Facultatea de Automatică și Calculatoare

Universitatea Politehnica Timișoara

Programare orientată pe obiecte

Curs 3

Constructori și destructori

- **Deoarece cerinţele de iniţializare sunt atât de uzuale încât s-a simţit nevoia creării unui mecanism special, C++ permite obiectelor *să se iniţializeze singure*, atunci când sunt create, prin intermediul unei funcţii speciale numite *constructor*.**
- **O funcţie constructor este o funcţie membră a clasei şi are acelaşi nume cu clasa.**
- **De exemplu, pentru clasa stiva putem prevedea o funcţie constructor, astfel :**

```
// declararea clasei stiva
class stiva {
   int stiv[SIZE];
   int virf_stiva;
public:
   stiva();    // constructor  (are acelasi nume ca si clasa)
   void pune(int );
   int scoate();
};
```

Constructori și destructori (continuare 1)

- Funcţia constructor **stiva()** nu are specificat nici un tip de returnat, deoarece în **C++** funcţiile constructor **nu pot să returneze valori**.
- Definirea funcţiei constructor pentru exemplul nostru :

```
// urmeaza definirea functiei constructor pentru stiva
stiva::stiva()
{
   virf_stiva=0;
   cout << "In regula, stiva este initializata\n";
}
```

Evident că în practica curentă nu vom realiza afişări în funcţia constructor. Acestea vor realiza pur şi simplu iniţializările necesare.

Constructori și destructori (continuare 2)

- Opusul constructorului (în acţiunea sa) este **destructorul**. În multe cazuri, un obiect va trebui să efectueze o anumită acţiune sau unele acţiuni atunci când nu mai este nevoie de el şi deci trebuie "distrus".
- Obiectele locale sunt create la intrarea în blocul în care sunt definite şi distruse când blocul este părăsit. Obiectele globale sunt distruse la terminarea programului. Când este distrus un obiect, aceasta se întâmplă prin apelarea unui destructor (dacă este definit vreunul).
- **Motive** pentru care un destructor devine **util** : spre exemplu, un obiect trebuie să elibereze memoria ocupată sau trebuie să închidă fişierele pe care le-a deschis.
- Destructorul are în **C++ acelaşi nume** ca şi constructorul (deci acelaşi nume cu clasa) însă precedat de caracterul ~.

Constructori și destructori (continuare 3)

- Pentru exemplul nostru stiva (**Program 3**), declararea clasei, conţinând şi destructor, va arăta astfel :

```
//  declararea clasei stiva cu constructor si destructor

class stiva {
        int stiv[SIZE];
        int virf_stiva;
public:
        stiva();// constructor
        ~stiva();// destructor
        void pune(int);
        int scoate();
};
```

```
//functia constructor
stiva::stiva() {
        virf_stiva = 0;
        cout << "In regula, stiva este initializata\n";
}

//functia destructor
stiva::~stiva()
{
        cout << "In regula, stiva e distrusa\n";
        _getch();
}
```

Capitolul 3. Clase și obiecte

- **Clasele**
  - create - cu ***class***
  - **o clasa** este o *abstractizare logica*, iar **un obiect** are o *existență fizică*
  - **sintaxa (similara cu cea de la structuri):**

class nume_clasa{

date si functii particulare (implicit)

specificator_de_acces:

date si functii

specificator_de_acces:

date si functii

…

specificator_de_acces:

date si functii

} lista de obiecte; ***// optionala***

Clasele (continuare 1)

- *Specificatori de acces :*
  - **public, protected, private (protected este folosit la mostenire !)**
  - se poate modifica specificatorul de clasa oricât de des (***Programul 4)***
  - ***dar nu se recomanda ca in program ci asa :***

```
class angajat {
   char nume[80];
   double salar;
public:
   void pune_nume(char *);
   void scoate_nume(char *);
   void pune_salar(double );
   double furnizeaza_salar();
};
```

Clasele (continuare 2)

- ***Restricții care se aplică membrilor clasei :***
  - o variabilă membru care nu este de tip ***static*** nu poate sa aibă o inițializare
  - nici un membru nu poate fi ***obiect*** al clasei care se declară (deși un membru poate fi un ***pointer*** către clasa care este declarată)
  - nici un membru nu poate fi declarat ***auto, extern*** sau ***register***

- ***Accesul la variabilele membre :***

  dacă se dorește, se poate direct - cu punct, ca si la funcțiile membre (v. ***Programul 5***)

Structuri și clase

- **O structură defineste un tip special de clasa (dar implicit membrii unei structuri sunt publici) - *Program 6*.**
- **Nu este nevoie la declarare de cuvintul-cheie *struct*.**

Exemplul poate fi rescris :

```
class sir_exemplu{
    char sir[80];           // implicit private
public:
    void face_sir(char *);     // aici nu mai e implicit public
    void arata_sir();
};
```

Justificari pentru posibilitatea definirii cu struct :

- **facilitate in plus**
- **translatarea programelor din C in C++**
- **permitem lui class evolutie libera (desi struct trebuie pastrat pentru compatibilitate)**

Se recomanda totuși folosirea class.

Uniuni și clase

- ***Union* definește un tip special de clasă**
  - toți membrii sunt implicit publici
  - poate să conțină constructor și destructor
  - toate elementele de date impart aceeași locație de memorie

- **Restricții :**
  - nu poate moșteni nici un alt tip de clasă
  - nu poate fi clasă de bază
  - nu poate avea funcții virtuale membre
  - nu poate avea membre variabile static
  - nu poate avea ca membru un obiect care are supraîncărcat operatorul =
  - nu poate avea membri obiecte care conțin constructori sau destructori

Uniuni anonime

- Uniunea anonimă este un tip special de uniune care **nu conține un nume de tip** și nici o variabilă nu poate fi declarată ca fiind de acel tip de uniune (dar comunică compilatorului că variabilele membre ale uniunii vor împărți aceeași locație, iar variabilele vor putea fi utilizate **fără operatorul punct**)

  (exemplu *Programul 7*)

- Variabilele din union sunt la același nivel de existență ca și oricare alte variabile locale din același bloc;

- **Restricții** : cele de la **union** plus :
  - elementele dintr-o uniune anonimă trebuie să fie **date**
  - nu pot conține elemente **private** sau **protected**
  - uniunile **globale** anonime trebuie să fie specificate ca fiind **static**

Funcții prietene

- Sunt funcții care **nu sunt membre** dar **au acces** la membrii particulari ai clasei
- **Sintaxa** : cuvântul cheie **friend** + **includerea prototipului** în declararea clasei (funcția prietenă este apelată normal, fără punct) **(exemplu *Programul 0.1*)**

```
class clasa_exemplu {
int a, b;
public:
    // iata functia prietena
     friend int suma(clasa_exemplu);
     void furniz_ab(int, int);
};

void clasa_exemplu::furniz_ab(int i, int j)
{
    a = i;
    b = j;
}
```

```
//definirea functiei prietene suma (care nu
// este membra a nici unei clase)

int suma(clasa_exemplu cl) {
            return cl.a + cl.b; // functia are acces la a si b
}

int main()
{
      clasa_exemplu c;
      c.furniz_ab(3, 4);
      cout << suma(c); // functia prietena este apelata fara punct
      _getch();
      return 0;
}
```

Funcții prietene

- **Utilitatea** utilizării funcțiilor prietene :
  - la **supraîncărcarea** anumitor operatori (mai încolo la curs)
  - la crearea anumitor **operatori de intrare/ieșire**
  - pentru a **corela** diverse secțiuni ale programului (**exemplu *Programul 0.2***: două clase diferite afișează mesaje de eroare, dar fiecare dorește să știe dacă mesajul este deja afișat pe ecran, pentru a nu se suprapune iar prezență în fiecare clasă a unei astfel de funcții membre ar fi redundantă)
- **Referire ulterioară**: clasa c2 apărea printre parametrii de apel ai funcției friend din c1 deci înainte de declararea c2 și a trebuit să apară **class c2**
- **O functie friend a unei clase poate fi membră a altei clase** (**exemplu *Programul 0.3*** care rescrie altfel Programul 0.2)

Funcții prietene (continuare)

- **Restricții** :
  - o clasă derivată nu poate moșteni funcții friend
  - o funcție friend nu poate avea un specificator de clasă de memorare (nu poate fi static sau extern)

- **Clase prietene** :
  - (**exemplu *Programul 0.4)***
  - o clasă friend are acces doar la numele definite in interiorul celeilalte, ea nu moștenește cealaltă clasă, deci membrii primei clase nu devin membri ai clasei friend

Funcţii inline

- **Sunt funcţii scurte care nu sunt apelate efectiv ci codul este inserat la fiecare folosire**
- definirea trebuie precedată de cuvântul-cheie ***inline***
- utilizate din motive de eficienţă
- se recomandă doar pentru coduri scurte
- *inline* este pentru compilator o solicitare, nu o comandă
- nu se pot insera *inline* funcţii recursive
- funcţiile *inline* pot fi membre ale unei clase
- este posibilă definirea funcţiilor *inline* direct într-o declarare de clasă (nu e necesar *inline*)
- funcţiile ***constructor*** şi ***destructor*** pot fi ***inline***