Facultatea de Matematică şi Informatică
Universitatea din Bucureşti

Programare orientată pe obiecte

- suport de curs -

Andrei Păun
Anca Dobrovăț

An universitar 2019 – 2020
Semestrul II
Seriile 13, 14 şi 21

Curs 2

25-26/10/2020

Cuprinsul cursului

- Recapitularea discuţiilor din cursul anterior (Generalităţi despre curs, Reguli de comportament)
- Diferențe cu C
- Generalităţi despre OOP (Principiile programării orientate pe obiecte)

Principiile programării orientate pe obiecte

- Obiecte
- Clase
- Moștenire
- Ascunderea informației
- Polimorfism
- Șabloane

Obiecte

- au stare şi acțiuni (metode/funcţii)
- au interfață (acțiuni) și o parte ascunsă (starea)

- Sunt grupate în clase, obiecte cu aceleași proprietăți
- Un **program orientat obiect** este o colecție de obiecte care interactionează unul cu celălalt prin mesaje (aplicand o metodă).

Clase

- menționează proprietățile generale ale obiectelor din clasa respectivă
- clasele nu se pot “rula”
- folositoare la encapsulare (ascunderea informației)
- reutilizare de cod: moștenire

Moștenire

- multe obiecte au proprietăți similare
- reutilizare de cod

Ascunderea informației

foarte importantă
public, protected, private

Avem acces?	**public**	**protected**	**private**
Aceeaşi clasă	da	da	da
Clase derivate	da	da	nu
Alte clase	da	nu	nu

Polimorfism

- tot pentru claritate/ cod mai sigur
- Polimorfism la compilare: ex. max(int), max(float)
- Polimorfism la execuție: RTTI

Șabloane

- din nou cod mai sigur/reutilizare de cod
- putem implementa listă înlănțuită de
  - întregi
  - caractere
  - float
  - obiecte

Privire de ansamblu pentru C++

- Bjarne Stroustrup în 1979 la Bell Laboratories in Murray Hill, New Jersey
- 5 revizii: 1998 ANSI+ISO, 2003 (corrigendum), 2011 (C++11/0x), 2014, 2017 (C++ 17/1z)
- Următoarea plănuită în 2020 (C++2a)

- Versiunea 1998: Standard C++, C++98

- C++98: a definit standardul inițial, toate chestiunile de limbaj, STL
- C++03: bugfix o unică chestie nouă: value initialization
- C++11: initializer lists, rvalue references, moving constructors, lambda functions, final, constant null pointer, etc.
- C++14: generic lambdas, binary literals, auto, variable template, etc.

- C++17:
- If constexpr()
- Inline variables
- Nested namespace definitions
- Class template argument deduction
- Hexadecimal literals
- etc

typename is permitted for template template parameter declarations (e.g.,

```
template<template<typename> typename X> struct ...
```

Diferențe cu C

- <iostream> (fără .h)
- int main() (fără void)
- using namespace std;
- cout, cin (fără &)
- // comentarii pe o linie
- declarare variabile

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 - functie care are ca nume un simbol (sau mai multe simboluri)

```
int main()
{int x,y,z;
   cin >>x;  // operator>>(cin,x) care  intoarce fluxul (prin referinta ) cin
din care s-a extras data x
   cin>>y>>z; //  operator>>(operator>>(cin,y), z)

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

```
#include <iostream>
using namespace std;
int main()
{
        int i;
        cout << "This is output.\n"; // this is a single line comment
        /* you can still use C style comments */
        // input a number using >>
        cout << "Enter a number: ";
        cin >> i;
        // now, output a number using <<
        cout << i << " squared is " << i*i << "\n";
        return 0;
}
```

```
#include <iostream>
using namespace std;
int main( )
{
        float f;
        char str[80];
        double d;
        cout << "Enter two floating point numbers: ";
        cin >> f >> d;
        cout << "Enter a string: ";
        cin >> str;
        cout << f << " " << d << " " << str;
        return 0;
}
```

- citirea string-urilor se face pană la primul caracter alb
- se poate face afișare folosind toate caracterele speciale \n, \t, etc.

Variabile locale

```
/* Incorrect in C89. OK in C++. */
int f()
{
        int i;
        i = 10;
        int j; /* aici problema de compilare in C */
        j = i*2;
        return j;
}
```

```
#include <iostream>
using namespace std;
int main()
{
        float f;
        double d;
        cout << "Enter two floating point numbers: ";
        cin >> f >> d;
        cout << "Enter a string: ";
        char str[80]; // str declared here, just before 1st use
        cin >> str;
        cout << f << " " << d << " " << str;
        return 0;
}
```

C++ vechi vs C++ nou

- fără conversie automată la int

```
func(int i)
{
        return i*i;
}
```

```
int func(int i)
{
        return i*i;
}
```

- nou tip de include
- using namespace

Tipul de date bool

- se definesc true şi false (1 si 0)
- C99 nu îl definește ca bool ci ca _Bool (fără true/false)
- <stdbool.h> pentru compatibilitate

2 versiuni de C++; diferente: Noile include

- <iostream> <fstream> <vector> <string>
- **math.h** este **<cmath>**
- **string.h** este **<cstring>**
- **math.h deprecated, a se folosi cmath**

Clasele in C++

```
#define SIZE 100
// This creates the class stack.
class stack {
        int stck[SIZE];
        int tos;
public:
        void init();
        void push(int i);
        int pop();
};
```

- init(), push(), pop() sunt funcţii membru
- stck, tos: variabile membru

- se creează un tip nou de date stack mystack;
- un obiect instanţiază clasa
- funcțiile membru sunt date prin semnătură
- pentru definirea fiecărei funcții se folosește ::

```
void stack::push(int i)
{
        if(tos==SIZE) {
                cout << "Stack is full.\n";
                return; }
        stck[tos] = i;
        tos++;
}
```

- :: scope resolution operator
- şi alte clase pot folosi numele push() şi pop()
- după instantiere, pentru apelul push()

```
stack mystack;
```

- mystack.push(5);
- programul complet în continuare

```
#include <iostream>
using namespace std;
#define SIZE 100
// This creates the class stack.
class stack {
        int stck[SIZE];
        int tos;
public:
        void init();
        void push(int i);
        int pop();
};
void stack::init()
{
        tos = 0;
}
void stack::push(int i) {
if(tos==SIZE) {
        cout << "Stack is full.\n";
        return;
        }
        stck[tos] = i;
        tos++;
}
```

```
int stack::pop()
{
        if(tos==0) {
          cout << "Stack underflow.\n";
          return 0;
        }
        tos--;
        return stck[tos];
}
int main()
{
        stack stack1, stack2; // create two stack objects
        stack1.init();
        stack2.init();
        stack1.push(1);
        stack2.push(2);
        stack1.push(3);
        stack2.push(4);
        cout << stack1.pop() << " ";
        cout << stack1.pop() << " ";
        cout << stack2.pop() << " ";
        cout << stack2.pop() << "\n";
        return 0;
}
```

Encapsulare

- următorul cod nu poate fi folosit în main()

```
stack1.tos = 0; // Eroare, tos este definit private.
```

Overloading de funcții

- polimorfism
- simplicitate/corectitudine de cod

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

// abs is overloaded three ways
int abs(int i);
double abs(double d);
long abs(long l);

int main()
{
        cout << abs(-10) << "\n";
        cout << abs(-11.0) << "\n";
        cout << abs(-9L) << "\n";
        return 0;
}

int abs(int i)
{
        cout << "Using integer abs()\n";
        return i<0 ? -i : i;
}
```

```
double abs(double d)
{
        cout << "Using double abs()\n";
        return d<0.0 ? -d : d;
}

long abs(long l)
{
        cout << "Using long abs()\n";
        return l<0 ? -l : l;
}
```

```
Using integer abs()
10
Using double abs()
11
Using long abs()
9
```

overload de funcții

- Același nume
- diferența e în tipurile de parametri
- tipul de întoarcere nu e suficient pentru a face diferența
- se poate folosi şi pentru funcții complet diferite (nerecomandat)
- overload de operatori: mai târziu

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;
}
```

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; }
```

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 = 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;
}
```

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.

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.

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.

o referinta trebuie să fie initializata când este definita, dacă nu este membra a unei clase, un parametru de functie sau o valoare returnata;

referintele nule sunt interzise intr-un program C++ valid.

nu se poate obtine adresa unei referinte.

nu se pot crea tablouri de referinte.

nu se poate face referinta catre un camp de biti.

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;
}
```

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.

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;
}
```

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;
}
}
```

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;
}
```

Moștenirea

- încorporarea componentelor unei clase în alta
- refolosire de cod
- detalii mai subtile pentru tipuri şi subtipuri
- clasă de bază, clasă derivată
- clasa derivată conține toate elementele clasei de bază, mai adăugă noi elemente

```
class building {
        int rooms;
        int floors;
        int area;
public:
        void set_rooms(int num);
        int get_rooms();
        void set_floors(int num);
        int get_floors();
        void set_area(int num);
        int get_area();
};

// house e derivată din building
class house : public building {
        int bedrooms;
        int baths;
public:
        void set_bedrooms(int num);
        int get_bedrooms();
        void set_baths(int num);
        int get_baths();
};
```

tip acces: public, private, protected
mai multe mai târziu

public: membrii publici ai building devin publici pentru house

- house NU are acces la membrii privați ai lui building
- așa se realizează encapsularea

- clasa derivată are acces la membrii publici ai clasei de baza şi la toți membrii săi (publici şi privați)

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

class building {
        int rooms;
        int floors;
        int area;
public:
        void set_rooms(int num);
        int get_rooms();
        void set_floors(int num);
        int get_floors();
        void set_area(int num);
        int get_area();
};
// house is derived from building
class house : public building {
        int bedrooms;
        int baths;
public:
        void set_bedrooms(int num);
        int get_bedrooms();
        void set_baths(int num);
        int get_baths();
};
```

```
// school este de asemenea derivată din building
class school : public building {
        int classrooms;
        int offices;
public:
        void set_classrooms(int num);
        int get_classrooms();
        void set_offices(int num);
        int get_offices();
};
```

```
void building::set_rooms(int num)
{ rooms = num; }
void building::set_floors(int num)
{ floors = num; }
void building::set_area(int num)
{ area = num; }
int building::get_rooms()
{ return rooms; }
int building::get_floors()
{ return floors; }
int building::get_area()
{ return area; }
void house::set_bedrooms(int num)
{ bedrooms = num; }
void house::set_baths(int num)
{baths = num; }
int house::get_bedrooms()
{ return bedrooms; }
int house::get_baths()
{ return baths; }
void school::set_classrooms(int num)
{ classrooms = num; }
void school::set_offices(int num)
{ offices = num; }
int school::get_classrooms()
{ return classrooms; }
int school::get_offices()
{ return offices; }
```

```
int main()
{
        house h;
        school s;
        h.set_rooms(12);
        h.set_floors(3);
        h.set_area(4500);
        h.set_bedrooms(5);
        h.set_baths(3);
        cout << "house has " << h.get_bedrooms();
        cout << " bedrooms\n"; s.set_rooms(200);
        s.set_classrooms(180);
        s.set_offices(5);
        s.set_area(25000);
        cout << "school has " << s.get_classrooms();
        cout << " classrooms\n";
        cout << "Its area is " << s.get_area();
        return 0;
}
```

house has 5 bedrooms
school has 180 classrooms
Its area is 25000

Moștenire

- terminologie
  - clasă de bază, clasă derivată
  - superclasă subclasă
  - părinte, fiu
- mai târziu: funcții virtuale, identificare de tipuri in timpul rulării (RTTI)

Constructori/Destructori

- inițializare automată
- obiectele nu sunt statice
- constructor: funcție specială, numele clasei
- constructorii nu pot întoarce valori (nu au tip de întoarcere)

```
// Aceste linii creează clasa stack.
class stack {
        int stck[SIZE];
        int tos;
public:
        stack(); // constructor
        void push(int i);
        int pop();
};

// constructorul clasei stack
stack::stack()
{
        tos = 0;
        cout << "Stack Initialized\n";
}
```

- constructorii/destructorii sunt chemați de fiecare dată o variabilă/obiect de acel tip este creată/distrusă. Declarații active nu pasive.
- Destructori: reversul, execută operații când obiectul nu mai este folositor
- memory leak

stack::~stack()

```
// Creăm clasa stack.
class stack {
        int stck[SIZE];
        int tos;
public:
        stack(); // constructor
        ~stack(); // destructor
        void push(int i);
        int pop();
};

// constructorul clasei stack
stack::stack()
{
        tos = 0;
        cout << "Stack Initialized\n";
}

// destructorul clasei stack
stack::~stack()
{
        cout << "Stack Destroyed\n";
}
```

```
// Using a constructor and destructor.
#include <iostream>
using namespace std;
#define SIZE 100
// This creates the class stack.
class stack {
        int stck[SIZE];
        int tos;
public:
        stack(); // constructor
        ~stack(); // destructor
        void push(int i);
        int pop();
};
// stack's constructor
stack::stack()
{
        tos = 0;
        cout << "Stack Initialized\n";
}
// stack's destructor
stack::~stack()
{
        cout << "Stack Destroyed\n";
}
void stack::push(int i){
        if(tos==SIZE) {
                cout << "Stack is full.\n";
                return;
        }
        stck[tos] = i;
        tos++;
}
int stack::pop(){
        if(tos==0) {
                cout << "Stack underflow.\n";
                return 0;
        }
        tos--;
        return stck[tos];
}
int main(){
        stack a, b; // create two stack objects
        a.push(1);
        b.push(2);
        a.push(3);
        b.push(4);
        cout << a.pop() << " ";
        cout << a.pop() << " ";
        cout << b.pop() << " ";
        cout << b.pop() << "\n";
        return 0;
}
```

```
Stack Initialized
Stack Initialized
3 1 4 2
Stack Destroyed
Stack Destroyed
```

Clasele în C++

- cu “class”
- obiectele instanţiază clase
- similare cu struct-uri şi union-uri
- au funcții
- specificatorii de acces: public, private, protected
- default: private
- protected: pentru moștenire, vorbim mai târziu

```
class nume_clasă {
        private variabile şi funcţii membru
specificator_de_acces:
        variabile şi funcţii membru
specificator_de_acces:
        variabile şi funcţii membru
// ...
specificator_de_acces:
        variabile şi funcţii membru
} listă_obiecte;
```

- putem trece de la public la private şi iar la public, etc.

```
class employee {
        char name[80]; // private din oficiu
public:
        void putname(char *n); // acestea sunt  publice
        void getname(char *n);
private:
        double wage; // acum din nou  private
public:
        void putwage(double w); // înapoi la public
        double getwage();
};
```

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

- se folosește mai mult a doua variantă
- un membru (ne-static) al clasei nu poate avea inițializare
- nu putem avea ca membri obiecte de tipul clasei (putem avea pointeri la tipul clasei)
- nu auto, extern, register