Programozási nyelvek I. C++.

7. előadás

A jegyzetet *Pataki Norbert* előadásán *Lanka Máté* készítette.
Előadás ideje: 2017.04.05. 12:00-14:00.

//Versfelolvasás, Pataki elmenekült. Jól kezdődik ez az előadás.

//Kezdődik a móka ~~és a kacagás.~~

Mai téma, amiről beszélni fogunk, azok az **alprogramok**, és a **paraméter átadás.** Többnyire megint két irányból fogjuk ezeket megközelíteni, és megnézzük a programozási nyelvek általános tudnivalóit, leginkább C-re, és C++-ra. Az alprogramok több különböző jelentéssel is bírnak, több jelentésük is adódott az elmúlt években, évtizedekben.

Amire az ember elsőre gondol, az hogy adott egy kódrészlet. Mi is az alprogram? Megírunk egy kódrészletet, amihez hozzárendelünk egy azonosítót, egy nevet, paraméterekkel is elláthatjuk, függően attól, hogy mennyire van ezeknek létjogosultságuk. Utána meghívással ezeket meg tudjuk hívni őket, akár újra és újra. Általánosságban az alprogram egy lefordítható, leteszthető egységet képvisel. A meghívásnál adjuk meg ezeket a paramétereket, amiket a függvény definíciójánál adunk meg.

Alprogram alatt sok különböző dolgot szoktak megadni. Aki látott anno ADA-t, ott vannak még külön eljárások, és vannak külön függvények is. A kettő nem is ugyanúgy működik, meghívásilag is máshogy működik. A C++-ban ez úgy működik, hogy vannak függvényeink, amelyek megváltoztat(hat)ják a paramétereket. A C++ igazából csak az egyik fogalmat ismeri, a függvényeket. Alapesetben még lehet olyanról is szó, hogy *tagfüggvény*ek, más néven *metódusok*. Vannak virtuális tagfüggvények is, de alapvetően bizonyos fokú különbséggel is rendelkeznek ezek. Vannak még a korrutinok. Az az alapkoncepciója, hogy elindítom az alprogramot, történik valami, de a következő meghívásnál ott folytatódik az alprogram, ahol az előző meghívásnál félbeszakadt. C++-ban ilyen még nincs, de kísérleteznek azzal, hogy belekerüljön.

Nézzünk erre egy példát, hogy lehessen érteni, miről is van szó. Nézzünk egy maximumfüggvényt:

```
int max(int x, int y)
{
        return x<y?y:x;
}
```

Itt *int* a visszatérési érték, *max* a függvény neve, két paramétert vár, és visszaadja a nagyobbik elemet. Ha viszont van egy kételemű tömböm, akkor mondhatom azt, hogy írjuk ki a standard outputra a maximumát a két tömbelemnek.

```
int v[2] = {3, 7};
std::cout<<max(max(v[0],v[1]),2);
```

Itt először kiválasztjuk a tömbelemek közül a maximumot, majd ezt összehasonlítjuk a 2-essel.

Az első példában, amit látunk, az *int y* például egy **formális paraméter**. Míg a másodiknál, a v[0] például **aktuális paraméter**. Nagyjából az a kérdés merül fel a paraméterátadásnál, hogy a formális és az aktuális paraméterek között mik a különbségek? Ahol a programozási nyelveknél az alprogramok más-más válaszokat adtak, így a paraméterátadásnál is más-más dolog szokott történni. Megnézzük ezeket C-re, C++-ra is, de igyekszünk majd ezt nyelvfüggetlen módon is megtenni.

Először nézzük, hogy a programozási nyelvek milyen paraméterátadást találtak ki?

Ami viszonylag elterjedt paraméterátadás, az az érték-szerinti. Fogunk majd még beszélni cím-szerintiről, de lesz még szó eredmény, illetve érték/eredmény-szerint. Lesz még szó név-szerinti paraméterátadásról is. Nagyjából ezek azok az alapkoncepciók, amelyek megjelentek.

Érték szerinti viszonylag elterjedt, ismeretes. A legelső példában például érték szerinti történik. C-ben gyakorlatilag minden paraméter értékben adódott át. Kis finomhangolás történt, amikor a tömbök első elemre mutató pointerként adódtak át. Azt megnézzük, ha hagyományos függvényhívásoknál ha ilyen paraméterátadást használunk, mire is számíthatunk? Alapesetben azt fogjuk látni, hogy C-hez / C++-hoz hasonlító megoldások fognak születni.

Valid C kódnak minősülhet majd a következő példa is:

```
void f( int i )
{
        ++i;
        …
}
```

Ahogy az látszódott, ennek az érték szerinti átadásnak át tudunk adni egy változót (int s = 2), konstanst (5), kifejezést (s+4) is. Ennek az s-nek az értéke átmásolódik az *i*-be, és ott egy új lokális változó jön létre. Ez azért praktikus, mert ha megnöveli az értékét, akkor az *i* értékét változtatja meg, de *s* értéke marad utána is 2. Ez azért praktikus, mert jól látszódik, hogy ha átadom ezt a változót, a változó értéke nem változhat meg. Ha ránézek a kódra, láthatom, hogy az *s* értéke hol változott meg, vagy honnan változhat meg. Nem kell analizálni a kódot. Másodrészt, ezt könnyen lehet implementálni. Ezek a lokális változók a stack-en adódhatnak át. Amikor van a stack-en van a függvényünk, annak van egy *stack-frame*-je, aminek egyik része a *lokális változók*, a másik pedig a *függvény paramétereinek* a része. Tök ugyanaz, csak annyiban másabb, hogy az *s* kezdőértéke onnan adódik, ahol meghívtam a függvényt. Ennyi a különbség a lokális változó és a függvény paraméter között.

Ez a függvény paraméter megváltozik. Ha megváltozik az *i*, az kifele nem közvetít információt, csak befele tud. Ennek az tökéletes ellentéte a **cím-szerinti** átadás. Nézzük, hogy az hogy működik.

Első körben képzeljük azt, hogy van egy saját nyelvünk, mi találjuk azt ki. Alapértelmezetten cím-szerint adunk át paramétert. Ez most nem C, nem C++ lesz, hanem csak egy példa kedvéért kitalált valami. Itt cím-szerint adjuk át a paramétert.

```
void f( int s )
{
        ++s;
}
```

Itt annyiban másabb, hogy ha konstanst (2) adok át, a fordítási hiba! Mivel cím szerint nem tudjuk meghívni a konstanssal a cím szerinti átadást. Ha viszont egy *int x = 3; f(x)*;-et hajtok végre, hogy ha itt x-re hivatkozok, az teljesen ugyanaz a tárterület lesz. A *x* tárterületét az alprogramban az *s*-en keresztül fogom elérni, ennek hatására *x* értéke megváltozik, és 4 lesz. Ha cím szerint adok át paramétert, akkor tárterület alapján az *s* és az *x* ugyanaz. Mondhatjuk azt is, hogy a függvényen keresztül *s* álnéven elérjük a kinti *x*-et is.

Ez a paraméterátadás C-ben anno nem létezett! C++ viszont azt mondta, hogy erre időnként szükségünk lenne. Képzeljük el azt a függvényt, amely paraméterül kap két lokális változót, és ezek értékét felcseréli. C++-ban erre kitalálták a **referenciák** fogalmát. Referenciákkal tudjuk kifejezni a cím-szerinti paraméterátadás tulajdonságait.

```
void swap( int& a, int& b)
{
	int tmp = a;
	a=b;
	b=tmp;
}
```

Hogy tudjuk ezt a függvényt meghívni C++-ban?

```
int x = 5;
int y = 8;
swap(x, y);
```

Ilyenkor fel fog cserélődni a két változó értéke. Ezek után *y* értéke lesz 5, *x*-é pedig 8. Független attól, hogy a függvényben milyen névnek nevezem el a lokális változókat, ugyanis az *a* és *b* álnevekkel el tudom érni az *x* és az *y* tárterületet. A referenciát a & jel jelöli.

C-ben gyakran felmerülő probléma volt ez, viszont erre egy olyan megoldást találtak ki, hogy a C-ben a függvény vár két memóriacímet:

```
void swap( int* a, int* b)
{
	int tmp = *a;
	*a = *b;
	*a = tmp;
}
```

Ha ezt a *swap* függvényt akarom meghívni, arra az a megoldás, hogy ha van

```
int i = 3;
int j = 8;
swap(&i, &j);
```

akkor ne keverjük össze a C++-szal! C-ben a & ugyanis a címképzés jele. Az okozza a cím-szerinti paraméterátadást, hogy szemmel egy idő után nehéz követni, ha sokszor adom át ugyanis, akkor nehéz észrevenni, hogy hol változik meg pontosan az eredeti változóm értéke. Ezzel szemben a C-s paraméterátadásnál látszik, hogy először átadom a változóm címét, és nem a változót másolom le. Volt lehetőség a C-nél, hogy írjunk egy olyan függvényt, amely megváltoztatja a lokális változót.

Érdekes, hogy amikor jött a C#, akkor egyrészről a függvénynél is ki kell írni, hogy az cím szerinti paraméterátadás, illetve meghívásnál is kell jelezni. C++-ban nincs így, mert a referencia jelzi ezt.

Miért akarta a referencia fogalmát bevezetni a C++, ha C-ben a memóriacímes megoldás működött? Válasz: a C-s megoldásban túl volt bonyolítva. Nem azt fejeztük ki a kóddal, amit ki akartunk fejezni. C++-ban csak álnevet adtunk a változóknak, de C-ben túlbonyolítottuk memóriacímekkel és tárterületekkel. Főleg, hogy ha volt egy pointer értékünk, hogy *swap(&i,0)*, akkor lefordul a kód. Viszont bele kell kódolnom, hogy ha nullpointerrel akarok dereferálni, akkor ilyen-olyan kivételt lekezeljen, vagy bízzak benne, hogy a felhasználó nem lesz olyan, hogy nullpointert akar használni, mert futási idejű hibát fog adni. C++-ban az érdemi különbség, hogy a referencia garantálja, hogy ha van mögötte egy pointer, akkor nem tudom meghívni nullpointerrel, mert ott már vár egy meglévő tárterületet. A másik dolog viszont, hogy ha le kell ezt is írni, akkor megint túlbonyolítjuk a kódot. Ezért is vezette be a C++ a pointer fogalmát. Látszik, hogy a pointer és a referencia két, szorosan összekapcsolódó dolog. Nézzünk erre egy példát:

A pointerek C-ben és C++-ban is használható fogalmak. Ezzel szemben a referenciák C++-ban jelentek már meg. Nézzük meg az alapvető különbségeket. A pointer olyan változó, amelynek futási időben memóriacím. A referencia ezzel szemben egy álnév. Más a kialakult logikája a két eszköznek. A *swap*-et meg tudjuk csinálni pointerekkel és referenciákkal. A különbség a fenti példákban látszik. Pointereknél ellenben a dereferálás / címképzés is megjelenik. Ha van egy referenciám egy változóra, nem tudjuk őket megkülönböztetni, ez a fogalom utóbbinál nincs jelen, hiszen a kettő ugyanaz. Ha egy compiler eljátssza, hogy a referencia mögé felvesz egy pointert, figyelnie kell, hogy majd dereferálja az egészet.

A referencia mindig ugyanannak az álneve marad. A pointer, míg le nem tiltjuk, ellenben változhat, hogy hova mutat. És megintcsak valid állapot a pointernél, hogy nem mutat sehová, azaz lehet *nullpointer*, teljesen valid állapot. Majd később megváltozhat, hogy hova mutat, de nem kötelező, hogy hova mutasson. Referenciánál nincs erre lehetőség, nincs rá nyelvi eszköz, hogy megváltoztassam, nincs nullreferencia, nincs értelme. Nemcsak hogy nincs, még értelmezhetetlen is!

Hasonlóképp, referenciát köteles vagyok inicializálni. Ha azt mondom, hogy *int s=3; int &r = s,* akkor tulajdonképpen egy tárterülethez két nevem is van. C++-ban tudunk név nélküli tárterületet is létrehozni, míg egy tárterülethez két nevet is adni. Olyat nem tudok viszont csinálni, hogy *int &c*; aminek nincs meg az álneve, mert nincs rá későbbi eszközöm, ez már így fordítási hiba. A referenciát kötelező inicializálni, szemben a pointerekkel.

Azzal együtt ténylegesen nem hozott be a C++-ba olyan dolgot, ami korábban ne lett volna. Ha pointerek vannak mögötte, akkor olyan rossz dolog nem lehet. ~~Olyan nincs, hogy valami nem sörnyitó.~~ Legrosszabb esetben, ha csinálunk egy ilyen *swap* függvényt, legrosszabb esetben pointer lesz mögötte, nem tud úgy elszállni, hogy nullpointer lesz mögötte.

Mondhatjuk azt, hogy van konstans referenciánk is. Azt is, hogy *const int &c = r;* akkor még egy álnevet bevezettünk a korábbi tárterülethez. Ez egy olyan tárterület lesz, ami nem változhat meg. *c*-n keresztül hivatkozott tárterület nem változhat meg, mivel az konstans lesz. Viszont mostantól kezdve, ha bevezettük a referenciák fogalmát, nézzük meg, hogy nagy jelentőségük van.

Ha így írjuk meg a *swap*-et, hogy:

```
void swap( int a, int b )
{
        int tmp = a;
        a = b;
        b = tmp;
}
```

és van valahol két változóm, legyenek például *int s = 2; int t = 6;*, és azt mondom, hogy *swap(s,t);*, akkor ez lefordul. Csak az a gond, hogy erre nem tud a compiler hibajelzést adni, hogy ez nem az, amit én akarok. Végeredményben *s* értéke 2 lesz, *t* értéke pedig 6. Le fog fordulni (miért ne fordulhatna le?), ahol a függvény érték szerint veszi át a paramétereket, létrehoz két másik lokális változót, azok értékét kicseréli, de az eredeti értékek maradnak. Itt van a jelentősége, hogy a plusz 1 karakter (&) oda van-e írva.

Nézzünk egy másik példát erre. Tegyük fel, hogy egy olyan függvényt szeretnénk írni, ami beolvas a standard inputról számokat, és ezeket a számokat vektorban eljuttat a hívóhoz. Szeretnénk írni két függvényt, C++ értékben vett függvényt. Két lehetőségünk van. Az első megoldás azt mondja, hogy

```
void read(std::vector<int>& v)
{
	v.clear();
	int i;
	while(std::cin>>i)
	{
		v.push_back(i);
	}
}
```

Ezt már nem kell visszaadnom, referenciaként megkaptam, *v* álnéven dolgoztam vele. Azonban van erre egy másik megoldásunk is.

```
std::vector<int>read(){
{
	std::vector<int> ret;
	int i;
	while(std::cin>>i)
	{
		ret.push_back(i);
	}
	return ret;
}
```

Az első *read*-nél, ha lehagyom az &-t, akkor létrehozunk egy másolatot, amelybe belerakjuk az elemeinket, de az eredeti vektort békén hagyjuk. Ha hozzáírjuk a &-t, akkor a *v.clear()*-ig nem érdekel, mi volt az eredeti vektorban. A második esetnél, ha azt mondanánk, hogy azt referencia szerint adnánk vissza, hogy a belső lokális változó meg fog szűnni, de az utolsó sornál meg volt szűnni a tárterület, és az jön, amiről előző EA-n volt szó. Lokális változóra se pointert, se referenciát ne adjunk vissza. Ezért fontos, hogy bár van referencia visszatérési érték, de az nem jó, ha egy lokális változóra adunk referenciát. Mindkét esetben igaz, hogy bár kiírom, bár nem, de míg utóbbi esetben ha kiírjuk, akkor olyan változóra mutatunk, ami megszűnik. Ha viszont az elsőnél nem írjuk ki, akkor csak egy másolattal dolgozunk. Ezért nagyon hangsúlyos ezen karakterek használata.

Hogy még bonyolultabb legyen a téma, a másodiknál az az elképzelés, hogy mi történik, ha érték szerint adjuk vissza? Kimásoltatjuk az adatokat. Viszont ott van az a mondás, hogy „Próbáljuk meg kevesebbre venni a másolást!", mert a felesleges másolások csak lassítják a kódot. Ezért vezették be a **konstans referencia** fogalmát.

Nézzünk ezekre példákat:

```
void f(std::vector <int> v)

void g(const std::vector<int>& v)
```

A f függvény egy másolattal dolgozik, míg a *g* egy referenciával. Viszont utóbbinál elkerüljük a másolást is, ott nem történik ilyen, és nincs deallokáció. Előbbinél mindkettő van, és meg kell fizetni mindkettőt futási időben.

Nézzük az eredmény-szerinti átadást. Térjünk vissza a fiktív nyelvünkhöz.

```
void f(int x)
{
```

```
        x=2;
}
```

Ha ezt valahol meghívom, hogy *int s=7; f(s);*, akkor az eredmény szerinti paraméterátadás úgy viselkedik, hogy ide olyan változót kell megadni, amibe tud írni, mert ez kifele közvetít. Az a logikája, hogy a függvényben létrejön egy új lokális változó. Viszont itt a függvényhívás végén másolódik bele az eredeti, aktuális paraméterbe. A végén végül *s* értéke 2 lesz.

Az érték / eredmény szerinti átadás annyival másabb, hogy ott befelé is másolódik, meg visszafelé is másolódik, tehát két másolás lesz. Ha azt mondom, hogy

```
void f ( int s)
{
        ++s;
}
```

akkor s-be bemásoljuk az eredeti változót, s értékét növelem, és azt másolom vissza. Ha *int x = 5, f(x);*, akkor x bemásolódik s-be, *s* értékét növelem 1-gyel, és a 6-ot visszamásolom *x*-be. Ezeket le lehet modellezni C++-ban is.

Van még végül a név-szerinti paraméterátadás. Nézzük először a viselkedését. Legyen megint a fiktív nyelvünk, amiben ez a függvényünk van:

```
void f (int x)
{
        …
}
```

Ha meghívom egy konstanssal *f(3)*, akkor úgy viselkedik, mintha érték szerint adnám át. Ha átadok egy változót (*int s; f(s);*), az olyan ,mintha cím szerint adnánk át. Ha viszont átadok egy olyat, hogy *f(x+y+2*k);*, akkor időközben ezek ha megváltoznak, más és mást jelenthet, *f* tartalma újra és újra kiértékelődhet. Ilyennel már találkoztunk. A makrók szövegszerű behelyettesítése is hasonlóan működik. Egy ideje azonban már nem nagyon használjuk ezeket, azonban a *Scala* programozási nyelv tervezői gondolták, hogy visszahozzák (mert miért ne?). De nézzük meg, mert ilyennel mi már tényleg találkoztunk.

Ha azt mondom, hogy írunk egy ilyen makrót, amit nem szeretünk, mert szerencsétlen a paraméterátadásuk:

```
#define SQ(x) ((x)*(x))
std::cout<<SQ(4);
std::cout<<SQ(i);
```

Megbeszéltük, hogy ha olyan szerencsétlen módon dolgozunk, hogy ha egy mellékhatással rendelkező kifejezést adunk át:

std::cout<<SQ(i++);

akkor itt *i* értéke kettővel nő, ugyanis itt az *i* értéke többször változik.

Ha írok azonban egy olyan makrót, hogy

```
#define NPRINT(N,X) for(int i=0;i<(N);++i)\std::cout<<(X);
```

Ha ezt meghívom, a következő konstrukció történik:

NPRINT(3,4);, akkor ez háromszor kiírja a 4-es értéket! 444

Mondjuk viszont ha van egy *int i=4;* és azt mondom, hogy *NPRINT(3,i);*, ez mást fog kiírni. Egészen pontosan 012-t. Miért? Mert berakja az i értékét az X-be, és amikor ezt keresi, akkor ő a makróban lévő *i*-re gondol, és nem arra az *i*-re, amit én átadtam neki. Ugye milyen gyönyörű?