C++ osztályok

OOP és DB - OOP 1. óra

Structok

A korábbiakban mindenki megismerhette a C++-ban a struct-ok működését.

A struct egy felhasználó által definiált típus / adatszerkezet, amiben változók és jellemzően hozzájuk tartozó függvények találhatóak. Például:

```
struct Date {
    int d, m, y;
    void init(int dd, int mm, int yy); //inicializalashoz
    void add_year(int n);
    void add_month(int); //signature: nem muszaj nevet adni az arg-nak
    void add_day(int n);
};

void Date::init(int dd, int mm, int yy) {
    d = dd;
    m = mm;
    y = yy;
}
```

Structok

A structok legfőbb hátulütője, hogy alapértelmezett esetben nem valósítják meg az adatrejtés elvét.

Minden tagváltozó alapértelmezett esetben publikus, így kívülről is módosítható, ami nehezen követhető / inkonzisztens állapotokhoz vezethet.

```
Date mydate;
mydate.init(5, 12, 1977);
mydate.d = 32; // hupsz, inkonzisztens!
std::cout << mydate.y << "/" << mydate.m << "/" << mydate.d << std::endl;
```

Persze minden további nélkül létre lehet hozni privát tagokat is, de a structok esetében nem ez az alapértelmezett viselkedés.

C++ osztályok

Az osztályok **felhasználó által definiált típusok**, melyek / melyeknek:

- Tagváltozókat és tagfüggvényeket tartalmazhatnak
- Tagfüggvényei definiálják a létrehozás, inicializálás, másolás, mozgatás és törlés működését
- Tagfüggvényei felüldefiniálhatnak operátorokat (+, -, *, /, !, [])
- Tagváltozóik és tagfüggvényeik tekintetében saját névteret alkotnak
- Kívülről elérhető függvényeiket public, implementációjukat private / protected tagokkal valósítják meg. Alapértelmezett esetben az osztály minden tagja privát! (a structokkal ellentétben)

Példa: Date osztály deklarációja

```
class Date
{
    int d, m, y; //default: minden private
public:
    void init(int dd, int mm, int yy);
    void add_year(int n);
    void add_month(int); //signature: nem muszaj nevet adni az arg-nak
    void add_day(int n);
    void print();
};
```

Példa: Date osztály definíciója

Egy típus deklarációja megmondja a fordítónak, hogy példányosításkor mekkora memória-területre van szükség.

Ez alapján akár példányosítható is az objektum / típus (de implementáció nélkül nem feltétlenül használható).

> Ezért van, hogy ha egy .h fájlban van a deklaráció és .cpp fájlban a definíció, akkor egy másik .cpp fájlból (pl. main.cpp) elég a .h fájlt include-olni - a fordító nem fog panaszkodni.
>
> (A teljes használhatóságot majd a linker biztosítja, amikor a külön lefordított .cpp fájlokból keletkező object fájlokat összelinkeli)

Date osztály - metódusok definíciója

A metódusok definiálhatóak az osztályon belül, vagy rajta kívül (egy olyan fájlban, ahonnan a deklaráció látszik). Utóbbi esetben ügyelni kell arra, hogy az osztály saját névteret képez!

```
#include <iostream>

class Date
{
    int d, m, y; //default: minden private
public:
    void init(int dd, int mm, int yy);
    void add_year(int n);
    void add_month(int); //signature: nem muszaj nevet adni az arg-nak
    void add_day(int n);
    void print() { // belul definialt metodus...
        std::cout << "Date is: " << d << "/" << m << "/" << y << std::endl;
    }
};

// kivul definialt metodus:
void Date::init(int dd, int mm, int yy) {
    d = dd;
    m = mm;
    y = yy;
}
```

Date osztály - metódusok definíciója

```
class Date
{
    int d, m, y; //default: minden private
public:
    void init(int dd, int mm, int yy);
    void add_year(int n);
    void add_month(int); //signature: nem muszaj nevet adni az arg-nak
    void add_day(int n);
    void print() { // belul definialt metodus...
        std::cout << "Date is: " << d << "/" << m << "/" << y << std::endl;
    }
};
```

Date osztály - metódusok definíciója

```
// kivul definialt metodus:
void Date::init(int dd, int mm, int yy) {
    d = dd;
    m = mm;
    y = yy;
}

void Date::add_year(int n) {
    y = y + n;
}

void Date::add_month(int n) {
    m = m + n;
}

void Date::add_day(int n) {
    d += n;
}
```

```
int main()
{
    Date mydate;
    mydate.init(23, 7, 1985);
    mydate.print();

    mydate.add_year(1);
    mydate.add_month(3);
    mydate.add_day(2);
    mydate.print();

    std::cin.get();
    return 0;
}
```

Deklaráció és definíció szeparálási elve

Általában célszerű a header file-okba tenni az osztályok deklarációját, a definíciót pedig az osztályon kívül, egy külön fordítási egységbe (.cpp fájlba) tenni. Miért?

A program többi része csak a kisebb méretű header file-t kell, hogy include-olja - garantáltan nem másolódik be több fordítási egységbe u.az az implementáció

Ha egy osztály implementációja megváltozik, elég csak az adott forrás fájlt (.cpp) újra lefordítani, majd linkelni - a program többi részét nem kell újra fordítani.

Mikor lehet érdemes egy metódust mégis az osztály deklarációján belül implementálni?

Abban az esetben, ha nagyon rövid egy metódus törzse, gyakran mégis az osztály deklarációján belül (.h fájlban) szokás definiálni.

A fordító ilyenkor a metódust *inline*-olhatja. Ez azt jelenti, hogy a metódus meghívásának (összes) helyére bemásolhatja a metódus törzsét, így a metódus meghívásakor nem történik context switch (nem jelenik meg a stack-változók menedzselésével járó pluszköltség).

Ez egy lehetőség, de nem minden fordító él vele minden esetben.

Date osztály értékelése

Mi az, ami a jelenlegi Date osztályban jó és mi az ami rossz?

Ami jó, az jó

Pozitívum, hogy:

A Date osztályban megvalósul az adatrejtés

A nap, hónap és év tagváltozók privát elérésűek, vagyis csak az osztály implementációjából(*) érhetőek el

Ha egy dátum valaha is rosszul jelenik meg, arról csakis az osztály metódusai tehetnek (a debuggolás első lépése - a *lokalizáció* - már futtatás előtt megtörténik)

(*) Ez alól kivételt képeznek a friend osztályok és metódusok, amikről később lesz szó.

Ami rossz, azon változtatni kell

Hiányosság, hogy:

(Az implementáció kissé bárgyú, nem ellenőrizzük, hogy az adatok konzisztensek-e, de ezt most hagyjuk)

Az init() metódust nem kötelező meghívni. Mi történik, ha egy programozó (az osztály, mint interfész / API felhasználója) elfelejti?

```
int main()
{
    Date mydate;
    //...
    mydate.print(); // whoops!
```

Konstruktor

Erre a problémára nyújt megoldást a **konstruktor.**

A konstruktor az osztály tagfüggvénye, mely / melynek:

- neve megegyezik az osztály nevével
- nincs visszatérési értéke
- akárhány és akármilyen típusú argumentuma lehet (így egy osztálynak lehet akárhány konstruktora, csak mindegyiknek más számú és / vagy típusú argumentuma kell, hogy legyen)

Konstruktor - automatikus default konstruktor

Ha nem definiálunk egy konstruktort sem, a fordító automatikusan generál egy argumentum nélküli default konstruktort. Ekkor az osztály kinézetre példányosítható úgy, mint egy sima beépített típus - bár több, mint valószínű, hogy nem lesz jól inicializálva!

```
int main()
{
    Date mydate; // mivel nincs konstruktor, automatikusan generalodik egy default konstruktor
    // ugyanakkor ez semmi hasznosat nem fog csinalni
```

Bonyolult szabályok vannak arra, hogy ilyenkor az egyes tagváltozók milyen kezdeti értéket kapnak (hogyan inicializálja őket az automatikusan generált default konstruktor). Pl. függ attól, hogy statikus allokálásúak vagy a heap-en vannak.

Konstruktor - automatikus default konstruktor

Ráadásul ha a tagváltozók között nemcsak alaptípus van, hanem összetett típus is, amit nem lehet üres argumentum-listával inicializálni, az automatikusan generált default konstruktor nem lesz jó - hibát fog eredményezni.

Ezért - annak ellenére, hogy a fordító legenerál nekünk egy default konstruktort - **érdemes mindig saját konstruktort definiálni**!

> Még akkor is, ha a saját konstruktorunk sem vár egyetlen argumentumot sem, esélyes, hogy az általunk definiált konstruktor legalább helyesen tud működni.

Ha egy osztályban legalább 1 konstruktort definiálunk, a fordító már nem fog default konstruktort generálni, és kötelező az egyik létező konstruktort használni!

Konstruktor - automatikus konstruktorok

Halkan megjegyezzük, hogy ha nem definiálunk saját copy konstruktort (amiről később lesz szó), ilyet is automatikusan legenerál a fordító.

A copy konstruktor majd azt a célt szolgálja, hogy egy már létező objektum alapján példányosítsunk és inicializáljunk egy új objektumot.

A példára visszatérve: Egy lehetséges konstruktor

```
class Date
{
    int d, m, y; //default: minden private
public:
    Date(int dd, int mm, int yy);
    void add_year(int n);
    void add_month(int); //signature: nem muszaj nevet adni az arg-nak
    void add_day(int n);
    void print() { // belul definialt metodus...
        std::cout << "Date is: " << d << "/" << m << "/" << y << std::endl;
    }
};
```

Vesd össze:

```
class Date
{
    int d, m, y; //default: minden private
public:
    void init(int dd, int mm, int yy);
    void add_year(int n);
    void add_month(int); //signature: nem muszaj nevet adni az arg-nak
    void add_day(int n);
    void print() { // belul definialt metodus...
        std::cout << "Date is: " << d << "/" << m << "/" << y << std::endl;
    }
};
```

Konstruktor implementációja lehet kívül vagy belül

```
class Date
{
    int d, m, y; //default: minden private
public:
    Date(int dd, int mm, int yy);
    Date() { d = 1; m = 1; y = 1970; } // default konstruktor
    // ...
};

Date::Date(int dd, int mm, int yy) {
    d = dd;
    m = mm;
    y = yy;
}
```

Ha vannak konstruktorok, kötelező valamelyiket használni

```
int main()
{
    Date mydate(23, 7, 1985);
    Date mydate2; // ez is ok, mert van default konstruktor
    // ha nem lenne, hibat dobna a fordito
    mydate.print();
    mydate2.print(); // 1/1/1970
```