Programozási nyelvek I. C++.

1. előadás

A jegyzetet *Pataki Norbert* előadásán *Lanka Máté* készítette.
Előadás ideje: 2017.02.15. 12:00-14:00.

Technikai információk:

- e-mail cím: patakino@elte.hu
- 12:15-13:45.
- **számonkérés:**
  - gyakorlatokon +/-, ebből 3 darab lesz, +, 0, illetve – értékelést lehet kapni.
  - a géptermi vizsga két részből áll:
    - elméleti rész
      - ebbe beleszámít a +/- eredménye
      - 15 kérdéses teszt
      - 8 ponttól van meg a beugró
    - gyakorlati rész
      - három órás a vizsga
  - a jegy a két rész átlagából jön ki, amelyet nem egész eredmény esetén a gyakorlati jegy irányába kerekítenek.
  - opcionális beadandó
    - megnézik, mit hogyan használt, jók-e a megoldások, amiket alkalmazott, visszajelzést is kapunk róla
    - jellemzően algoritmikus feladat köré épül
- katalógusos az előadás
- a gyakorlatok táblásak, de a kódokat érdemes otthon kipróbálni
- **Ajánlott irodalmak:**
  - Bjarne Stroustrup: A C++ programozási nyelv
  - Scott Meyers: Hatékony C++
- **Nem ajánlott irodalmak:**
  - Benkő Tiborné: Objektumorientált programozás C++ nyelven
  - Sipos Mariann: Visual C++ és az MFC

Az előadás nemcsak a C++-ról, hanem a programozási nyelvekről is szól. Ezek a nyelvek változnak, javulnak, romlanak. Ez igaz a processzorokra. „Olyan nincs, hogy valami nem sörnyitó.” A processzor csak gépi kódokat tud értelmezni. Magas szintű nyelvi konstrukciókkal kell elérni, hogy a mi kódjainkat is tudja értelmezni a processzor.

A C nyelvből lett a C++, Bjarne Stroustrup kezdte meg 1970-től ezt csinálni, bővíteni a C-t, és kb 1980-tól beszélhetünk C++-ről. Míg JAVA-ban vannak, metódusok adattagok, addig C++-ben, bár ezek megvannak, de máshogy.

Milyen viszonyban áll a C++ és a JAVA? Ezekre is kitérünk majd a későbbi előadások során.

Bizonyos konstrukciók nem voltak benne a C++03-ban, míg a C++11-es szabványban már igen.

Szerencsére a C++ a leghatékonyabb nyelv. „Minden más nyelvhez csak felejteni kell.” A C++ tele van bonyolult konstrukciókkal, de ha ezeket megértette az ember, más nyelveket már könnyebben megért.

Programozási nyelvek története, programozási paradigmák

Ugyebár a processzor (CPU) egy egyszerű, gépi kódot futtat. Számokkal működik. Erre épül egy olyan nyelv, amelyet **Assembly**nek hívnak. Megjelenik ezekhez a számokhoz azonosító is. Ezek direktben egy-egy processzorutasítást jelentenek, a gépi kóddal megegyező absztrakció. Ezt a kettőt például alacsony szintű nyelveknek nevezünk.

Amikor egy feladatot kapunk programozásban, az általában komplexebb. Fel kell bontani részekre. Ehhez kellenek a **paradigmák**. A paradigmák eldöntik, milyen absztrakciók mentén bontsuk fel a feladatot. Ehhez van eszközkészlet is, illetve némi háttérismeret is. Ezek járultak hozzá, hogy programozási paradigmák kezdtek kialakulni. Ebből jött létre az ún. imperatív programozás.

Ezek után először a **procedurális programozás** jelent meg, amikor már nyelvi vezérlési szerkezetek, változók, alprogramok fogalma, paraméterezhetőség jelennek meg. Ebbe az irányba (is) indult el a fejlődés. Tipikusan ilyesmi a C nyelv.

1967-ben jelent meg az akkoriban nehezen megérthető nyelvnek számító **Simula 67**. Sajátossága, hogy ez volt az első objektumorientált programozási nyelv. Akkoriban nehezen értették, nem teljesen tudták a programozók, hogy mit is akar ez a nyelv. Az elképzelés az volt, hogy objektumokból épül fel minden, vannak állapotai, tudnak üzenetet küldeni objektumok egymásnak (pl.: metódushívás), ez lett az egész alapja.

Még szintén az 1960-as években jelent meg egy másik szemléletmód, a **LISP**. Ez egy elég borzalmas szintaxissal rendelkező, a funkcionális programozás első ilyen nyelv. Valójában a program futása egy függvény kiszámítása, és minden dolog egy állapotmentes számítást próbál leírni. A gond az, hogy a programok, amiket használunk, tele vannak állapottal. Azóta vannak már ennél jobb nyelvek is, pl.: **Haskell**, **Scala** (utóbbi elég beteg nyelv).

1960-ban jelent meg a **SQL** deklaratív programozás. Ebből egy példa:

```
SELECT *
FROM USERS
WHERE userid=”xy”;
```

A **HTML** szintén deklaratív. <html> <head> blabla </head> <body> blabla </body> </html>.

Szintén szélsőségesen deklaratív nyelv a **Prolog**. Ez logikai programozás, az 1970-es években jelent meg. A program logikai tényekből és eldöntendő állításokból épül fel. A program futása alatt el akarja dönteni, hogy az állítás következménye-e a tényeknek.

A C++ egy multiparadigmás nyelv. A C++, amikor Straustrup megalkotta, azt mondta, hogy minden leforduló C kód egyben egy leforduló C++ kód is, és ugyanúgy működjön is. Ma már nincs így. A C++ közel 50 éves nyelv, amelyet Straustrup szeretett volna objektumorientáltsággal kiegészíteni. A C++ sokáig bővült, az első szabvány pedig **1998**-ban jelent meg (C++98). Ebben megjelentek pl.: a sablonok (*template*-ek). Ezen template-ek szintjén jelent meg a generikus programozás. A C++STL valamikor 1994-ben lett implementálva a HP által. Ez egy olyan könyvtár, melyben vannak adatszerkezetek és algoritmusok. Ez azt oldotta meg, hogy egy-egy algoritmus különböző adatszerkezetekkel is képes működni. Tudunk új algoritmusokat is hozzávenni, amíg a megfelelő adatszerkezetekkel működik, illetve tudunk új adatszerkezeteket hozzáadni, ha a meglévő algoritmusokkal továbbra is működik. Ez a generikus programozást is elkezdte támogatni.

Ezek után eljutottunk oda, amit úgy hívnak, hogy **generatív programozás**. Ez nagyjából arról szól, hogy a kódok egy eszköz által kerülnek feldolgozásra. Azaz kód generálódik eszköz által. Szintén elterjedt az **aspektus-orientált programozás**. Idén már 50 éves az objektumorientált programozás, ezalatt nemcsak kiismertük a trükkjeit, hanem a gyengéit is. Utóbbira válasz a generikus programozás. Az aspektus viszont úgy alakult ki, hogy megvizsgáltak egy objektumelven működő kódot és arra jöttek rá, hogy vannak olyan kódrészletek, amelyek logikátlanságuk ellenére be kellett,

hogy kerüljenek. Ha a kód végrehajtása meghatározza, hogy ennek itt is, ott is ott kell lennie. Az aspektus programozás arra jött létre, hogy ha van olyan dolog, kódrészlet, ami felesleges, ezeket szervezzük ki egy aspektusba és szőjük bele a kódba oda, ahol tényleg végre is kell hajtani.

Még egy másik típusú megoldás a C. Simonyi-féle Intentional programming.