Cursul 4 - CLASE în C++

- Declarație clasă
- Date membru
- Functii membru
  - Manageri: constructori, destructor
  - Functii de acces
  - Implementori (operatori)
  - Functii ajutatoare
  - Functii membru const
- Pointerul this
- Prietenii unei clase
- Constructor si operatorul new
- Destructor si operatorul delete
- Copiere si atribuire

CLASE

- O clasă în C++ se caracterizează prin:
  - Un nume - identificator
  - O colecție de *date membru* –prin aceste date este reprezentat un obiect. Datele sunt create pentru fiecare obiect (instanță a clasei)
  - O colecție de *funcții membru* – tratamente (metode) ce se aplică obiectelor. Un trartament 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 clasă cu **`class`**:

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

```
class <Nume_clasa> {
        //Declaratii membri privati (nerecomandat)
public:
        // Declaratii functii (eventual date)
protected:
        // Declaratii date (eventual metode)
private:
        // Declaratii date (eventual metode)
};
```

CLASE - Declarație

- Declarație clasă cu **`struct`**:

```
struct <Nume_clasa> {Corp_clasa};

struct <Nume_clasa> {
        // Declaratii membri public (nerecomandat)
public:
        // Declaratii functii (eventual date)
protected:
        // Declaratii date (eventual metode)
private:
        // Declaratii 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 şi în clasele derivate
- Domeniul private:
  - Componenta este vizibilă doar în funcțiile clasei şi în funcțiile şi clasele prieten

Exemplul 1

```
//FILE stiva1.h
class Stack {
public:
   enum { MaxStack = 50 };
   void init() { top = -1; }
   void push( int ) ;
   int pop() ;
   bool isEmpty() { return top < 0; }
   bool isFull() { return top >= MaxStack - 1; }
   void dump() ;
private:
   int top;
   int arr[ MaxStack ];
};
```

Exemplul 2

```
#include <iostream>

using std::ostream;

class Data {
 public:
  Data(int o_zi = zi_curenta(), int o_luna = luna_curenta(),
    int un_an = an_curent());
  ~Data() {}
  void aduna_zi(int n);
  void aduna_luna(int n);
  void aduna_an(int n);
  Data operator ++(int); // postfix
  Data& operator ++();    // prefix
  void set_zi(int zi_noua);
  void set_luna(int );
  void set_an(int an_nou);
```

Exemplul 2

```
  int get_zi() const;
  int get_luna() const;
  int get_an() const;
  friend ostream& operator << (ostream&, const Data&);
  static Data azi();
 private:
  int zi, luna, an;
  static int zi_curenta();
  static int luna_curenta();
  static int an_curent();
  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(1,4,2003);
// Apel constr. Data(unsigned,unsigned,unsigned)
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
  - În afara clasei (precedat de **`inline`** pentru implementare inline):

```
<tip> <Nume_clasa>::<Nume_functie>(<Lista_parametri>)
{
        // Corp functie
}
```

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.set_an(2006);
azi.set_luna(3);
azi.set_zi(12);
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:

  `Data::get_zi(), Data::set_an()`

- Implementori:

  `void Data::aduna_zi()`

- Operatori :

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

CLASE – Funcții membru

- Funcții ajutătoare:

```
bool Data::esteAnBisect(int)
```

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

```
int Data::get_zi() const{return zi;}
int Data::get_an() 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.get_an();
   d.aduna_an(2);
   int j = cd.get_an();
   cd.aduna_an(1); // Eroare: cd este obiect const!
   //…
}
```

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

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

CLASE - Pointerul implicit **this**

- Pentru fiecare funcție membru există o singură instanță accesibilă fiecărui obiect
- Pointerul **this** – pointer implicit accesibil doar în funcțiile membru ale unei clase (structuri); pointează la obiectul pentru care este apelată funcția

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

```
Data azi(12,3,2006);
cout << azi.get_luna() << endl;
cout << azi.get_luna(&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; Echivalent
    //(*this).luna = m; Echivalent
}
Data& Data::aduna_an(int n) {
    if(zi==29 && luna==2 && !esteAnBisect(an+n){
       zi = 1; luna = 3;
    }
    an += n;
    return (*this);
}
d.aduna_zi(1).aduna_luna(1).aduna_an(1);
```

CLASE - Pointerul implicit **this**

```
#include <iostream>
using namespace std;
class ExThis{
    static int X;
    double Y;
public:
    ExThis() :Y(0.0){
        cout << "Obiect " << X++  ;
        cout << " : this = " << this << "\n";
   }
   ~ExThis(){
        cout << "Se distruge:" << --X << " this = " << this << "\n";
    }
};
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";
}
```

CLASE - 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

*/
```

CONSTRUCTORI

- Constructorii unei clase

```
void Data::init(int z, int l, int a)
   {zi = z; luna = l; an = a;}
Data d; d.init(12,3,2006);
```

- Constructor: o funcție membru cu "misiunea" de a inițializa obiectele declarate. Dacă o clasă are constructori, obiectele clasei sunt inițializate la declarare.

```
Data azi(2);
Data maine(3, 4);
Data acum;
Data Craciun("25 Decembrie");
Data d(12, 3, 2006);
Data d = Data(12, 3, 2006);
```

CONSTRUCTORI

- Constructor implicit: poate fi apelat fără parametri

```
Data(int z = 0, int l = 0, int a = 0);
Data(int z = ziua(), int l = luna(), int a = anul());
```

- 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(12, 3, 2006);// 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: creare la (execuția ajunge la) declarare şi distrugere la ieşirea din domeniul său de vizibilitate;
- Obiect în memoria heap: este creat cu **`new`** şi distrus cu **`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 odată la începutul programului şi distrus la terminarea programului
- Obiect temporar: este creat ca parte a evaluării unei expresii şi distrus la sfârşitul expresiei în care apare

PRIETENII UNEI CLASE

- O declarație de funcție membru înseamnă:
  - Funcția poate accesa partea privată a clasei;
  - Funcția este în domeniul de vizibilitate a clasei;
  - Funcția poate fi invocată de un obiect (are un pointer **`this`**)
- Declarare unei funcții membru **`static`** îi conferă doar primele două proprietăți
- Declararea unei funcții membru **`friend`**, îi conferă doar prima proprietate

```
Class Vector{}; class Matrice{}
Vector operator*(constMatrice& m, const Vector& v){
   Vector r;
   // Implementare r = m*v
   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
#include <iostream.h>
struct X;
// Declaratie necesara pentru definitia lui Y

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

struct X { // Definitia clasei X
public:
  void initialize();
  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;
};
```

```
void X::initialize() {
  i = 10;
}

void g(X* x, int i) {
  x->i = i;
}

void Y::f(X* x) {
  x->i = 47;
}

struct Z {
private:
  int j;
public:
  void initialize();
  void g(X* x);
};
```

```
void Z::initialize() {
  j = 99;
}

void Z::g(X* x) {
  x->i += j;
  cout << "In Z::g(X& x), x->i = " << x->i << endl;
}

void h() {
  X x;
  x.i = 100; // Acces la membrul privat i
}

int main() {
  X x;
  x.initialize();
  Z z;
  z.initialize();
  z.g(&x);
}
/*
In Z::g(X& x), x->i = 109
*/
```

CLASE: Copiere şi asignare

- Inițializarea şi asignarea implicită înseamnă copiere membru cu membru a unui obiect Ob1 în Ob2:

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

CLASE: Copiere şi asignare

- Cazul în care nu avem alocare dinamică:

```
Data d1(12,3,2006);
Data d2 = d1; //echiv. cu Data d2(d1);
```

- este apelat automat un constructor 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 asignare

- Atenție la clasele ce conțin pointeri! Folosirea onstructorului 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 asignare
}
```

```
#include <iostream.h>
class TipTablou{
    const char* s;
        //
};
class Tablou{
public:
    Tablou(int s=10){
        p = new TipTablou[marime = s];
        cout <<++i<< " Tablou()\n";}
    ~Tablou() {
        delete[] p; cout <<i--<<  " ~Tablou()\n";}
    Tablou(const Tablou& );
    Tablou& operator=(const Tablou&);
private:
    TipTablou* p;
    int marime;
    static int i;
};
```

```
Tablou::Tablou(const Tablou& t){
    p = new TipTablou[marime = t.marime];
    for (int k = 0; k < marime ; k++)
        p[k] = t.p[k];
    cout << "Copiere ob: "<< i << endl;
}

Tablou& Tablou::operator=(const Tablou& t){
    if(this != &t){ // evitare asignare t = t
        delete[] p;
        p = new TipTablou[marime = t.marime];
        for (int k = 0; k < marime ; k++)
                p[k] = t.p[k];
    }
    cout << "s-a asignat ob: " << i << endl;
    return *this;
}
```

```
int Tablou::i = 0;
void h(){
   Tablou x1(6);
   Tablou x2 = x1;
   Tablou x3;
   x3 = x1;
}
void main(){ h();}
/*
1 Tablou()
Copiere ob: 1
2 Tablou()
s-a asignat ob: 2
2 ~Tablou()
1 ~Tablou()
0 ~Tablou()
*/
```

CLASE: Copiere şi asignare

- Constructorul de copiere se apelează
  - 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

```
#include <iostream>
using namespace std;
class Clasa1{
    int i;
public:
    Clasa1() : i(13) {}
    Clasa1(const Clasa1 &a) : i(a.i+1) {cout << " copie ";}
    int get_i() {return i;}
};
Clasa1 f(Clasa1 a){
    cout << a.get_i() << "\n";
    return a;
}
int main(){
    Clasa1 a;
    Clasa1 a1=a;// Apel copiere
    cout << a1.get_i() << endl;
    a1 = f(a1);// Apel copiere de 2 ori
    cout << a1.get_i() << endl;
    return 0;
}
/* copie 14 copie 15 copie 16  */
```