Programozási nyelvek I. C++.

3. előadás

A jegyzetet *Pataki Norbert* előadásán *Lanka Máté* készítette.
Előadás ideje: 2017.03.01. 12:00-14:00.

Emlékeztető:

A programkód fordítása három részből áll: preprocesszálás, nyelvi fordítás, és összeszerkesztés. Az elsőről már volt szó, míg utóbbival nem foglalkozunk, így ezeket kipipálhatjuk. A mai előadáson a nyelvi fordításról lesz szó.

A compiler (fordítás) alatt erről van szó. A compiler különböző tokeneket csinál, amelyeket ismer. Ezekre nézzünk egy programkódot példának:

```
int main()
{
	std::cout>”Hellor World!”<<std::endl;
	return 0;
}
```

Automatikusan a *main()* esetében ha nem írunk return-t, akkor is *return 0*-t fog visszadobni, nem kötelező odaírni. De mondjuk Linuxokon a ? környezeti változóba kerül be a return értéke, hogy pl. sikeres volt-e a futás, vagy hibakódot kell-e visszaadni.

Viszont nézzük a tokeneket, milyeneket lehet ezekben felismerni? Vannak benne pl. **kulcsszavak**. Ilyenek pl. *return*, *int*. Vannak **azonosító**k is, lásd *std*, *cout*. Viszont az *std::cout* nem egybefüggő azonosító. Nyelvi token szinten ezek két külön azonosítónak minősülnek. **Konstans**ok is léteznek, ez a *return 0*-ban lévő nulla is. Ehhez hasonló fogalom a **konstans szövegliterál** is, amelyet nyelvi szempontok alapján meg kell tudni különböztetni. Ilyen literál pl. a *„Hello World!”* is. További tokenek még az **operátor**ok is, ebben a példában a *::* konstrukció is, ami az *std* és a *cout* van, ez itt operátornak minősül, de például a *<<* is az. Még egy kategória még a **szeparátor** is, ezekhez tartoznak például a kapcsoszárójelek, pontosvesszők.

Maguk ezek a tokenek annyira a nyelv egységei, olyan alapvető elemei, amelyek tovább már nem bonthatók. Vannak olyan nyelvi részletek, amelyeket ezekből felépítve komplexebb egységet lehet kapni. Például az x = y + 3 egy kifejezés, de **nem** token. A tokeneket tetszőleges szóköz (whitespace) határolhat el, a lényeg, hogy következetesen használjuk ezeket, a kapcsos zárójeleket, tabulátorokat. A konvenció nem számít (azaz pl. JAVA-san írjuk meg), a lényeg a következetesség, illetve nézzen ki jól. Viszont vannak tokenek, amelyekben nem lehet határolókarakter (ld. *co ut* hibás).

Kezdjük el ezeket átnézni. Vannak tehát **kulcsszavak**. Önmagában egy kulcsszó jelentése nem változhat meg, azaz nem tehetünk rá más fogalmat, mint amit jelent. Pl. a *const* az különböző pontokon más-más jelentésű, de ezeken kívül más jelentést nem tehetünk rá. Hasonló ehhez még a *static* is, vagy az *int*. Vannak még a vezérlési szerkezetek, lásd *if, while,* stb.

Ennél izgalmasabbak az **azonosító**k témaköre. Hogy nézhet ki egy szabványos azonosító C++-ban? Mi az, amit feltehetünk róla, vagy nem? Egy azonosítót úgy lehet leírni nagyjából, hogy betűvel kezdődik (akár aláhúzás is (_) (underscore) (igen, ez is egy betű)). Nem kötelező, hogy 1-nél több betű legyen benne, viszont az első betű után folytatódhat betűvel vagy számmal is. Mivel a jelentésük nem változhat meg, ezért kulcsszavak **kizárva**. A *static* például lehet azonosító, de mivel az már kulcsszó, ezért így nem nevezhetünk el azonosítót.

Nézzünk néhány legális azonosítót. *__xy*, *abc666*, *ABC666*. Mivel a kisbetű és nagybetű meg van különböztető, ezért abc666 nem ugyanaz, mint ABC666 (azaz case-sensitive). Így viszont a *ConsT* is valid (más kérdés, hogy mennyire logikus).

Ellenben nem legális a 666abc (számmal nem kezdődhet).

Hogy minek milyen nevet adunk, az mindegy, viszont érdemes ezeket logikusan elnevezni. Például ha egy azonosító több névből áll, akkor a szavak határát _ jellel jelölik. Például a vektornál, ha hozzá akarsz adni, azt a *push_back* paranccsal lehet megtenni. Ellenben vannak olyan névkonvenciók, amelyek leginkább JAVA-ban jelentek meg, ilyen a *CamelCase*. Ez azt jelenti, hogy ha több szóból áll a név, akkor az új szó nagybetűvel kezdődik (PlValahogyIgy).

A névkonvenció tehát arról szó, hogy próbáljunk meg logikát vinni az azonosítókba. Legrégebbi ilyen megoldás az ún. *Hungarian notation*, amelyet Charles Simonyi talált ki. Elkezdtek olyan információkat beletenni az azonosítóba, hogy az pl. milyen típusú. Az i-vel kezdődő pl. utal arra, hogy az egy *integer*. Később bele kellett tenni további információkat is az azonosítóba, pl. *lpzstrMsg*. Ez annyit ír le, hogy egy 32 bites pointer, ami egy \0-ra végződő string legelső karakterére mutat. Jól jött, hogy egy-egy ilyen azonosítóba belecsempésztek plusz információkat. Nyilván ez a *Hungarian notation* elkezdett elterjedni, de utána belerakták azt is, hogy ez pl. egy osztályon belüli statikus, nem statikus, vagy globális adattag. Ilyen logikával rengeteg információt lehet beletenni az azonosítóba, de ha elveszik a lényege az egésznek, akkor nem teljesen szerencsés, hogy ki kell bogozni, hogy ez mit is csinál. Ez 20 éve még hasznos volt, de manapság mellőzve van.

Bizonyos esetekben, ha több szóból áll az azonosító, akkor jön a korábban említett *CamelCase*. A C++ a JAVA-val szemben ezt nem szokta kikényszeríteni. Szabványkönyvtár szinten az _ jelet szoktuk használni. A *NULL* viszont preprocesszor szimbólumnak minősül, érdemes mellőzni. Érdemes tanulás közben kialakítani egy saját névkonvenciót, amelyet könnyen meg lehet szokni, legyen következetes, de nem kötelező.

Van néhány problémás eset is azonban C++-ban, ez az aláhúzással kezdődő azonosító. Nagyon sokan szeretnek osztályon belüli adattagot aláhúzással kezdeni. Aláhúzással jelzik, hogy ott osztályszintű adattagról van szó, és meg lehet különböztetni a konstruktor paramétertől.

```
class complex
{
        double _re, _im;
public:
        complex(double re, double im)
        {
                _re = re;
                _im = im;
        }
}
```

Szabványkönyvtárban vannak olyan adatszerkezetek, amelyek rendezettek. Ezekből 4 darab van, neveik pedig: *set, multiset, map, multimap*. Elmondja a szabvány, hogy a keresés logaritmikus műveletigénnyel kell, hogy rendelkezzen. Miért? Mert rendezettek. Adódik, hogy ha rendezett, ez az optimális, de azt nem mondja ki, hogy ha pl. a *set*-nél (amely egy halmaz) milyen adatszerkezet található meg ténylegesen (keresőfa, kiegyensúlyozott keresőfa, hogy van az kiegyensúlyozva, vagy rendezett lista jellegű adatszerkezet). Ezekről nem szól a szabványkönyvtár, csak annyiról, hogy ez egy rendezett adatszerkezet. Gyakori, hogy a szabványkönyvtár implementál egy olyan adatszerkezetet, amely garantálja ezt a rendezettséget. Jellemző pl. a piros-fekete fa. Fordítási hibaüzenetben, ha ezeket az adatszerkezeteket használjuk, olyan típusokra ad hibaüzenetet a

program, hogy __rb-tree….Vagy_Rb_tree… Az a helyzet, hogy erről a szabvány nem tud nyilatkozni, hogy milyen azonosítók állhatnak az adatszerkezetek között. Ezeket használják fel az adatszerkezetek implementáláshoz. Vannak olyan azonosítók azonban, amelyeket a szabványkönyvtár nem készít fel jól, mert nem tudja, hogy pl. ilyenek lesznek. A programozó felelőssége, hogy az ilyen azonosítókkal ne legyen probléma.

Ezért az a javaslat, hogy ne használjunk aláhúzással kezdődő azonosítót. Miért? ~~*Nem szexi.*~~ Nem néz ki jól, és ne ütközzön, ennyit nem ér.

A folytatásban, sorrendet cserélve, a **konstans szövegliterál**okról lesz szó, majd később a konstansokról. Igaz mindkettőre, hogy van típusuk és értékük is. Adódik a kérdés, hogy mi a „Hello”-nak a típusa. A konstans már jól hangzik, de a valódi típus *const char[6]* (C++-ban). Amikor a C-t kialakították, ott még nem volt a *const*. Valójában ilyenkor a C és a C++ úgy kezeli a stringeket, hogy ezek egybefüggő tárterületen vannak, egy olyan tömbben, amelynek van 6 eleme.

'H'	'e'	'l'	'l'	'o'	'\0'

C-ben ugyebár nem volt *const*, azaz csak *char[6]* volt, ellenben C++-ban már *const char[6]*. Amikor ezt a C++ bevezette, a compiler-nek van joga arra, hogy ezt olyan tárterületre pakolja be, ahol az operációs rendszer levédi, hogy ezt ne lehessen módosítani.

C programozók azt csinálták, hogy a lokális változók között nem egy tömböt vettek fel, hanem azt mondták, hogy ha van a lokális változók között olyan változó, amely futás közben memóriacímet foglal le, kezdetben rámutat a string legelső elemére. Ekkor már csak memóriacímet kell foglalni, és tudjuk, hol ér véget az egész: ameddig el nem érjük a legelső '\0'-t. Nem kell tehát tömböt kezelni. Miután ezt így elkezdték használni, jött a C++, és úgy döntött, hogy:

char p = „Szia”;*

Ez hibát fog visszadobni. Önmagában ez még nem gond, hogy engedi lefordítani, viszont a mutatott értéket meg lehet változtatni. Például az azt írom, hogy *p[1] = 't'*, az teljesen valid. A p-t kirakta a C++ konstansoknak fenntartott tárterületre. Ha viszont ezt a fenti parancsot megpróbáljuk lefordítani, az lefordul, *warning*-ot se dob vissza, viszont futási időben az operációs rendszer leállítja a programot: olyan tárterületre próbált írni, amelyhez már nincs jogosultsága.

```
void f()
{
	char* p = „Hello”;
	int len = strlen(p);
}

int strlen(char* str)
{
…
}
```

Ez még C-s megoldás volt, ahol addig tartott a szöveg, amíg nem ért el a '\0'-ig, azonban a C++ átvette, csak annyiban lett tudatosabb, hogy ezt olyan tárterületre kell rakni, ahol a konstansok vannak és konvertálódik egy első elemre mutató pointerré! Ha ezt megpróbáljuk megváltoztatni, akkor futási idejű problémát fog visszadobni.

A C-vel nagyjából egyidős Pascal pl. olyan stringábrázolást használ, ahol a tömb legelső bájtján van a szöveg hossza. Ezt azt jelenti, hogy maximum 255 karakterből állhatott egy string, mert csak annyi fért el egy bájton. Ehhez képest sokkal jobb és elterjedtebb a C / C++ stringábrázolását.

Érdemes még megjegyezni a JAVA-ét is. Tisztább konstrukciója ennek, hogy nem kellett ragaszkodni egy 50 éves konstrukcióhoz, ehhez képest a JAVA stringkezelése sokkal tisztább,

úgynevezett *immutable*. Ha van egy String objektumom, akkor annak karakterlánca nem tud megváltozni. Ha hozzá akarok fűzni egy új szöveget, vagy változtatni akarok rajta, akkor egy teljesen új objektumot hoz létre, ezt jelenti az *immutable* fogalma. Sehogyan se lehet megváltozni az állapotát. Bevezetésre került a StringBilder és a StringBuffer. Az egyik garantálja a szálbiztonságot, a másik viszont nem. Mindig változik, hogy éppen mit reprezentál, ebből kapjuk meg azt az objektumot, amit szeretnénk. Ez a kettő például már *mutable*.

Térjünk rá a **konstans**okra. Miért van külön kezelve ez, és a szövegliterál. Itt is van típus és érték is. A konstansoknak nem kell területet foglalni, nincsenek feltétlenül benne a memóriában. A *8*-at, ha vesszük, itt az értéke 8-as, típusa pedig *int*. Felmerül a kérdés, hogy mi is az, hogy típusrendszer, mi az, hogy int? Megint olyan absztrakcióban vagyunk, ami a számítógépnek nem triviális? Számítógépszinten nem beszélünk olyanokról, hogy típusok, csak bitsorozatokról tudunk. Az *int*, mint minden más a számítógépnek, tehát egy bitsorozat. A sorozatról elárulja annyit a C++, hogy az int lehet negatív is, tehát ez egy *előjeles egész* típus. Azt már nem definiálja, hogy ez hány bitből áll. Azzal a logikával él a C++, hogy ez ne 16, 32, 64 bitből álljon, hanem akkora legyen, hogy ez jól kifejezhető, kezelhető legyen a processzor műveleteivel. Ezt úgy kell érteni, hogy legyen *optimális* a processzor szempontjából.

Van egy olyan operátor a C++-ban, amely megadja, hogy hány bájtos, ez a *sizeof(int)*. Ez viszont változhat. Az a koncepció, hogy a processzort minél optimálisabban tudjuk kihasználni. Ha összeadok két intet, akkor azt egy lépéssel meg lehessen oldani.

$sizeof(short) \leq sizeof(int) \leq sizeof(long)$

A mai 64 bites rendszereken a long el tud térni. Linuxon és Windowson például teljesen más méretekkel rendelkezik.

Miért érdekes ez? ~~Semmiért, de valamiről muszáj pofázni.~~ Régen voltak 16 bites processzorok, azaz 16 regiszter voltak benne. Ha veszünk egy ilyet, és veszünk egy előjeles intet, akkor a legnagyobb ábrázolható int, hogy 1 db 0 (előjelnek), és 15 db 1es, ez kettes számrendszerből átalakítva 32767. Mi a gond ezzel? Ha van egy integer változónk, amelynek az értékét nem tudjuk, és le akarom írni azt, hogy 20000+x, akkor előfordulhat, ha az x is 20000, akkor túlcsordulás fordul elő, azaz nem 40000 lesz az eredménye, hanem valami negatív szám. Ilyenkor tudjuk jelezni, hogy lehessen meghatározni egy olyan konstanst, hogy az *ne int legyen*. Alapesetben, ha egy konstanshoz akarok hozzáadni, az legyen egy int. Ezért suffixeket vezetett be a C++, azaz ha azt írom, hogy 20000L, akkor az nem int, hanem *long* típusú konstans lesz.