Facultatea de Automatică și Calculatoare

Universitatea Politehnica Timișoara

Programare orientată pe obiecte

Curs 4

Funcții constructor cu parametri

- **Se pot transmite parametri către funcţiile constructor (folosiţi mai ales la iniţializări)** v. **Programul1**
- **pasări de parametri :**

  clasa_exemplu ob(7,77) (mai des utilizată)

  sau

  clas_exemplu ob=clasa_exemplu(7,77) (există totuşi mici diferenţe referitoare la copierea funcţiei constructor, vezi mai departe la curs)

```
class clasa_exemplu {
int a, b;
public:
clasa_exemplu(int i, int j) { a = i; b = j; }
void arata() { cout << a << " " << b << "\n"; }
};

int main()
{
clasa_exemplu a(7, 77);
a.arata();
_getch();
return 0;
}
```

Funcții constructor cu parametri

- exemplu mai complet **Program2**

- **utile** deoarece permit evitarea unui apel de iniţializare
- caz special : funcţii constructor cu un parametru : exemplu **Program3**

Execuția constructorilor și destructorilor

- **Constructorul este apelat la declararea obiectului iar destructorul la distrugere dar:**
  - **constructorii sunt apelaţi în ordinea în care obiectele sunt declarate, de la stânga la dreapta în cadrul aceleiaşi instrucţiuni; destructorii sunt executaţi în ordine inversă**
  - **constructorii de obiecte globale sunt executaţi înaintea lui main(), în ordinea în care au fost întâlniţi, de la stânga la dreapta şi de sus în jos (în cadrul aceluiaşi fişier); destructorii globali sunt executaţi în ordinea inversă, după încheierea main()**
  - **e greu de depistat ordinea în cazul fişierelor diferite**
- **exemplu *Program4***

Membrii de tip static ai claselor

- **Membrii statici de tip date :**
  - **precedate de cuvântul-cheie static vor exista într-un singur exemplar**
  - **în cadrul unei clase nu se alocă memorie pentru acestea de aceea trebuie definire globală în afara clasei**
  - **exemplu *Program5***

```
class exemplu_static {
    static int a;// aici e o variabila membru de tip static
    int b;
public:
    void seteaza(int i, int j) { a = i; b = j; }
    void arata();
};

//definirea globala pentru variabila de tip static
int exemplu_static::a;

void exemplu_static::arata()
{
cout << "Acesta este a static: " << a << "\n";
cout << "Acesta este b ne-static: " << b << "\n";
}

// programul principal

int main()
{
exemplu_static ob1, ob2;// doua obiecte de tip exemplu_static

ob1.seteaza(1, 2);// a va fi 1, b va fi 2
ob1.arata();// sa vedem daca chiar asa e

ob2.seteaza(3, 4);// setari in celalt obiect, diferite
ob2.arata();// sa verificam !

ob1.arata();// dar vedem ca a s-a modificat si pentru ob1
// ceea ce e firesc deoarece pur si simplu a este
//  acelasi pentru toate obiectele
_getch();
return 0;

}
```

```
Acesta este a static: 1
Acesta este b ne-static: 2
Acesta este a static: 3
Acesta este b ne-static: 4
Acesta este a static: 3
Acesta este b ne-static: 2
```

Membrii de tip static ai claselor – continuare 1

- o variabilă static există înainte de a fi creat orice obiect din clasă de aceea poate dobândi valoare oricând exemplu Program6

```
Valoarea initiala a lui a : 789
Valoarea lui a din c1 : 789
```

```
class comun {
public:
    static int a;// static si public totodata
};

int comun::a;// definirea variabilei statice

// programul principal

int main()
{
//
// initializarea variabilei statice inainte de crearea vreunui obiect
//folosind operatorul ::
comun::a = 789;
cout << "Valoarea initiala a lui a : " << comun::a << "\n";

comun c1;// abia acum se creeaza un obiect de tip comun
cout << "Valoarea lui a din c1 : " << c1.a;

_getch();
   return 0;
}
```

Membrii de tip static ai claselor – continuare 2

- uzual sunt folosite la controlul accesului la resurse comune (semafoare) **exemplu Program 7**

```
class semafor {
    static int resursa;// aici e o variabila membru de tip static
public:
    int ocupa_resursa();
    void elibereaza_resursa() { resursa = 0; }
};

int semafor::resursa;// aici marcam ocuparea (in variabila statica)

int semafor::ocupa_resursa()
{
   if (resursa) return 0;// marcam resursa ocupata
   else {
            resursa = 1;// ocupam resursa
            return 1;// si transmitem apelantului
        }
}
```

```
// programul principal

int main()
{
    semafor s1, s2;// doua obiecte de tip semafor
  if (s1.ocupa_resursa())
          cout << "Obiectul 1 a ocupat resursa.\n";
  if (!s2.ocupa_resursa())
          cout << "Obiectul 2 nu are acces la resursa.\n";
  s1.elibereaza_resursa();// eliberarea resursei
  if  (s2.ocupa_resursa())
          cout << "Obiectul 2 poate acum folosi resursa.\n";
_getch();
   return 0;
}
```

```
Obiectul 1 a ocupat resursa.
Obiectul 2 nu are acces la resursa.
Obiectul 2 poate acum folosi resursa.
```

Membrii de tip static ai claselor – continuare 3

- **Funcţii membre statice :**
  - restricţii :
    - pot să aibă acces doar la alţi membri statici
    - nu pot avea pointer this (se va vedea mai departe la curs)
    - nu pot coexista versiuni statice şi nestatice ale aceleiaşi funcţii
  - funcţiile membre de tip static au o utilizare limitată, spre exemplu la iniţializarea datelor de tip static exemplu **Program8**

Operatorul de specificare a domeniului

- **Folosit la asocierea unui nume de clasă cu un nume de membru dar permite de asemenea accesul la un nume dintr-un domeniu, dublat de un acelaşi nume printr-o declarare locală (Program 8_2):**

```
…
int i;        // i global
void f()
{
        int i;        // i local
        …
        i=19;
        …
        ::i =7;      // aici e i global
}
```

Clase imbricate, clase locale

- **Se poate defini o clasă în interiorul altei clase (*clase imbricate* - rar folosite*)*** valabilă doar în interiorul clasei care o conţine
- **O clasă definită în interiorul unei funcţii este *locală* - exemplu Program 9**
- **Nu putem face referiri la clasa locală în afara funcţiei**
- **Restricţii :**
  - **toate funcţiile membre trebuie definite în interiorul declaraţiei clasei**
  - **clasa locală nu poate folosi variabile locale ale funcţiei, cu excepţia variabilelor statice din interiorul funcţiei**
  - **în interiorul unei clase locale nu pot fi declarate variabile statice**
- **Clasele locale sunt de asemenea rar folosite**

Unele aspecte privind manipularea obiectelor

Transmiterea obiectelor către funcţii

- **Obiectele pot fi pasate către funcţii ca oricare alte variabile utilizând mecanismul standard, de copiere - dar constructorul nu este apelat la crearea noului obiect deoarece ar distruge valorile care trebuie transmise; însă destructorul este apelat la încheierea funcţiei - exemplu *Program10***
- Câteodată apare problema distrugerii obiectului original dacă se foloseşte alocare dinamică - prevenită dacă se foloseşte ***constructorul de copiere***

Returnarea obiectelor

- **o funcţie poate returna un obiect în modulul de program apelant exemplu - exemplu *Program11***
- la returnarea obiectului este creat automat un obiect temporar care conţine valoarea returnată; după transmiterea valorii obiectul temporar este distrus dar dacă acesta are un destructor care eliberează memorie alocată dinamic, aceasta va fi eliberată chiar dacă obiectul care primeşte valoarea o mai foloseşte; acest dezavantaj este prevenit prin supraîncărcarea operatorului de atribuire şi definirea constructorului de copiere

Unele aspecte privind manipularea obiectelor

- **Atribuirea obiectelor - se poate realiza între două obiecte de acelaşi tip**
  - **exemplu *Program12***

Capitolul 4. Matrice, pointeri și referințe

- **Matrice de obiecte**
  - sintaxa la fel ca la oricare alt tip de variabilă exemplu **Program13**

```
class cl {
int i;
public:
void pune_i(int j) { i = j; }
int da_i() { return i; }
};

int main()
{
cl mo[3];
int i;
for (i = 0; i < 3; i++)  mo[i].pune_i(i + 1);
for (i = 0; i < 3; i++)
cout << mo[i].da_i() << "\n";
_getch();
return 0;
}
```

Capitolul 4. Matrice, pointeri și referințe

Inițializare

- Constructor cu un singur parametru (Program14)

```
class cl {
  int i;
public:
  cl(int j) { i = j; }
  int da_i() { return i; }
};

int main()
{
  cl mo[3] = { 1, 2, 3 }; //  initializare
  int i;
  for (i = 0; i < 3; i++)
  cout << mo[i].da_i() << "\n";

  return 0;
}
```

Capitolul 4. Matrice, pointeri și referințe

Inițializare

- constructori cu mai mulți parametrii (Program15)

```
class cl {
  int h;
  int i;
public:
  cl(int j, int k) { h = j; i = k; } // constructor
  int ia_i() { return i; }
  int ia_h() { return h; }
};

int main()
{
  cl mo[3] = {
  cl(1, 2),
  cl(3, 4),
  cl(5, 6)
}; //  initializare
int i;
for (i = 0; i < 3; i++) {
  cout << mo[i].ia_h();
  cout << ",";
  cout << mo[i].ia_i() << "\n";
}

 return 0;
}
```

Matrice inițializate/matrice neinițializate

- dacă dorim atât matrice iniţializate cât şi matrice neiniţializate, spre exemplu:

```
class cl {
    int i;
public:
    cl(int j) { i = j; }
    int da_i() { return i; }
};
...
cl a[ 9 ]; // eroare, constructorul necesita initializarea
```

- **de aceea este necesară *supraîncărcarea* funcţiei constructor:**

```
class cl {
  int i;
public:
  cl() { i = 0; } // apelare pentru matrice neinitializate
  cl(int j) { i = j; } //apelare pentru matrice initializate
```

Matrice inițializate/matrice neinițializate – continuare 1

- şi sunt valabile instrucţiunile:

```
c1 a1[3] = { 3, 5, 6 };   //  initializat
c1 a2[4];       //  neinitializat
```

Pointeri către obiecte

- Pot exista pointeri către obiecte (ca şi la alte tipuri)
- accesul către membrii unei clase cu ajutorul unui pointer cu ->

exemplu Program16

```
class cl {
  int i;
public:
  cl(int j) { i = j; }
  int da_i() { return i; }
};

int main()
{
  cl ob(23), *p;
  p = &ob;// da adresa lui ob
  cout << p->da_i(); // foloseste -> pentru a apela da_i()
  _getch();
  return 0;
}
```

Pointeri către obiecte – continuare 1

- **aritmetica pointerilor este relativă la tipul obiectului :**

exemplu Program17

```
class cl {
  int i;
public:
  cl() { i = 0; }
  cl(int j) { i = j; }
  int da_i() { return i; }
};

int main()
{
  cl ob[3] = { 1, 2, 3 };
  cl  *p;
  int i;

  p = ob;// preia inceputul matricei
  for (i = 0; i < 3; i++) {
  cout << p->da_i() << "\n";
  p++;// indica spre urmatorul obiect
}
 _getch();
 return 0;
}
```

Pointeri către obiecte – continuare 2

- se poate atribui unui pointer adresa unui membru public al unui obiect:

exemplu Program18

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

int main()
{
  acces_p poin(1);
  int *p;

  p = &poin.i;// se preia adresa lui poin.i

  cout << *p;// si se afiseaza cu acces prin pointer

  return 0;
}
```

Pointeri către obiecte – continuare 3

- verificarea tipului la pointeri în C++ : un pointer poate fi atribuit altuia doar dacă au tipuri compatibile - exemplu de eroare :

```
int *in;
float *fl;
...
fl=in;   // eroare
```

- Se poate elimina orice nepotrivire folosind un modelator (adică o conversie de tip) dar astfel se trece peste mecanismul de verificare a tipului din C++.
- În C++, se realizează o verificare mai strictă a tipurilor decît în C