarile practice (proiecte)

25% - nota la testul practic

50% - nota la testul scris.

2 Completări aduse de C++ faţă de C

Extras din Programa cursului

2. Recapitulare limbaj C (procedural) și introducerea în programarea orientată pe obiecte.
2.1 Funcții, transferul parametrilor, pointeri.
2.2 Deosebiri între C și C++.
2.3 Supradefinirea funcțiilor, Operații de intrare/ieșire, Tipul referință, Funcții în structuri.

3. Proiectarea ascendenta a claselor. Incapsularea datelor in C++.
3.1 Conceptele de clasa și obiect. Structura unei clase.
3.2 Constructorii și destructorul unei clase.
3.3 Metode de acces la membrii unei clase, pointerul this. Modificatori de acces în C++.
3.4 Declararea și implementarea metodelor în clasă și în afara clasei.

2 Completări aduse de C++ faţă de C

Intrări şi ieşiri

Limbajul C++ furnizează obiectele cin şi cout, în plus faţă de funcţiile scanf şi printf din limbajul C. Pe lângă alte avantaje, obiectele cin şi cout nu necesită specificarea formatelor.

```
// operator este o functie care are ca nume un simbol (sau mai multe simboluri)
int main()
{int x,y,z;
   cin >>x;  // operatorul  >>(cin,x) care  intoarce fluxul (prin referinta ) cin din care s-a extras data x
   cin>>y>>z; //este de fapt  >>(>>(cin,y), z)

cout<<x; // operatorul  <<(cout, x ) -intoarce fluxul cout (prin referinta) in care s-a inserat x
cout<<y<<z; // este de fapt <<(<<(cout,y),z) -afiseaza y si z
```

2 Completări aduse de C++ faţă de C

Supraîncărcarea funcţiilor (un caz de *Polimorfism la compilare*)

Limbajul C++ permite utilizarea mai multor funcţii care au acelaşi nume, caracteristică numită supraîncărcarea funcţiilor. Identificarea lor se face prin numărul de parametri şi tipul lor.

Exemplu:

```
void afis (int a)
{
    cout<<”int”<<a;
}

void afis (char a)
{
    cout<<”char”<<a;
}
```

2 Completări aduse de C++ faţă de C

Funcţii cu valori implicite

In C++: Într-o funcţie se pot declara valori implicite pentru unul sau mai mulţi parametri. Atunci când este apelată funcţia, se poate omite specificarea valorii pentru acei parametri formali care au declarate valori implicite.

Argumentele cu valori implicite trebuie să fie amplasate la sfârşitul listei.

```
void f (int a, int b = 12){  cout<<a<<" - "<<b<<endl;}

int main(){
  f(1);
  f(1,20);

  return 0;
}
```

2 Completări aduse de C++ faţă de C

Funcţii cu valori implicite

Valorile implicite se specifică o singură dată în definiţie (de obicei în prototip).

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

void f (int a, int b = 12); // prototip cu mentionarea valorii implicite pentru b
int main()
{
  f(1);
  f(1,20);
return 0;
}

void f (int a, int b) {  cout<<a<<" - "<<b<<endl; }
```

2 Completări aduse de C++ faţă de C

Alocare dinamica

C

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int *a,*b;
  a = (int *)malloc(sizeof(int));
  b = (int *)malloc(4 * sizeof(int));
  b = (int *) realloc(b,7 * sizeof(int));
  free(a);
  free(b);

  return 0;
}
```

C++

```
#include <iostream>
#include <stdlib.h>
using namespace std;

int main()
{
  int *a,*b;
  a = (int *)malloc(sizeof(int));
  b = (int *)malloc(4 * sizeof(int));
  b = (int *)realloc(b,7 * sizeof(int));
  free(a);
  free(b);
  a = new int(); // valoare oarecare
  cout<<*a<<endl;
  delete a;
  a = new int(22);
  cout<<*a<<endl;
  delete a;
  b = new int[4];
  delete[] b;
  return 0;
}
```

2 Completări aduse de C++ faţă de C

Tipul referinta

O referinţă este, in esenta, un pointer implicit, care actioneaza ca un alt nume al unui obiect (variabila).

```
int a;
int *p;

int & ref = a; //ref este alt nume pentru variabila a
p=&a;  // p este adresa variabilei a
*pi=3;   //in zona adresata de p se pune valoarea 3
```

Pentru a putea fi folosită, o referinţă trebuie iniţializată in momentul declararii, devenind un alias (un alt nume) al obiectului cu care a fost iniţializată.

2 Completări aduse de C++ faţă de C

Tipul referinta

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

int main()
{
  int a = 20;
  int& ref = a;
  cout<<a<<" "<<ref<<endl; // 20 20
  ref++;
  cout<<a<<" "<<ref<<endl; // 21 21
  return 0;
}
```

Obs: spre deosebire de pointeri care la incrementare trec la un alt obiect de acelasi tip, incrementarea referintei implica, de fapt, incrementarea valorii referite.

2 Completări aduse de C++ faţă de C

Tipul referinta

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

int main()
{
  int a = 20;
  int& ref = a;
  cout<<a<<" "<<ref<<endl; // 20 20
  int b = 50;
  ref = b;
  ref--;
  cout<<a<<" "<<ref<<endl; // 49 49
  return 0;
}
```

Obs: in afara initializarii, nu puteti modifica obiectul spre care indica referinta.

2 Completări aduse de C++ faţă de C

Restrictii pentru referinte:

1. o referinta trebuie să fie initializata când este definita, dacă nu este membra a unei clase, un parametru de functie sau o valoare returnata;
2. referintele nule sunt interzise intr-un program C++ valid.
3. nu se poate obtine adresa unei referinte.
4. nu se pot crea tablouri de referinte.
5. nu se poate face referinta catre un camp de biti.

2 Completări aduse de C++ faţă de C

Transmiterea parametrilor

C

```
void f(int x){   x = x *2;}

void g(int *x){  *x = *x + 30;}

int main()
{
  int x = 10;
  f(x);
  printf("x = %d\n",x);
  g(&x);
  printf("x = %d\n",x);
  return 0;
}
```

C++

```
#include <iostream>
using namespace std;
void f(int x){   x = x *2;} //prin valoare
void g(int *x){  *x = *x + 30;} // prin pointer
void h(int &x){ x = x + 50;} //prin referinta

int main()
{
  int x = 10;
  f(x);
  cout<<"x = "<<x<<endl;
  g(&x);
  cout<<"x = "<<x<<endl;
  h(x);
  cout<<"x = "<<x<<endl;
  return 0;
}
```

2 Completări aduse de C++ faţă de C

Transmiterea parametrilor

Observatii generale

- parametrii formali - sunt creati la intrarea intr-o functie si distrusi la retur;
- apel prin valoare - copiaza valoarea unui argument intr-un parametru formal ⇒ modificarile parametrului nu au efect asupra argumentului;
- apel prin referinta - in parametru este copiata adresa unui argument ⇒ modificarile parametrului au efect asupra argumentului.
- functiile, cu exceptia celor de tip void, pot fi folosite ca operand in orice expresie valida.

2 Completări aduse de C++ faţă de C

Transmiterea parametrilor

Regula generala: in C o functie nu poate fi tinta unei atribuiri.
In C++ - se accepta unele exceptii, permitand functiilor respective sa se gaseasca in membrul stang al unei atribuiri.

```
char s[10] = "Hello";
char& f(int i) {  return s[i];}

int main() {
f(2) = 'X';
  cout<<s;
  return 0;
}
```

2 Completări aduse de C++ faţă de C

Transmiterea parametrilor

Cand tipul returnat de o functie nu este declarat explicit, i se atribuie automat int.

Tipul trebuie cunoscut inainte de apel.

```
f (double x)
{
      return x;
}
```

Prototipul unei functii: permite declararea in afara si a numarului de parametri / tipul lor:

```
void f(int); // antet / prototip
int main() { cout<< f(50); }
void f( int x)
{
      // corp functie;
}
}
```

2 Completări aduse de C++ faţă de C

Functii in structuri

C

```
#include <stdio.h>
#include <stdlib.h>
struct test
{
  int x;
  void afis()
  {
    printf("x= %d",x);
  }
}A;

int main()
{
    scanf("%d",&A.x);
    A.afis(); /* error ‘struct test’ has
no member called afis() */
    return 0;
}
```

C++

```
#include <iostream>
using namespace std;
struct test
{
  int x;
  void afis()
  {
    cout<<"x= "<<x;
  }
}A;

int main()
{
    cin>>A.x;
    A.afis();
    return 0;
}
```

2 Completări aduse de C++ faţă de C

STL (Standard Template Library)

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

int main()
{
  vector<int> v;
  v.push_back(10);
  v.push_back(20);
  v.push_back(30);
  vector<int>::iterator i;
  for(i = v.begin();i!=v.end();i++)
  cout<<*i<<" ";
  return 0;
}
```

3 Principiile programarii orientate pe obiecte

O **clasa** defineste atribute si metode.

```
class X{
    //variabile
    //functii
};
```

Un **obiect** este o instanta a unei clase care are o anumita stare (reprezentata prin valoare) si are un comportament (reprezentat prin functii) la un anumit moment de timp.

Un **program orientat obiect** este o colectie de obiecte care interactioneaza unul cu celalalt prin mesaje (aplicand o metoda).

3 Principiile programarii orientate pe obiecte

Principalele concepte care stau la baza POO sunt:

Abstractizarea

Incapsularea

Modularitatea

Ierarhizarea.

3 Principiile programarii orientate pe obiecte

Abstractizarea

- procesul de grupare a datelor şi metodelor de prelucrare specifice rezolvării unei probleme.

- identifică trăsăturile caracteristice esenţiale ale unui obiect, care îl deosebesc de toate celelalte feluri de obiecte.

- oferă o definire precisă a graniţelor conceptuale ale obiectelor din perspectiva unui privitor extern.

3 Principiile programarii orientate pe obiecte

Abstractizarea

Tipul abstract de date - Curs

```
struct Curs
{
string denumire;
int nr_ore;
};   // Instantiere: Curs c = {“POO”,2}
```

Tipul abstract de date - Punct

```
struct punct
{ int x,y;
void afis() {cout<<x<<” “<<y;}
}; // Instantiere punct p;
p.x = 7; p.y = 9; p.afis();
```

3 Principiile programarii orientate pe obiecte

Incapsularea

- ascunderea de informaţii (data-hiding);
- separarea aspectelor externe ale unui obiect care sunt accesibile altor obiecte de aspectele interne ale obiectului care sunt ascunse celorlalter obiecte;
- defineşte apartenenţa unor proprietăţi şi metode faţă de un obiect;
- doar metodele proprii ale obiectului pot accesa starea acestuia.

3 Principiile programarii orientate pe obiecte

Incapsularea(ascunderea informatiei)

foarte importanta
public, protected, private

Avem acces?	**public**	**protected**	**private**
Aceeasi clasa	da	da	da
Clase derivate	da	da	nu
Alte clase	da	nu	nu

3 Principiile programarii orientate pe obiecte

Incapsularea(ascunderea informatiei)

```
class Test
{
  private:
int x;
  public:
  void set_x(int a) {x = a;}
};

int main()
{
  Test t;
  t.x = 34; // inaccesibil
  t.set_x(34); // ok
  return 0;
}
```

3 Principiile programarii orientate pe obiecte

Modularitatea

- partiţionarea programului în module care pot fi compilate separat, dar care au conexiuni cu alte module ale programului;

- modulele servesc ca şi containere în care sunt declarate clasele şi obiectele programului.

3 Principiile programarii orientate pe obiecte

Ierarhizarea

- o ordonare a abstracţiunilor.

Principalele tipuri sunt:

- **Moştenirea** (ierarhia de clase) relaţie între clase în care o clasă mosteneste structura şi comportarea definită în una sau mai multe clase (relaţie de tip “is a” sau “is like a”);
- **Agregarea** (ierarhia de obiecte) compunerea unui obiect din mai multe obiecte mai simple. (relaţie de tip “has a”)

3 Principiile programarii orientate pe obiecte

Agregarea

```
class Profesor
{
string nume;
int vechime;
};

class Curs
{
  string denumire;
  Profesor p;
};
```

3 Principiile programarii orientate pe obiecte

Mostenirea

```
class Curs
{
  string denumire;
  Profesor p;
};

class Curs_optional : public Curs
{
  int an;
};
```

3 Principiile programarii orientate pe obiecte

Mostenirea multipla

```
class Forma
{
  string denumire;
  public:
  void afis()
  {
    cout<<"baza";
  }
};
```

```
class Patrat: public
Forma
{
  int latura;
  public:
  void afis()
  {
    cout<<"patrat";
  }
};
```

```
class Dreptunghi:
public Patrat
{
  int lungime;
  public:
  void afis()
  {

cout<<"dreptunghi";
  }
};
```

3 Principiile programarii orientate pe obiecte

Mostenirea din baza multipla

```
class Imprimanta
{  };

class Scaner
{  };

class Multifunctionala: public Imprimanta,
public Scaner
{   };
```

3 Principiile programarii orientate pe obiecte

Mostenirea in diamant

```
class Produs
{ };

class Perisabil: virtual public Produs
{ };

class Promotie: virtual public Produs
{ };

class Perisabil_la_promotie: public Perisabil, public Promotie
{ };
```

3 Principiile programarii orientate pe obiecte

Polimorfism

La compilare

- supraincarcare de functii
- supraincarcarea de operatori
- sabloane

La executie

- functii virtuale.

4. Conceptele de clasa si obiect

4.1. 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.

4. Conceptele de clasa si obiect

4.1. 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.

4. Conceptele de clasa si obiect

4.1. 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();
};
```

4. Conceptele de clasa si obiect

4.1. Structura unei clase

Exemplu – specificatori de acces (*sursa H. Schildt*):

```
class myclass {
public:
      int i, j, k; // accessible to entire program
};

int main()
{
      myclass a, b;
      a.i = 100; // access to i, j, and k is OK
      a.j = 4;
      a.k = a.i * a.j;
      b.k = 12; // remember, a.k and b.k are different
      cout << a.k << " " << b.k;
return 0;
}
```

4. Conceptele de clasa si obiect

4.1. Structura unei clase

Struct vs Class

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

Union vs Class *(detalii in cursul viitor)*

- 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.

4. Conceptele de clasa si obiect

4.2 Constructorii si destructorul unei clase.

- 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).

4. Conceptele de clasa si obiect

4.2 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).

4. Conceptele de clasa si obiect

4.2 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 (=)
}
```

4. Conceptele de clasa si obiect

4.2 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();
}
```

4. Conceptele de clasa si obiect

4.2 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;
}
```

4. Conceptele de clasa si obiect

4.2 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;
}
```

4. Conceptele de clasa si obiect

4.2 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!

4. Conceptele de clasa si obiect

4.2 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 };
```

4. Conceptele de clasa si obiect

4.2 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;
}
```

4. Conceptele de clasa si obiect

4.2 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
```

4. Conceptele de clasa si obiect

4.2 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
}
```

4. Conceptele de clasa si obiect

4.2 Constructorii si destructorul unei clase.

Exemplu – Ce se afiseaza?

```
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;
}
```

4. Conceptele de clasa si obiect

4.2 Constructorii si destructorul unei clase.

```
class cls { public:
    cls() { cout << "Inside constructor 1" << endl; }
    ~cls() { cout << "Inside destructor 1" << endl; } };

class clss {
   cls xx;
public:
    clss() { cout << "Inside constructor 2" << endl; }
    ~clss() { cout << "Inside destructor 2" << endl; } };

class clss2 {
    clss xx;
    cls xxx;
public:
    clss2() { cout << "Inside constructor 3" << endl; }
    ~clss2() { cout << "Inside destructor 3" << endl; } };

int main() {
    clss2 s;
}
```

Exemplu – Ce se afiseaza?

Perspective

Cursul 3:

Proiectarea ascendenta a claselor. Incapsularea datelor in C++.

Conceptele de clasa și obiect.
3.3 Metode de acces la membrii unei clase, pointerul this. Modificatori de acces în C++.
3.4 Declararea și implementarea metodelor în clasă și în afara clasei.