UNIVERSITATEA BABEŞ-BOLYAI
Facultatea de Matematică şi Informatică

Programare orientată obiect

Curs 03

Laura Dioşan

POO

- Elemente de bază ale limbajului C++
  - Fişiere de IO
- Clase
  - Declarare
  - Constructor/destructor
  - Metode
  - Utilizare (static şi dinamic)
  - Clase ca şi date membre
  - UML

Fişiere de Intrare/Ieşire

- Procesarea fişierelor la 2 nivele:
  - Nivel inferior
    - Acces direct la SO
  - Nivel superior
    - Utilizarea unor funcţii speciale de intrare-ieşire

Procesarea fişierelor

- Deschiderea unui fişier
- Citirea dintr-un fişier
- Scrierea într-un fişier
- Accesarea unui fişier
- Închiderea unui fişier

Procesarea la nivel superior

- header <fstream>
  - ifstream
  - ofstream

- a se consulta exemplul din
  - 03/files/appFiles.cpp

POO

- Dezvoltarea proiectării sistemelor care presupune utilizarea:
  - obiectelor – unităţi de bază, fiecare obiect fiind o instanţă a unei clase
  - clase – relaţionate prin relaţii de compoziţie şi moştenire

POO – concepte de bază

- clasă
  - defineşte în mod abstract caracteristicile unui lucru:
    - caracteristici (atribute, câmpuri, proprietăţi)
    - comportament (metode, operaţii, trăsături)
  - implementarea unui TAD
- obiect
  - instanţă (obiectul actual creat în timpul rulării) a unei clase
  - variabilă de tip clasă
- metodă
  - modalitate de comunicare între obiecte
  - toate metodele formează interfaţa unui obiect

Dezvoltarea TAD-urilor în C++

- TAD
  - Exportarea unui nume (un tip de date)
  - Definirea unui domeniu de valori pentru date
  - Definirea unei interfeţe (operaţiile TAD-ului)
  - Restricţionarea accesului la componentele TAD-ului (acces doar prin intermediul operaţiilor)
  - Ascunderea implementării unui TAD

POO - caracteristici

- Încapsulare
  - capacitatea de a comprima datele cu operaţiile
  - ascunderea implementării → controlul accesului
- Moştenire
  - Reutilizarea codului (folosirea şi îmbunătţirea codului unei clase)
- Polimorfism
  - Permite un comportament adaptat contextului

Definirea clasei

- Similar unei date de tip ***struct***:

```
class <nume_clasă>{
        [<protecţie>:] listă_membri;
};
```

- Protecţie:
  - ***private***
    - Nu se permite nimănui accesul la membri din afară clasei
    - Excepție: clasele ***friend*** cu clasa X pot accesa datele private ale lui X
  - ***protected***
    - Nu se permite nimănui accesul la membri din afară clasei
    - Excepție: clasele derivate din clasa X pot accesa datele protected ale lui X
  - ***public***
    - Se permite oricui accesul la membri din afară clasei

- Lista de elemente membre
  - date membre
  - metode membre

Definirea clasei

- Membri
  - date
  - metode

```
class <class_name>{

        [<protection>:]
                <type> <identif>;


        [<protection>:]
                <return_type> <fc_name>(<lfp>);
};
```

Clase – definirea metodelor

- imediat, în interiorul clasei ➔ metode *inline*
  - definiţie completă (antet + corp)
  - pentru metodele simple, fără bucle

```
class Flower{
private:
    char* name;
    int price;
public:
    int getPrice(){
        return this->price;
    }
};
```

- în afara clasei -> prin utilizarea operatorului de rezoluţie/scop ***::***
  - antetul metodei în interiorul clasei
  - corpul metodei în afara clasei

```
class Flower{
private:
    char* name;
    int price;
public:
    int getPrice();
};

int Flower::getPrice(){
    return this->price;
}
```

Obiecte

- Declararea obiectelor

  <class_name> <identif>;

  - Alocarea şi iniţializarea unui obiect
    - la declarare
    - prin apelul **automat** al ***constructor***-ului
  - De-alocarea
    - la sfârşitul programului/funcţiei
    - prin apelul **automat** al ***destructor***-ului
- Utilizarea obiectelor
  - Componentele clasei sunt accesate:
    - direct – în interiorul metodelor membre ale clasei
    - prin folosirea operatorului "." – în alte metode (care nu sunt membre); operatorul este precedat de un obiect
    - prin folosirea operatorului "->" – operatorul este precedat de o referinţă a unui obiect
    - prin folosirea operatorului "::" – în alte metode (care nu sunt membre); operatorul este precedat de numele unei clase

```
class Flower{
private:
    char* name;
    int price;
public:
    static int no;

    char* getName(){
        return this->name;
    }
    int getPrice();
};

int Flower::getPrice(){
    return price;
}

int Flower::no = 0;

void main(){
    Flower f;
    cout << f.getPrice();
    cout << Flower::no;
}
```

Constructorul unei clase

- Constructor – o metodă publică specială:
  - numele metodei = nume_clasă
  - fără tip de return (nici măcar ***void***)
  - scop:
    - rezervă spaţiu pentru datele membre
    - iniţializează datele membre
  - apelat **automat** la declararea unui obiect

- Destructor – o metodă publică specială:
  - numele metodei = ~nume_clasă
  - fără tip de return (nici măcar ***void***)
  - scop:
    - eliberează spaţiul ocupat de datele membre
  - apelat **automat** la sfarţitul domeniului de vizibilitate
    - la sfârşitul programului → pentru obiectele globale
    - la sfârşitul funcţiei → pentru obiectele locale

Constructorul unei clase

- Fiecare clasă trebuie să conţină cel puţin 2 constructori:
  - constructor implicit
  - constructor de copiere
- Alţi constructori posibili
  - constructor cu parametri
  - constructor de conversie

Constructor implicit

- fără argumente sau cu argumrente implicite
- rezervă spaţiu pentru datele membre
- iniţializează datele membre cu valori implicite
- este apelat automat la declararea unui obiect

```
class Flower{
private:
    char* name;
    int price;
public:
    Flower(){
        cout << "implicit construtor" << endl;
        name = new char[10];
        price = 0;
    }

    Flower(int p = 0){
        cout << "implicit construtor 2" << endl;
        name = new char[10];
        price = p;
    }
```

```
class <class_name>{
   …
   public:
      <class_name>();
      <class_name>(<implicit_arg>);
   …
};
```

Constructor general (cu parametri)

- cu argumente
- rezervă spaţiu pentru datele membre
- iniţializează datele membre cu valori date

class <class_name>{
  …
  public:
    <class_name>(<type1 arg1>, <type2 arg2>, …);
  …
};

```
class Flower{
private:
    char* name;
    int price;
public:
    Flower() { ... }
    Flower(int p = 0) { ... }

    Flower(char* n, int p){
        cout << "construtor with param" << endl;
        name = new char[strlen(n) + 1];
        strcpy(name, n);
        price = p;
    }
```

Constructor de copiere

- crează o copie a unui obiect
- este apelat când:
  - un obiect este transmis ca parametru prin valoare
  - un obiect este returnat prin valoare
  - se iniţializează un obiect cu un alt obiect

```
class <class_name>{
   ...
   public:
     <class_name>(<class_name> &o);
     <class_name>(<class_name> &o, <impl_arg>);
     <class_name>(const <class_name> &o);
     <class_name>(const <class_name> &o, <impl_arg>);
   ...
};
```

```
class Flower{
private:
     char* name;
     int price;
public:
     Flower() { ... }
     Flower(int p = 0) { ... }
     Flower(char* n, int p) { ... }

     Flower(Flower &f){
          cout << "copy construtor" << endl;
          if (f != NULL){
               name = new char[strlen(f.name) + 1];
               strcpy(name, f.name);
               price = f.price;
          }
     }
```

Constructor de copiere

- crează o copie a unui obiect
- este apelat când:
  - un obiect este transmis ca parametru

  prin valoare
  - un obiect este returnat prin valoare
  - se iniţializează un obiect cu un alt obiect

- dacă nu există, compilatorul produce un contructor de copiere implicit care realizează o copie bit cu bit a obiectului curent → probleme cu datele alocate dinamic

- alternative ale contructorului de copiere
  - prevenirea transmiterii prin valoare

Constructor de conversie

- construieşte un obiect de tipul clasei curente dintr-un obiect/variabilă de alt tip

```
class <class_name>{
   ...
   public:
     <class_name>(<other_type> &o);
     <class_name>(<other_type> &o, <impl_arg>);
     <class_name>(const <other_type> &o);
     <class_name>(const <other_type> &o, <impl_arg>);
   ...
};
```

```
class Plant{
private:
    char* plantname;
    int value;
public:
    Plant();
    ~Plant();
};
```

```
class Flower{
private:
    char* name;
    int price;
public:
    Flower() { ... }
    Flower(int p = 0) { ... }
    Flower(char* n, int p) { ... }
    Flower(Flower &f) { ... }

    Flower(Plant &p){
        name = new char[strlen(p.plantname) + 1];
        strcpy(name, p.plantname);
        price = p.value;
    }
```

Destructor

- Eliberează resursele de memorie alocate de constructor
  - memoria alocată în heap
  - conexiunile cu baze de date
  - conexiunile în reţea
  - fişierele deschise
- Este unic
- Nu returnează nimic
- Nu are parametri

```
class <class_name>{
   ...
   public:
     ~<class_name>();
   ...
};
```

```
class Flower{
private:
    char* name;
    int price;
public:
    Flower() { ... }
    Flower(int p = 0) { ... }
    Flower(char* n, int p) { ... }
    Flower(Flower &f) { ... }
    Flower(Plant &p) { ... }

    ~Flower(){
        cout << "destrutor" << endl;
        if ( name != NULL){
            delete[] name;
              name = NULL;
        }
    }
```

Forma standard a unei clase

- Date
  - atributele clasei
- Metode
  - Constructor implicit
  - Constructor de copiere
  - Destructor
  - Operator de atribuire (a se consulta cursul următor)
  - Alte metode

Obiecte ca şi parametri

- Dacă obiectul nu-şi schimbă valoarea în interiorul funcţiei, el va fi apelat ca parametru ***const***

```
Flower::Flower(const Flower &f){
    cout << "copy constructor" << endl;
    name = new char[strlen(f.name) + 1];
    strcpy(name, f.name);
    price = f.price;
}
```

```
void change(Flower& f){
    f.setName("daisy");
}
```

- Dacă o metodă nu modifică obiectul apelant, metoda va fi ***const***

```
class Flower{
private:
    char* name;
    int price;
public:
    Flower();
    Flower(char* name, int p);
    Flower(const Flower &f);
    ~Flower();

    char* getName() const{
        return name;
    }

    int getPrice();
    void setName(char* n);
    void setPrice(int p);
    char* toString();
    bool compare(Flower &f);
};
```

Obiecte ca şi parametri

- Obiectele sunt transmise prin referinţă pentru a evita apelarea contructorului de copiere

```
void withRef(Flower &f){
    char* s = f.toString();
    cout << s << endl;
    if (!s){
        delete[] s;
        s = NULL;
    }
}
```

```
void withoutRef(Flower f){
    char* s = f.toString();
    cout << s << endl;
    if (!s){
        delete[] s;
        s = NULL;
    }
}
```

- Un obiect de ieşire este returnat prin ***valoare*** când el a fost creat ca obiect temporar în interiorul funcţiei

```
Flower newFlower(){
    Flower f("daisy", 10);
    return f;
}

Flower& newFlowerRef(){
    Flower f("daisy", 10);
    return f;
}
```

```
void main(){
    Flower f1 = newFlower();
    cout << f1.toString();  //daisy;10;
    Flower f2 = newFlowerRef();
    cout << f2.toString();  //error
}
```

Exemplu – clasă simplă

- Clasa Flower
  - Date membre:
    - Name
    - Price
  - Metode:
    - Constructors
    - Destructor
    - Set-ers
    - Get-ers
    - toString
    - Compare
- a se consulta exemplul din
  - 03/class

Exemplu – o clasă mai complexă

- Este nevoie de un constructor implicit pentru clasa *Flower*
  - Dacă clasa *Flower* nu are un constructor implicit, compilatorul va sintetiza unul
  - Dacă clasa *Flower* are alte tipuri de constructori ➔ erori

```
class Flower{
private:
    char* name;
    int price;
public:
    Flower(char* name, int p);
    Flower(const Flower &f);
    Flower(char* s);
    //Flower(Plant &p);
    ~Flower();
    char* getName();
    int getPrice();
    void setName(char* n);
    void setPrice(int p);
    char* toString();
    bool compare(Flower &f);

};
```

```
class Gardener{
private:
    char* name;
    Flower f;
public:
    Gardener();
    Gardener(char* n, Flower& f);
    Gardener(char* s);
    Gardener(const Gardener &g);
    ~Gardener();
    void setName(char* n);
    void setFlower(Flower &f);
    char* getName();
    Flower& getFlower();
    char* toString();
    bool compare(Gardener g);
};
```

```
void main(){
    Gardener g; //error
}
```

Temă

- Scrieţi un program pentru a defini şi utiliza:
  - clasa Student (PIN, nume, vârstă) şi
  - scrieţi un program de test pentru această clasă

- Scrieţi un program pentru a defini şi utiliza:
  - clasa Punct(x, y),
  - clasa Segment (punct1, punct2) şi
  - clasa Figură (nrDeSegmente, segmente).
  - Scrieţi un program de test pentru aceste clase

Cursul următor

- Clase
  - Elemente prietene
  - Supraîncărcarea operatorilor