Programozási nyelvek I. C++.

8. előadás

A jegyzetet *Pataki Norbert* előadásán *Lanka Máté* készítette.
Előadás ideje: 2017.04.19. 12:00-14:00.

//12:40-kor időben kezdünk, de laptopos kivetítés van.

Ma az objektumorientáltsággal kezdünk foglalkozni, ezen fogalmait fogjuk áttekinteni.

Honnan jött ez, miért jött? Volt arról szó, hogy a C előtt megjelent a Simula64, de ezt az objektumorientált megközelítést a C sose vállalta fel. A *struct* ellenben különböző adatokat köt össze, ezt használta már a C. Ebben minden mező publikus volt. Az objektumorientáltságnál jól jön, hogy új dolgokat tudjunk definiálni. Valójában ehhez szervesen kapcsolódó műveleteket is támogat.

Ehhez nézünk mindjárt egy-két példát is. Mai napig ez az ipar legjelentősebb paradigmája, ez programozási nyelvtől függetlenül így van, akár C-ben, C#-ban, JAVA-ban is. Másik dolog, hogy ha visszagondolunk a hagyományos procedurális függvényekre, ott az a függvény, hogy beadunk paramétert, azt megváltoztatjuk, és visszaadunk valamit. Ellenben az OO azt jelenti, hogy egymás után állapotokat, változóhalmazon történő állapotokat tudunk változtatni. Valójában létrehoztunk valami objektumot egy állapotban, valamilyen műveletekkel ezeket változtatjuk.

Mai alkalommal egy dátumtípust szeretnénk megnézni, hogy ezt hogyan lehet elképzelni (a példakódot inkább nem írnám le teljes egészében). Elgondoljuk, hogy milyen lehetőségeink vannak erre típusokat, nyelvi eszközöket tekintve. Első dolog, hogy hogyan lehet erre használni saját dátumtípust? A főprogram elején létrehoztunk egy *date* típusú változót. Első alapvető különbség a C++ és a JAVA között, hogy előbbi azt mondja, egy *date* objektumot ugyanúgy otthagyhatok, lehet kezelni, mint bármelyik más változót, ez ugyanúgy ottlehet a stack-en. Míg JAVA-ban minden felhasználói változómat a heap-en kell létrehozni, ott foglalja a memóriát, míg a garbage collector fel nem szabadítja. A C++ azt az elvet vallja, hogy ha lehet, ezt próbáljuk meg a stack-en tartani. Miben különbözik? A stack-en olyan változók vannak, amelyek méretét futási időben tudni kell. Hogyan fogjuk reprezentálni ezt C++-ben azonban, illetve mekkora tárterületet kell foglalnia? Ezért kell az a sor, hogy *#include "date.h"*, ahol a dátum definíciója van. Itt úgy döntöttünk, hogy ezt három *int*-tel reprezentáljuk, egy-egy az évnek, hónapnak és napnak. Itt tehát három integer-t kell leallokálnia. Ez pedig egy dátum típusú objektum. Ezeket úgy csoportosítja, hogy vannak benne nem használt adatok. A két "szélén" vannak az adattagok, míg középen vannak a padding byte-ok, hogy ezt még gyorsabban tudja elérni.

Másik logika, hogy az a logika, hogy ez ugyanúgy fog viselkedni, mint bármelyik más típusunk. Pl. standard inputról beolvasható legyen, össze tudjuk hasonlítani mással, hogy kisebb / nagyobb. A C++ alapfilozófiája, hogy egy saját típusú objektum ugyanúgy kelljen viselkednie, mint bármely másik, nem kell a heap-en allokálni. Kifejezésekben tudjuk összehasonlítani, minden egyéb.

Ami a hangsúlyosabb rész, hogy van egy dátum osztályunk. C++ a *struct* mellett bevezeti a *class* fogalmát. Sok különbség nincs, az alapvető annyi volt, hogy C-ben még nem került fel az igénye, hogy

elrejtsük a mögöttes reprezentációt. C++ viszont azt mondja, hogy egy osztállyal valamilyen programozói entitást kell reprezentálni, de nem kell foglalkozni azzal, hogy mögötte mi van. Igazából az, hogy látnia kell, hány darab int van ott, az nem egy objektumorientált elképzelés. Azt mondja, hogy használunk egy dátumot, de kit érdekel, mi van mögötte? Van egy publikus interfészem, leírja, hogy miket tudunk használni, de hogy mi van mögötte, azzal nem kell foglalkozni. Sőt, lehetőség szerint ezt rejtsük el a világ elől, és értsük úgy, hogy persze a C++ bevezetése után ezeket privátnak is lehet deklarálni. Nem tartoznak a külvilágra, hogy ezt hogyan reprezentálom. Úgy döntöttünk, hogy három *int*-tel reprezentáljuk, de lehet, jövő héten átírjuk. Például Linuxon máshogy van reprezentálva. Technikailag van rá lehetőség. Másik technikai issue, hogy a compiler szeretné tudni, hány bájtos. Mivel privát adattagok vannak, nem lehet erre ráhivatkozni, csak a compiler belső logikája miatt szeretné tudni, hogy hány bájtot foglal el. Ezért mondhatjuk nyugodtan, hogy ne kelljen a hívónak azzal foglalkozni, mi van mögötte, mert változhat. (Nyilván a dátumot nem fogjuk kéthetente átírni,), de mást se nagyon szoktunk. Nem ez a lényeg. Az a lényeg, hogy azokat használjuk, ami kint van a publikus interfészen.

Nézzük, erre milyen lehetőségeink vannak.

A C++ megnézte magának azt a fogalmat, hogy *konstruktor*, hogy milyen adatok megadásával lehet létrehozni a típust. Az első dolog tehát, ami felmerülő nyelvi eszköz, az a konstruktorok lehetőségei, hogy milyen paraméterekkel lehet létrehozni dátumot. Itt két lehetőségünk van dátumnál, hogy 0, 1, 2, vagy 3 integerrel hozunk létre. A másik, hogy egy string változóból hozzuk be a dátumot. Nézzük az előbbire, hogy ezt hogy lehet meghívni?

date a(2017,4,19)

Azonban van lehetőség default érték megadására. Ha a három számból pl. egyet elhagyunk, hogy *date b(2017,5)*, akkor jobbról elkezd életbe létni a *default* érték. Ha kettő paramétert adunk be, akkor jelen példában ez május elseje lesz, tehát mindenki boldog, hogy munkaszünet van. :) Ez nemcsak konstruktorokra érvényes, hanem bármilyen függvényre is. Illetőlegesen lehetőség van arra is, hogy egy számot adunk meg *date c(2017),* akkor idei év január 1.-je van. Innentől kezdve ezzel a konstruktorral elhagyhatunk 0, 1, 2, vagy akár 3 paramétert is. Van a másik konstruktorunk is, amely szöveges reprezentációval dolgozik. Ha ilyenünk van, hogy *date s("2017.4.16");*. Itt látszódik, és alapvető különbség a JAVA és C++ között, hogy vannak műveletek, amelyeket kifejtünk, és van, amit csak deklarálunk egy fordítási egységben.

Vajon mi alapján van, hogy egyes műveleteket a headerben megírunk, és a fordítási egységében kifejtünk? Mi lehet a különbség, hogy egyes dolgokat kifejtünk a header-ben, másokat pedig nem. Gondoljuk el, hogy valójában máshogy kezeljük a header és a .cpp fájlokat. A header fájlokat pl #include-oljuk. Itt tehát bekopipésztel bizonyos pontokra. De a .cpp fordítási egység lefordul, linkeléskor megkeresik a különböző pontokat. Tehát a headert bemásoljuk. Ha tehát ott egy függvénydefiníció / -törzs van, az bemásolódik. Ha ez többször történik meg, akkor sérül az "egyszer beillesztés" törvénye. Miután ezt lekopipészteli, szükséges, hogy ez valahogy kezelve legyen. Az a függvény, amelyet bemásolunk, automatikusan *inline függvény*nek minősül. Ez azt jelenti, hogy a compiler nem akarja alacsony szintű objektumnak tekinteni, ahol meghívjuk, oda szeretnénk beilleszteni. Ha egy inline függvényt szeretnénk megosztani, hogy nem lesz belőle alacsony szintű

kód, azt preprocesszor segítségével fogjuk megtenni. A példakódban látszódik, hogy azok a műveletek, amelyek *inline* ajánlással megjelennek, nem akarjuk, hogy object szintű alacsony kód legyen belőle, mert akárhova beillesztjük, az lenne belőle. Ezt akarjuk elkerülni. Tehát akárhány függvényhíváshoz helyettesítse be a törzsét a függvénynek. Itt is látjuk, hogy header-ben a függvény deklarációja, vagy inline függvény definíciója van!

Ugyanez a logika jelenik meg fent is. Azokat a függvényeket szeretnénk *inline*-osítani, amelyek egyszerű. Miért nem inline mindegyik? Azért, mert a bonyolultabb törzseket több helyre behelyettesíti, és az azt jelenti, hogy elkezd duzzadni a kód mérete, és lassabb lesz a fordítás, futtatás. Ha nagyon a függvények, nem illik *inline*-osítani, sőt, a fordító sem fogja engedélyezni. Tehát itt is a méret a lényeg.

Jönnek a többi tagfüggvények. Nézzük meg, ezeket hogyan lehet meghívni. Ha mondjuk vannak dátumaink és tagfüggvényeink (amelyeket mindig az objektumon hívjuk meg), azaz

std::cout<<c.get_day();

akkor a *c*-n keresztül hívjuk meg (az egyik fenti példán át).

Nézzük, van az az *s* objektum. Ha például *s.set_month(4);*, akkor meghívjuk az egyik módosító függvényt az objektumon. Mit látunk? Azt látjuk, és azt szeretnénk, hogy abban a műveletben, azon az állapoton, történjen meg egy állapotváltoztatás, amelyet ha lekérdezek (*s.get_month();*), akkor ezt a megváltoztatott állapotot írja ki. Ahhoz, hogy ezen a ponton a *month*-ra hivatkozunk, azt akarjuk elérni, hogy az már nem az a month lesz, amit már leírtunk. Ha azt akarjuk, hogy a hónapunk legyen egyenlő a paraméterünkkel (ami ez esetben a 4-es), akkor mi lesz? Akkor annak az objektumnak lesz a *month* tagja, amelyet meghívtunk. Azt szeretnénk, hogy ezen a ponton az objektummal tényleg történjen valami. Különböző objektumorientált nyelvek ezeket a *this*-szel jellemzi. Ha egy osztályon belül vagyunk, azt a dátumot mindig megkapjuk, amelyen meghívjuk a tagfüggvényt.

Nézzük meg, hogy néz ki ez a *this*? Nyilván valahogy el kell érni azt az objektumot. Ha ezen a *set_month()*-t végrehajtom, le is kell tudni kérdezni. Ha azt mondom, hogy *s.set_month(4);* akkor olyan függvényhívás történik a C++ kapcsán, hogy valójában olyan formátumban történik, hogy *set_month(&s,4);*, ez a *this* kulcsszóval memóriacímet tudok elérni. Ezen a pointeren keresztül tudom változtatni ezt az *s*-t, tudom elérni és módosítani is. Annyiban másabb ez a *month* azonosító ezen kontextusban, hogy minden dátumnak van hónap része. Azonban ahol hivatkozom rá, hogy ez annak a dátumnak a *month* része, amelyen meghívtuk. Valójában nem ugyanaz a kontextust jelenti mindkettő.

Ezen ponton mi lehet ezen *this* típusa? Ezek szerint ez egy dátumra fog mutatni, tehát egy pointer, mégpedig egy olyan pointer, amelyet nem akarunk, hogy megváltozzon. Tehát végig a tagfüggvény hívás alatt végig arra kell mutatnia. Ne lehessen nullpointerre állítani, ne lehessen egyáltalán másra mutattatni, csak arra, amelyikre meghívtuk a függvényt. Ezek szerint az az objektumorientált logika is, hogy a tagfüggvényeken keresztül állapotmódosításokat hajtok végre. Megváltozik az objektum, amelyet meghívtam.

Vannak olyan dátumok azonban, amelyek ritkán / sose változnak. Ilyenek például a születésnapok. Arról nyilván nem szeretnénk, hogy megváltozzon. Ha van egy konstans születésnapom, akkor szeretnénk, hogy a szülinapom ne változzon. Azonban az objektumorientáltság azt jelenti, hogy ezt meg lehet változtatni. Innentől kezdve az a logika, hogy ha szeretném, hogy ne változtassa meg az állapotát, hogy ha szerepel a *const* kulcsszó, akkor nem változtatja meg az objektum állapotát. Például *std::cout<<bday.get_day();* nem változtatja meg. Azaz ha *bday.set_year(1985)*, akkor dobjon fordítási hiba lesz.

Ha azt mondom, hogy:

```
date s("2017.3.2");
s.get_day();
```

Akkor az *s* címe átadódik *this*-ként. Ez szintén *this* lesz, csak a típusán lesz változás. Itt a típusa *const date* const this* lesz. A legelső *const* miatt a compilernek fordítási hibát kell dobnia, ha meg akarjuk változtatni az objektum állapotát.

Van még a visszatérési értéke a módosító műveleteknek (*set_year(), set_month(), set_day()*). Látható volt a főprogiban egy olyan létrehozás, hogy *date d*. A C++ úgy döntött, hogy default konstruktorral hozzuk létre, akkor is kell a zárójel. Azonban itt a *date d* a d nevű függvény deklarációja, amely paraméter nélküli. A másik dolog, hogy mi a visszatérési értéke? *d.set_year(2016);*, ennek visszatérési érték típusa egy dátumreferencia. Ez valaminek az álneve, mégpedig azé, amit visszaadunk. Ez tehát nem más, mint maga a year. Fontos, hogy referenciával térünk vissza. Ha az nem lenne ott, érték szerint térnék vissza, és másolattal térnénk vissza. Ezen az álnéven meghívhatnánk tehát azt is, hogy *d.set_month(11);* A *d* tehát egy álnév lesz az objektumra. Csak közben beállításra került az év és a hónap része. Akkor mondhatjuk azt is, hogy *set_day(5);*. Így máris láncba lehet kötni a műveleteket, amiatt, hogy referenciával térünk vissza.

Innentől kezdve ezek a műveletek láncba köthetőek.

Nézzük tovább. Van még két művelet. A *next()*-tel a rá következő dátumra állítjuk, illetve az *add(int n)* pedig hozzáad *n* napot. A két *operator++* ezután azért van, hogy tudjuk dátumot növelni, akár prefix, akár postfix módon is.