Programozási nyelvek I. C++.

4. előadás

A jegyzetet *Pataki Norbert* előadásán *Lanka Máté* készítette.
Előadás ideje: 2017.03.08. 12:00-14:00.

Emlékeztető:

Tokenekről volt szó előző előadáson, ahol a kódban vannak tokenek. Vannak **kulcsszavak**, **azonosítók**, elkezdtük megbeszélni a **konstans**okat is, de beszéltünk **konstans szövegliterálok**ról is, előbbieket még nem fejeztük be. Lesz még szó **operátorok**ról, illetve nem is ezek lesznek érdekesek, hanem a kifejezések kiértékelése. Maradtak még a **szeparátorok**, amelyekről pár szót fogunk csak ejteni, de sok tartalmat nem fognak jelenteni.

Ha valahova leírunk valamit (pl. 15), annak van típusa és értéke is. Van egy olyan egész számábrázolás, amelyet *int*-nek nevezünk, ez a típusa. Egy C++ nem akarja megmondani, hogy hány biten / bájton ábrázolja, hanem van processzor architektúránk (32, 64 bites), és abban vannak regiszterek. Nyilván lehetnek olyan architektúrák, hogy lehetnek olyan regiszterek, amelyek nagyon picik, ezeket bonyolítani kell, de lehet olyan is, ha igen nagy számot akarunk ábrázolni. Lényeg, hogy ilyen ábrázolás mellett az összes aritmetikai ábrázolás bonyolultabb lesz.

De a C++ azt mondta, hogy ha vesszük az alapértelmezett egész szám típust, akkor nem mondjuk meg, hogy hány biten legyen, hanem ami a legoptimálisabb. A *sizeof(int)* erre azt mondja, hogy ez implementációfüggő. Lekérdezhető, de a compiler joga, hogy eldöntse, ez mekkora legyen. Vannak olyan pontok a szabványnak, hogy ezeket a compiler döntse el. Az *int* egy garantáltan előjeles típus.

Amellett viszont a C++ azt mondta, hogy ha valakinek erre az előjelre nincs szüksége, akkor lehessen olyan típust létrehozni, ahol azt az előjelbitet helyiérték-bitként lehessen kezelni. Erre mondja még azt a C++, hogy:

$$\text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)}.$$

Létezik egy *unsigned int*, ez az előjel nélküli int, azaz az előjelet lefoglaló biten is számot tudunk tárolni. Azt mondja még ki a C++ szabványa, hogy ennek ugyanolyan méreten kell lennie, mint az *int*-nek:

$$\text{sizeof(T)} == \text{sizeof(unsigned T)} == \text{sizeof(signed T)}.$$

Van a típusrendszerben egy egység, amelyet a *char* ad meg. Ez a típusrendszer **egysége** lesz. Minden típus, amelyet ki tudunk fejezni, az méretileg a *char* többszöröse lesz:

$$\text{sizeof(char)} \equiv 1.$$

Abban az értelemben, hogy ennél nem lehet kisebb, illetve nincs olyan típus, ami nem ennek a többszöröse. Bár előfordul, hogy 8 bites *char* van, de ez is főleg implementáció-függő.

Előző előadáson volt szó, hogy konstansoknál tudunk beletenni olyan szuffixeket, amelyekkel tudjuk módosítani a típusát. Pl. ha 15-öt írunk, az *int*. Ellenben ha azt írjuk, hogy 6L, az egy *long*. De ha azt írjuk, hogy 9U, akkor az már *unsigned int* lesz. Ilyen szuffixekkel tudunk eltérni az alapértelmezett típustól is. Lehet ezeket kombinálni is, a 12UL *unsigned long* lesz. A *short*nak viszont nincs megfelelő szuffixe.

Sokszor van még olyan, hogy nem csak tízes számrendszerre van szüksége. Az előbb említett példáinknak pl. az értéke ugyanannyi. Viszont programozóként belefuthatunk olyanba, ha nem tízes számrendszerre van szükségünk. A 0x24-ben a *0x* azt jelenti, hogy ez a szám hexadecimális, azaz 16-os számrendszerben van leírni. Innentől kezdve a 0x24 az 36-os értéknek felel meg.

Értelemszerűen $2 * 16 + 4$. Itt már lehet használni az A-tól F-ig terjedő „számokat” is. Hasonlóképpen a korábbi a szuffixekhez, itt is tudunk olyat mondani, hogy 0x15L, azaz a 21-es érték *long* típusú konstans. De nemcsak 16-os számrendszerben lehet leírni a számjegyeket, hanem le lehet írni 8-as (oktális) számokat is, az egy 0-s prefixel kezdődik. Például a 023 értéke 19 lesz ($2 * 8 + 3$). Sokszor jobban ki lehet fejezni egy feladat megoldását, ha nem csak decimálisan, hanem pl. oktálisan számolunk.

Vannak lebegőpontos konstansok is. Hasonlóképpen érvényes az, hogy van egy ilyen számábrázolás. Ha azt mondjuk, hogy 1.5, akkor ez egy lebegőpontos konstans, típusa *double*.

$(-1)^0 * 15 * 10^{-1}$. Az ilyen adatokkal le lehet írni bizonyos értékeket. Ilyen megoldásokkal általában el vannak fogadva a lebegőpontos számábrázolás. Ennél általában jobb megoldásokat szoktak alkalmazni, és sokszor trükkösebben.

Az ilyen konstanst engedi is leírni a C++, csak a következőképpen: $15e$-1, értéke szintén 1.5.

Hasonlóképpen érvényes az, hogy a *sizeof(double)* szintén implementáció-függő.

Van egy olyan típus, amely az alapértelmezett dupla pontosságnál kevésbé precíz, a *sizeof(float)*, ehhez képest a *sizeof(double)* legyen optimális.

$$\text{sizeof(float)} \leq \text{sizeof(double)} \leq \text{sizeof(long double)}$$

Szerencsére ezekre is van szuffix. A 3.25f például egy *float* típusú konstans. Illetőlegesen érvényes itt is az, hogy $3.27e$-2L, az L itt is *long*-ra utal, de a típus szerint ez *long double* konstans lesz.

Van még egy-két szabály, annyira nem megdöbbentők. Pl. a

$$\text{sizeof(char)} \leq \text{sizeof(short)}$$
$$\text{sizeof(char)} \leq \text{sizeof(bool)}$$

A *bool* nincs igazából konstans, főleg mert a TRUE és a FALSE nem igazán konstansként jelennek meg, hanem mint logikai értékek.

Van olyan, hogy *signed T*. Miért? Míg az *int*-ben van előjel, az *unsigned*-ban nincs. De akkor miért kell *signed*? Érdekesség, hogy a *char* típus előjelessége nem definiált (legalábbis implementáció-függő), azaz amikor számmá konvertálódik egy *char*, akkor ha nem tudjuk eldönteni az előjelességét, ezzel lehet megadni, hogy legyen.

A konstansokat ezzel nagyjából meg is beszéltük.

Az **operátorok** fogalomköre következik. Ami talán egy fokkal fontosabb, azok a **kifejezések**. Előző előadáson volt szó, hogy nem is előbbiek hangsúlyosak (tokenként bár ez jelenik meg), de különböző kifejezéseket írunk le igazából. A C bevezette azt, hogy az $=$ operátor értékadást határoz meg, a $==$ egyenlőség-operátor, de továbbiak még a *++, //, &&*. Egy adott kontextusban, ha leírok egy operátor, nem biztos, hogy az teljesen egyértelmű. A *&* például címképzést, vagy akár „bitenkénti és”-t is. A kifejezés fogja ezt adott esetben kifejezni, azzal együtt, hogy mik az operandusai egy operátornak a kifejezésben.

Kifejezések pl.: a+b. Ha a és b egy-egy *int*, akkor ez egy kifejezés. De akár az *x/2*, vagy *s=6*. Ezek nyilván bonyolíthatóak, és komplexebbeket is össze lehet rakni: *2*b+3*c*. Talán már volt említve, jellemzően, ha a C++, amikor elemzik a programunkat, egy fa adatszerkezetbe pakolják össze a kódot, ezt *absztrakt szintaxisfá*nak nevezzük, röviden AST. Hogy néz ez ki? Ha leírunk egy ilyen kifejezést, hogy *a=4*b+9*c*, akkor ebből a fordítóprogram ebből képes egy fát felépíteni.

Koncepciója, ebben van egy értékadás, amely az *a*-nak ad értéket, és van benne egy összeadás. Az összeadás két szorzást ad össze, ahol mindkettőnek van egy-egy operandusa. Bal oldali

szorzásnak pl. 4-es és b, míg a jobb oldalinak 9 és c. Ez azért absztrakt, mert vannak olyan pontok, amelyek elvesznek a fordítás során, vagy zárójelezéssel a részfák máshova kerülnek. Ez azért van, mert az operátoroknak vannak precedenciájuk. Van körülbelül 30-40 operátor C++-ban, és el kell dönteni, melyik értékelődik ki leghamarabb. A korábbi példánál pl. azt akarjuk, hogy előbb a szorzások, majd az összeadás legyen kiértékelve. A második szorzást például hamarabb ki kellene értékelni, mint az összeadást. A Straustrup-könyvben van egy jó precedenciatáblázat, de interneten is megtalálható.

Ha azt akarjuk, hogy az összeadás előbb legyen kiértékelve, akkor át kell alakítani a következő módon: *a=4*(b+9)*c*. Ha viszont mindenképpen szeretnénk, hogy a szorzást értékelje ki előbb, akkor be lehet zárójelezni úgy, hogy *a=(4*b)+(9*c)*, a plusz zárójelek nem zavarnak senkit, nem baj, ha több a zárójel, nem zavar az senkit, nem jár büntetéssel, sőt ebből egyértelműbb is lesz a kifejezés.

De hogy is vannak ezek a precedenciák? Melyik a legmagasabb precedencia? A :: pl, de ezek után jönnek még a metódushíváshoz szükséges . vagy a pointerekhez szükséges →. Érdekes, hogy mi a legalacsonyabb? Az értékadó jellegű operátorok. Az is egy értelmes kifejezés, hogy *a=4*. Miután az értékadással kifejezést írok le, annak van egy eredménye. Mondhatjuk azt, hogy *(a=4)*b*. Ennek a kifejezésnek van eredménye, ahol az *a* értéke 4 lesz, amit utána összeszorzok b-vel, amellyel nem kezdek már semmit, viszont az *a* értékét megszorzom *b*-vel. A másik dolog pedig, ami megjelenik, az hogy van ennek egy mellékhatása (side effect). Nemcsak azt csináltam, hogy megadtam az a értékét, hanem hogy az *a* fel is vette ezt az értéket.

Talán már volt arról szó, hogy ha van egy *int x = 3*, akkor az x értékét meg tudom változtatni a ++ operátorral. Ha kiiratom, hogy *std::cout<<x++*, akkor a kiírt eredmény a meg nem változtatott érték. Viszont az a mellékhatás, hogy x eredménye közben 4-re változott. Ha azt mondjuk viszont, hogy kiíratjuk: *std::cout<<x+y*, akkor nincs mellékhatás, kiszámolja a két értéket, és azt írja ki. Az előtte lévő példánál viszont a kiíratás után *x==4* lesz. Ha viszont *std::cout<<++x*, akkor az lesz, hogy amit kiír, az már 5 lesz, miközben *x==5*. Látszólag van egy prefix operátor, a legtöbb esetben pedig kihasználjuk a mellékhatásokat.

Van még egy dolog, operátorok kapcsán, ez pedig a *kötés*. Ez nem annyira széttaglalt, mint egy precedencia-táblázat, mert vagy balról, vagy jobbról köt egy operátor. De mit is jelent ez? A jobbról kötést könnyebb megérteni. Tegyük fel, azt mondjuk, hogy van három változóm: *int a, b, c*. Ha azt mondom, hogy *a=b=c=6*. Az értékadás tipikusan egy jobbról kötő operátor, mint ahogy itt is. Ez úgy lesz kiértékelve, hogy először a *c*-ben eltároljuk a 6-os értéket, amelynek az lesz az eredménye, hogy ezt eltároljuk *b*-ben, a *b=(c=6)*-nek pedig az lesz az eredménye, hogy az *a* értéke is 6 lesz. Ellentétben mondjuk egy összeadással, ha azt mondom, hogy a=b+c+6, értelemszerűen ezt is jobbról kezdjük kiértékelni. Van az *a=((b+c)+6)*. Ezek a kifejezések úgy működnek, hogy próbáljuk tárolni az eredményeket, majd azokat eldobjuk. Ha azt mondom, hogy *std::cout<<"Hello"*, akkor a mellékhatását használjuk ki, az eredmény ugyanis nem fontos, az eredmény ugyanis az std::cout, ezt ugyanis eldobjuk, ami minket érdekel, hogy kiírjuk a standard outputra, hogy „Hello”.

Van még egy olyan szabály, hogy *szokásos aritmetikai konverzió*. A C++ úgy viselkedik, hogy fordítási időben a compiler nemcsak a konstansokról dönti el a típusát, hanem az operátorokról is. Például ha azt mondom, hogy *std::cout<<13/10*, akkor a compiler rájön, hogy itt van két konstans, mindkettő *int*, közöttük egy operátor, akkor ez egy *int* művelet, és ezt *int*-ként értékeli ki. Ez nem racionális, se nem lebegőpontos számot ad vissza, hanem 1-et. Ha viszont azt csinálom, hogy *std::cout<<13/10.0*, akkor ennek a típusa már *double*, de ebben az esetben a compiler arra jön rá, hogy *int* és *double* között nincs művelet, az *int*-et *double*-lé konvertálja, ez már lebegőpontos művelet lesz, és 1.3-at ír ki. Ha van nekem egy kifejezésem, amiben van egy bináris operátor, akkor bármilyen bináris operátorról is van szó, úgy kell viselkedjen az operátornak,

hogy ha a és b típusa nem egyezik, valami konverziót kell alkalmazni, legalább az egyiket át kell alakítani. Ez egy elég hosszú szabály, de a logikája következő és el lehet sajátítani, hogy miről szól. Elég sok nyelv van, ami az operandus alapján dönti el, hogy milyen is a művelet, mit is hajt végre a processzor.

Visszatérve a szokásos aritmetikai konverzióra, ha két típus eltér, konvertálni kell. Az a logika, hogy a kisebbet konvertáljuk a nagyobbra.

- Ha az egyik típusa *long double*, akkor a másikat is *long double*-ra konvertáljuk.
- Különben ha az egyik operandus *double*, akkor másik is *double* lesz.
- Megint különben, ha az egyik *float*, akkor a másik is *float* lesz.
- De ha nem volt lebegőpontos, akkor életbe lépnek az integrális egészek.
    - Ha az egyik operandus *unsigned long*, akkor a másik is ez lesz.
    - *Itt már nehéz lesz:* Mivel nem tudjuk, hogy a *long* nagyobb-e, mint az *int*, ezért különben ha az egyik operandus *long*, a másik pedig *unsigned int*, akkor előfordulhat az, hogy a két típus ugyanakkora tárterületen van tárolva. Ekkor az utóbbiban nagyobb számokat tudunk tárolni, nem tud akkora számokat leírni a *long*.
      Ezért ha sizeof(int) == sizeof(long), akkor fel kell készülnöm, hogy pl. összeadás eredménye nem fér bele a *long*ba, ezért az a döntés, hogy mindkettő *unsigned long* lesz. Különben ha a *long* nagyobb, azaz sizeof(int) < sizeof(long), akkor *long*gá konvertálódjon mindkettő.
    - Különben ha az egyik *long*, a másik pedig NEM *unsigned int*, akkor *long* lesz a művelet.
    - Különben, ha az egyik operandus *unsigned int*, a másik is konvertálódjon *unsigned int*té, mivel ebben nagyobb számokat tudok tárolni, mint egy *int*-ben.
    - Különben minden *int*-té konvertálódik, ha egyik korábbi sem teljesült.

Az is egy érdekes dolog, ha egy logikai értéket kiértékelünk számként. Ez hogy működik?

Azaz van egy olyanunk, hogy TRUE → 1, FALSE → 0, azaz a 0-ból lesz FALSE, a nem 0-ból pedig TRUE.

Nyilván látszódnia kell ezeknek a szabályoknak, ha 5L + 3 az összeadás, akkor ez egy *long* összeadás lesz, az eredménye pedig *long* lesz.

Vannak olyan pontok a C++-ban, ahol ennél már kevésbé jól szabályozott a nyelv, ugyanis kiértékelések, kifejezések sorrendje is van. Eddig volt precedencia, azonban van egy olyan, hogy futási időben mi fog hamarabb kiderülni? Ezek nem egyértelműek. Nézzünk erre egy példát:

```
int v[5];
int i=0;
v[i++]=i;
```

Itt az *i* kezdeti értéke 0, a *v* első indexe 0 lesz. Azonban az nincs definiálva, hogy melyik kiértékelés történik meg hamarabb. Van viszont egy mellékhatásos kifejezése, és nem tudjuk, hogy az értékadás milyen viszonyban áll az i++-szal. Az viszont definiált, hogy az értékadás történik legutoljára. Ilyenkor *szekvenciapont*ok között tetszőleges sorrend alakulhat ki.

Melyik az a pont, amelyiknél tudom, hogy végrehajtódott a mellékhatás? Csak a *;* szekvenciapont után tudom megmondani garantáltan, hogy *i* értéke már 1. Azelőtt viszont nem tudjuk megmondani, hogy az *i++* mikor értékelődött ki, tehát a v[0] lehet 0, vagy 1 is. Az a legrémisztőbb, hogy az ő gépén ez valahogy működik, arra gondol, hogy ez mindig fog működni, de az a gond, hogy ez implementációfüggő. A fordító itt optimalizálhat, hogy ezt hogyan érdemes megcsinálni, de nem garantálható, hogy az értékadásnál az *i* 0, vagy 1 lesz.

Tehát vannak ezek a szekvenciapontok, ahol ezek a mellékhatások már garantáltan végrehajtódtak. Ilyenek pl: *;, &&, //.* Ha azt mondom, hogy:

```
int i;
if(i++ || i==5)
{
        …
}
```

Itt már lehet garantálni, hogy az *i==5*-nél már a megnövelt értékkel számol. Ha azt mondjuk, hogy növeljük meg az *i*-t, akkor ez azt a feltételt írja le, hogy az i a növelés előtt volt-e kisebb, mint 5. De legtöbb esetben nem tudunk sorrendiséget definiálni. Ha van itt 3 függvényem:

```
bool f()                bool g()                bool h()
{                       {                       {
        std::cout<<'f';         std::cout<<'g';         std::cout<<'h';
        return false;           return true;            return false;
}                       }                       }
```

```
if (f()==g()==h())
{
        std::cout<<'t';
}
else
{
        std::cout<<'f';
}
```

Itt ezeknél nem tudjuk megadni a sorrendjét, meg tudjuk adni az eredményeiket, de ezek nem függnek attól, hogy a függvényeket milyen sorrendben hívom meg, azaz ez garantáltan igaz lesz. Miért? Mert azt mondjuk, hogy az *f()* eredményét hasonlítsuk össze a *g()* eredményével, majd ezt hasonlítsuk össze a *h()*-val. Mivel az első hamis lesz, a hamis == hamis, ezért lesz igaz a függvény. Ezért vannak szekvenciapontok, amelyekre szükségünk van és lehet.

Nyilván a logikai értékeléseknél a C++ lusta kiértékeléssel rendelkezik. A || operátornál csak addig megy tovább, amíg nem tudja egyértelműen eldönteni az igazat. Ha van egy olyanunk, hogy *a* // *b*, akkor *b*-t csak akkor értékeli ki, ha *a* hamis. Ha viszont *a* igaz, akkor nem kezdi el *b*-t kiértékelni. Ez viszont garantálja, hogy az *a* ki lesz értékelve.

A pontosvessző adódik, mint szekvenciapont.

Van még a vessző operátor (,), amelyet érdemes megjegyezni, már csak azért is, mert C++-ban ez egy olyan karakter, mert állhat operátorként és szeparátorként is. Ha paraméterek között áll, akkor szeparátor. De ha azt mondom, hogy *std::cout<<(x,y);*, az azt mondja, hogy értékeljük az x-et, majd az y-t is, és ez itt az y-t írja ki. Érdemes időnként erre odafigyelni, mert a vessző időnként furán tud viselkedni. Még egy példa, ha van egy függvényem: f((1,2)); ez egy egyparaméteres függvény, nem egy kétparaméteres! Kiértékeli mindkettőt, előbb a baloldalit, majd a jobboldalit, az eredmény pedig a jobboldalit. Ez a függvény a *void f(int x)* függvényt hívja meg.