C++11 – Smart pointerek

Ebben a leírásban szeretnék egy rendkívül alapszintű bevezetést adni a C++11 szabvány egyik legnagyszerűbb újításához, a smart pointerekhez. Jó szívvel ajánlom mindenki számára, akik komolyan gondolják a programozást C++-ban, és mivel az egyetemen nem nagyon fogunk találkozni ezzel, ezt orvosolandó szeretném a legalapvetőbb dolgokat lefektetni ebben az írásban. Őszintén remélem, hogy sokak hasznára fognak válni az itt leírtak. Minden igyekezetem ellenére is előfordulhatnak hibák a leírásban, ezért ezekkel és minden mással kapcsolatban is szívesen fogadok bármilyen megjegyzést facebook-on és/vagy a tuto558@gmail.com e-mail címen és/vagy az egyetemen... Vágjunk hát bele!

Amiket meg fogunk tudni:

- mik azok a smart pointerek és mire jók
- birtoklási viszonyok kezelése (ownership transfer)
- smart pointerek közötti különbség (unique_ptr, shared_ptr, weak_ptr)

Nagyon fontos, hogy ahhoz, hogy az itt leírtak érthetőek legyenek, minimum a BevProg2-es szinten érteni kell a hagyományos („raw”) pointerek használatát és a dinamikus memóriakezelést, ezért ajánlom mindenkinek, hogy előbb azokkal legyen tisztában.

Tudjuk, hogy nagyon oda kell figyelni, amikor a freestore-on (dinamikus memóriában, avagy a „heap-en”, ahogyan azt sokszor – hibásan – emlegetni szokták) dolgozunk:

```
1{
2    int *correctUsage { new int { 42 } };
3    delete correctUsage;
4}
5
6{
7    int *memoryLeak { new int { 42 } };
8}
9
10{
11   int *doubleFreeOne { new int { 42 } };
12   int *doubleFreeTwo = doubleFreeOne;
13
14   delete doubleFreeOne; // OK
15   delete doubleFreeTwo; // oops, deleted same object
16}
```

Ezen a képen jól látszik a helyes használat az első esetben. A második esetben, mivel elfelejtettük meghívni a `delete`-et, a memóriaterület nem szabadult fel, és mivel az egyetlen pointerünk, amivel rámutattunk, kiment a scope-ból, már nem is fogjuk tudni felszabadítani azt a memóriaterületet a futás során („memóriaszivárgás”, memory leak történik).

Az is megtörténhet, hogy véletlenül két pointert állítunk ugyanarra az objektumra, azután ezt elfelejtjük, és véletlenül kétszer töröljük. Ez undefined behavior; jobb esetben a programunk jól reprodukálható módon elhasal, rosszabb esetben nem vesszük észre a hibát.

Küszöböljük ezt ki most úgy, hogy létrehozunk egy olyan típust, ami mutató lesz egy int-re, de nem kell (és nem is szabad) felszabadítanunk a delete-tel.

Ezután használhatjuk úgy, mint egy int-re mutató pointert:

```
1class NaiveIntSmartPtr {
2    int *pointer;
3
4public:
5    NaiveIntSmartPtr(int *pPointer) : pointer { pPointer } {}
6    ~NaiveIntSmartPtr() { delete pointer; }
7
8    int *get() const { return pointer; }
9};
10
11{
12    NaiveIntSmartPtr test { new int { 10 } };
13} // out-of-scope -> destoyed -> no leak
```

Jól látható, hogy a scope megszűnése előtt nem szabadítottuk fel az erőforrást delete-tel, de itt nincs is rá szükség, hiszen amikor a „test” kimegy a scope-ból, automatikusan lefut a destruktora, ami meghívja a delete-et, amire felszabadul a memória.

Nem kellett `delete`-et írnunk és nincs memory leak! Fantasztikus!

Persze ez még egy nagyon buta smart pointer lenne, hiszen nem generikus, vagyis nem működik akármilyen típussal. Ezt template segítségével viszonylag könnyen kezelni lehetne. A másolás és a mozgatás is kérdéses pont. Nem ártana megtervezni és implementálni a copy-konstruktort, értékadó operátort, stb. Emellett még azt sem tudjuk, hogy ha a standard tárolókban (`vector`, `list`, `map`, ...) akarnánk használni ezt, az hogyan történjen és mik a következmények? És még sorolhatnám...

Nos, van egy jó hírem: nem kell ezeken gondolkodnunk, ugyanis ezt már megtették előttünk. Ez az első – és egyben a „legegyszerűbb” smart

pointerünk: a **`std::unique_ptr`**. Ez nagyjából ugyanazt a problémát oldja meg, amit meg akartunk oldani a fenti saját készítésű osztállyal, csak sokkal okosabban és biztonságosabban. A smart pointerek használatához include-olni kell a `<memory>` header fájlt.

Nézzük hát meg, hogy mit is tud pontosan a `unique_ptr`:

```
1{
2    std::unique_ptr<int> test { new int { 10 } };
3}
4
5{
6    std::unique_ptr<double>      smartPointers { new double { 3.14 } };
7    std::unique_ptr<std::string> are { new std::string { "extremely" } };
8    std::unique_ptr<char>        versatile { new char { '!' } };
9}
```

A felső scope-ban láthatjuk ugyanazt, amit az előbb oldottunk meg a saját osztályunkkal. Az alsóban pedig láthatjuk a `unique_ptr` generikus voltát: bármilyen típusra tudunk smart pointert állítani.

A fenti két példában nincs szükség delete meghívására a memóriaterület felszabadításához. Amikor a `unique_ptr` kimegy a scope-ból, automatikusan felszabadítja maga után a memóriát. Ez minden esetben **garantált!** Ezzel tehát egyrészt kiszűrjük a programozói hiba lehetőségét (ha nem kell felszabadítani `delete`-tel, akkor nem lesz mit elfelejteni, hurrá! – mert igenis, amit el lehet felejteni, azt előbb utóbb el fogjuk felejteni).

Emellett, ha egy ponton kivétel dobódik - és a hagyományos („raw”) pointerek esetén a vezérlés nem jutna el a manuálisan begépelt `delete`-ig -, akkor is felszabadul a memória, hiszen a `unique_ptr` kimegy a scope-ból és felszabadítja ezzel a memóriaterületet, ahova mutatott.

Nagyszerű, örökre elfelejthetjük a raw pointereket! Vagy mégse?

A raw pointerek továbbra is hasznosak, de csak a megfelelő esetekben, az alapszabály így foglalható össze:

- Létre akarsz hozni egy objektumot a free-store-on, dinamikus memóriában: **használj smart pointert!** (vagyis aki komolyan gondolja a C++ programozást, annak production kódban *el kell felejtenie a* `new` *és* `delete` *párokat, ezeknek a használata maradjon csak meg az egyetemen...* és helyette, ahol azt akarjuk, hogy a pointerünk birtokolja azt a memóriaterületet, mindenhol használjunk smart pointereket. *Persze ettől még illene ismerni, hogy hogyan működik a dinamikus memóriafoglalás...*)
- Hivatkozni akarsz valamire, anélkül, hogy birtokolná a pointer az objektumot? (Vagyis már egy létező objektumra akarunk rámutatni, de nem ott foglaljuk a memóriát és nem is akarjuk felszabadítani): **használj raw pointert!**

Vigyázat! Az erőforrás akkor mindenképp fel fog szabadulni, amikor a smart pointer kimegy a scope-ból, tehát így még mindig „lábon lőhetjük magunkat”, ha a smart pointerből kinyert nyers pointer scope-ja nagyobb lenne, és nem szűnne meg biztosan hamarabb, mint a smart pointer maga.

```
int *footShot;

{
    std::unique_ptr<int> bullet { new int { 1337 } };
    footShot = bullet.get();
} // !!!

std::cout << *footShot; // Undefined behavior strikes back
```

Ne felejtsük el továbbá, hogy alapból a freestore-on való fölösleges memóriafoglalás kerülendő; ha lehetséges, preferáljuk az automatikus élettartammal való allokálást.

```
{
    std::string object_with_auto_storage { "Hello world!" };
    std::string *raw_ptr_to_auto_object = &object_with_auto_storage;
}
```

Ha pedig csak rá akarok mutatni egy ilyen objektumra, ami automatikus storage duration-nel rendelkezik („a stack-en” hozzuk létre, ha valaki csak így, pontatlan kifejezéssel ismerné), akkor is nyugodtan használhatunk raw pointert. Természetesen itt ugyancsak figyelni kell arra, hogy a raw pointer élettartama nehogy véletlenül nagyobb legyen, mint az objektumé, amire mutat.

És akkor jöjjön a unique pointer egy nagyon fontos tulajdonsága: ahogyan a neve is mutatja, ő „unique”, vagyis **„egyedi”**. A unique pointer kizárólagos tulajdonjoggal („ownership”) rendelkezik a mutatott objektum felett. Ebből következik, hogy *ugyanarra az objektumra egyszerre csak egy unique pointer mutathat.* Hiszen ha kettő vagy több unique pointer is mutathatna egy tartalomra, akkor már nem lenne kizárólagos ownership.

Képzeljük csak el: ha kettő tudna ugyanarra mutatni, majd az egyik kimegy a scope-ból (és viszi magával a tartalmat/felszabadítja az erőforrást), a másik pedig még létezik, akkor olyan területre kellene mutatnia tovább, ami már fel lett szabadítva, illetve a destruktora egy már egyszer felszabadított pointert próbálna meg ismét törölni. Ha ettől most kiráz a hideg, az jó dolog! *Ilyet nem akarunk, és szerencsére nem is tudunk csinálni* unique pointerrel.

Ebből következik, hogy a `unique_ptr` nem másolható (hiszen ha másolható lenne, az előbb leírt eset állna fenn). Meg is próbálhatjuk kísérletképp másolni, az eredmény fordítási hiba lesz.

A `unique_ptr` által menedzselt objektum paraméterként való átadására több lehetőség van. Íme egy koránt sem teljes lista:

```
struct Resource {};

// sometimes, for "sink" functions
void passByValue(std::unique_ptr<Resource> ptr);

// often, for 'observing-only' functions,
// i. e. when function need not know the lifetime
// when the object can be missing (a pointer can be nullptr)
void passByRawPtr(const Resource *ptr);
void passByRawPtrNonConst(Resource *ptr);

// observing-only functions, where
// the object is obligatory
// (a reference can't be null)
void passByRef(const Resource &ref);
void passByRefNonConst(Resource &ref);

// technically valid, but not of much use
void passSmartPtrByConstRef(const std::unique_ptr<Resource> &ptr);
```

A smart pointer átadása const referencia szerint is működik, ám itt explicit jeleznünk kell, hogy milyen smart pointerről van szó, ezért nyilván csak azzal fog működni. Ez tehát egy technikailag helyes, ám nem javasolt megoldás (pontosan azért, mert nem univerzális – a hívott függvénynek tudnia kell arról, hogy a hívó milyen élettartamú objektummal hívja, ez pedig egy fölösleges szemantikai függést vezet be).

```
{
    std::unique_ptr<Resource> res { new Resource() };
    passByRawPtr(res.get());
}
```

Ha tehát olyan függvényt szeretnénk, ami minden smart pointer esetében működik, akkor a legcélszerűbb megoldás az, ha raw pointer szerint vesszük át a paramétert, amennyiben az lehet nullptr, vagy referencia szerint akkor, ha nem. Ilyenkor a hívási helyen `std::unique_ptr::get()` függvénnyel (referencia szerinti átadás esetén pedig ennek a dereferálásával) adjuk át a függvénynek a raw pointert.

A másolás ugyan nem megengedett `unique_ptr` esetében, de a mozgatás igen! A „move semantics” szintén egy új feature C++11-ben, amivel hatékony értékadást/konstrukciót kivitelezhetünk, elkerülvén a sokszor fölösleges másolásokat, vagy éppen megoldván olyan eseteket, ahol a hagyományos másolás művelet egyáltalán nem végezhető el a típus szemantikus tulajdonságai miatt (pl. `unique_ptr`). Ezeket hivatott megoldani az `std::move()`, amely jobbérték referenciává konvertálja az argumentumát.

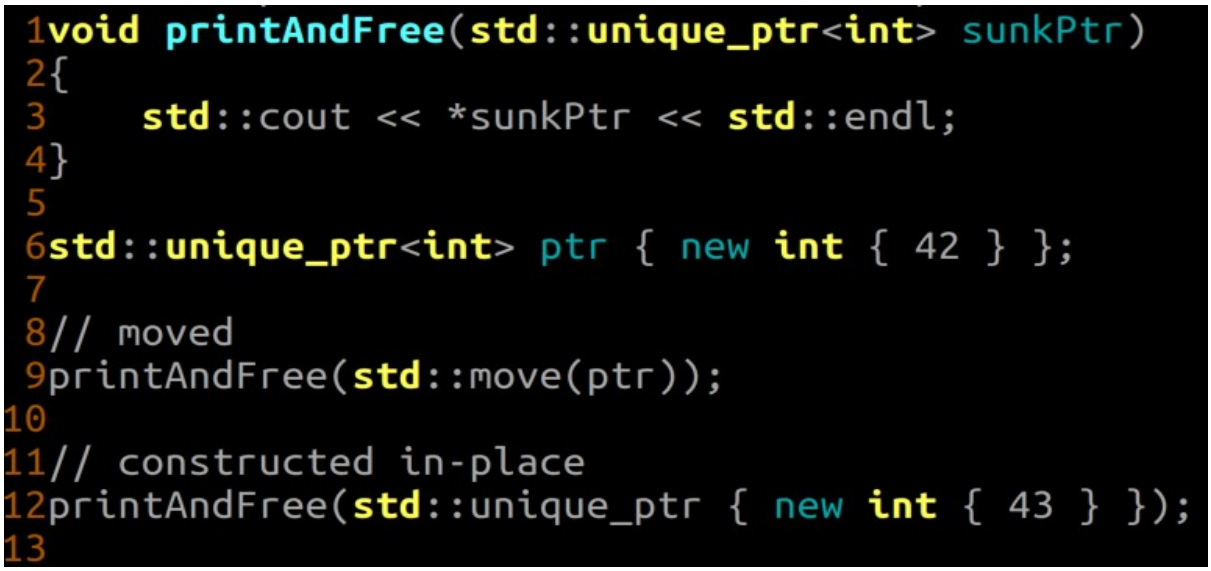
Egy objektum jobbérték referenciát átvevő, azaz „move” konstruktorát pedig úgy kell megírni, hogy az eredeti objektum adatait úgy vegye át az új objektum, hogy törli őket az eredetiből. Ez a háttérben általában a pointerek kicserélésével, a lefoglalt erőforrás (pl. memória, fájl descriptor, stb.) „ellopásával” történik, így nem történik felesleges allokáció-deallokáció, megússzuk a másolást.

```
1std::string source { "hello world" };
2std::string target { std::move(source) };
3
4// now 'target' is "hello world"
5// and 'source' may be anything (commonly the empty string)
```

Ebben az esetben a `move()` használata után eredeti váltózó (`source`) „üres”! A tartalom már kizárólag csak a `target` mögött van ott, az eredeti objektum egy nem specifikált, de *általában* „szemantikus nulla” állapotban lesz (pl. a string esetén egy ez **lehet** üres string, a `unique_ptr` esetén **lehet** null pointer, a standard azonban mást önmagában nem garantál).

A folyamatos ownership transfer azonban összezavarhat és még mindig nem a leghatékonyabb megoldás, így csak akkor használjuk, amikor tényleg szükség van rá, egyébként egészen nyugodtan használhatunk akár constref, akár pointer szerinti paraméterátadást.

Nem másolható objektumot is lehet érték szerint átadni, akkor, ha vagy explicite move-oljuk, vagy eleve másolást nem igénylő módon hívjuk (pl. egy temporary-val). Ez a `unique_ptr`-t átvevő függvényeknél egy esetben értelmes: ha egy ún. „nyelő”, „sink” függvényünk van, amely átveszi az ownership-et az átadott pointer felett. A fenti legelső függvényt is meg tudjuk tehát hívni, ha mozgatás vagy helyben való konstrukció történik:



```cpp
void printAndFree(std::unique_ptr<int> sunkPtr)
{
    std::cout << *sunkPtr << std::endl;
}

std::unique_ptr<int> ptr { new int { 42 } };

// moved
printAndFree(std::move(ptr));

// constructed in-place
printAndFree(std::unique_ptr { new int { 43 } });
```

A kérdés az, hogy mi a teendő akkor, ha több pointert is szeretnénk az adott objektumra állítani, tehát nem akarunk kizárólagos ownership-et? Erre kínál megoldást az **`std::shared_ptr`**. Ez egy *referenciaszámlált* pointer, ami azt jelenti, hogy folyamatosan számon van tartva, hogy éppen mennyi `shared_ptr` mutat az adott objektumra. Amikor a referenciaszámláló eléri a nullát, a memória felszabadításra kerül. A pointer azért „shared”, mert itt az összes pointer egyszerre birtokolja ugyanazt az objektumot, és amint az utolsó shared pointer is kimegy a scope-ból, az erőforrás felszabadításra kerül a delete használata nélkül. Tehát nem kell figyelnünk olyan dolgokat, hogy „melyik pointer birtokolja az objektumot?” és „hol hívjam a `delete`-et?”, „nincs-e olyan pointer, aminek a scope-ja nagyobb, mint ahol a delete történt, ezért érvénytelen memóriaterületre mutat (dangling pointer)?”, stb, stb. Mindezek feleslegessé válnak, ha shared pointert használunk. Egészen addig nem történik meg a memória felszabadítása, amíg még van `shared_ptr`, ami arra az objektumra mutat.

Az alábbi példa jól mutatja, hogy mi történik ilyenkor:

```cpp
struct Object {};

{
    std::shared_ptr p1 { new Object }; // refcount == 1

    {
        auto p2 = p1; // refcount == 2
        auto p3 = p1; // refcount == 3

        {
            auto p4 = p3; // refcount == 4
        }
        // refcount == 3
    }
    // refcount == 1
}
// refcount == 0, deleted
```

Nézzük a következő példát:

```cpp
struct Resource {};

struct NaiveGameEffect {
    Resource animation;
    Resource backgroundTexture;
    Resource sound;
};

```

Itt, amikor létrehozunk egy új `NaiveGameEffect` objektumot, vagy érték szerint átadjuk valahol, az erőforrások mindenhol újra lefoglalódnak és felszabadulnak, ami nem túl hatékony. Viszont ha shared pointereket használunk, nem fog történni költséges másolás, és mindaddig nem szabadul fel az erőforrás, amíg lesz életben levő `NaiveGameEffect` objektum, amiből mutat rá shared pointer. A példa talán egy kicsit erőltetett, de jól mutatja a `shared_ptr` használatának egy tipikus esetét: ha olyan objektumokról van szó, ahol a másolás szemantikailag lehetetlen (pl. widgetek), esetleg nehézkes vagy költséges lenne (mint itt), de szükségünk van az objektum élettartamának automatikus menedzselésére pointerezés mellett is. Íme shared pointerekkel:

```
 9struct GameEffect {
10    shared_ptr<Resource> animation;
11    shared_ptr<Resource> backgroundTexture;
12    shared_ptr<Resource> sound;
13};
```

És végül, de nem utolsó sorban egy widget-es példa kód nélkül. Ilyet mindenkinek kellett csinálnia, ezért talán még érthetőbbé válik ezen a példán keresztül.

Szeretnénk csinálni egy kiválasztó widget-et. Ebben a kiválasztó widget-ben nyilván vannak elemek, és ezeket ki tudjuk választani. Ezt a beadandóban is úgy kellett megvalósítanunk, hogy nem mindegyik elem látszik, és időben változik, hogy mikor melyik látszik (pl. fel/le görget a felhasználó, stb).

Minden elemet szeretnénk eltárolni attól függetlenül, hogy az éppen látszik-e vagy sem. Egy ilyen elemet viszont problémás lenne másolni, ezért ehelyett azt tesszük, hogy minden elemre ráállítunk egy `shared_ptr`-t, ezeket pedig egy `vector`–ban eltároljuk. Amikor látni kell az elemet, akkor a kiválasztó widget-ünkhöz a megfelelő helyre hozzáadjuk ezt az elemet (ráállítunk egy `shared_ptr`-t), amikor pedig nem kell látni az elemet, akkor letöröljük/másikra állítjuk rá azt a pointert, de ettől az előzőleg mutatott tartalom még mindig ugyanúgy elérhető, hiszen egy `shared_ptr` még mutat rá a vectorból.

Mi történik, ha egy elemet el akar távolítani a felhasználó? Ilyenkor töröljük a `vector`–ból, így a modellben már nem lesz benne, majd végrehajtjuk a törlést magából a kiválasztó widget-ünkből is (átállítjuk a pointert a következő elemre, egy üres elemre, stb). Ekkor - mivel a kiválasztó widget-ből való törléskor már nincs meg az elem a vectorban – már törlődni fog az objektum, fel fog szabadulni a memória, hiszen nem mutat már rá egy `shared_ptr` sem.

Az egyetlen smart pointer, amiről még nem beszéltünk, az nem más, mint az **`std::weak_ptr`**. Ez szorosan hozzátartozik a shared pointerhez (gyakorlatilag a „segédosztálya”), olyan objektumokra tudunk mutatni vele, amire már mutat `shared_ptr`. A `weak_ptr` olyan smart pointer, ami nem növeli az adott objektumra vonatkozó referenciák számát.

Mikor lehet erre szükség? Nézzük a következő esetet:

```
1struct Window;
2
3struct Button {
4    std::shared_ptr<Window> window;
5};
6
7struct Window {
8    std::shared_ptr<Button> button;
9};
10
11{
12    auto winPtr = std::make_shared<Window>();
13    auto btnPtr = std::make_shared<Button>();
14
15    winPtr->button = btnPtr;
16    btnPtr->window = winPtr;
17}
```

Aminek itt tanúi lehetünk, az a körkörös referencia. Ha van egy ablakunk, amiben shared pointerrel hivatkozunk a rajta levő gombra, a gomb pedig szintén egy `shared_ptr`-rel hivatkozik az ablakra, akkor az ablaknak és gombnak is folyamatosan legalább egy lenne a rá vonatkozó referenciák száma.

Hiszen gondoljunk csak bele: az ablakból mutatok egy shared pointerrel a gombra. Ez a gomb csak akkor törlődne a memóriából, ha az ablakunk is törlődik. De az ablakunk nem tud törlődni, hiszen az meg csak akkor tudna, ha a gomb törlődik, mert abból mutat rá egy `shared_ptr`. Így gyakorlatilag egymást „tartják életben”. Vagyis ezek soha nem kerülnének felszabadításra, nem törlődnének a memóriából.

Ez a ciklikus/körkörös referencia. Ezeknek a feloldására vezették be a `weak_ptr`-t. Mivel a `weak_ptr` nem növeli az adott objektumra vonatkozó referenciák számát (nem nő a referenciaszámláló), ezért a helyes használat:

```cpp
struct Window;

struct Button {
    std::weak_ptr<Window> window;
};

struct Window {
    std::shared_ptr<Button> button;
};

{
    auto winPtr = std::make_shared<Window>(); // refcount == 1
    auto btnPtr = std::make_shared<Button>(); // refcount == 1

    winPtr->button = btnPtr; // refcount == 2
    btnPtr->window = winPtr; // refcount == 1!
}

// button refcount == 1
// window refcount == 0
// -> deallocated
// -> button refcount == 0 -> deallocated
```

Ez elsőre bonyolultnak tűnhet, de gyakorlatilag birtoklási viszonyokról van szó. Akkor, amikor az ablakunk birtokolja a gombot, erős („strong”, „owning”) pointert használunk (`shared_ptr`), a gombunk viszont nem birtokolja az ablakot, csupán hivatkozik rá, ezért gyenge („weak”, „non-owning”) pointert használunk (`weak_ptr`). Ha így építjük fel a programunkat, akkor nem jöhetnek létre ciklikus referenciák és nem lesznek „örök életű” objektumok; ráadásul ez egy viszonylag könnyen megjegyezhető és alkalmazható szabály.

Feltehetnénk a kérdést, hogy akkor miért nem raw pointert használunk, hiszen az sem növelné a referenciák számát. A válasz egyszerű: raw pointereknél, a memória felszabadulása után kapunk egy dangling pointert (olyan pointer, ami érvénytelen memóriaterületre hivatkozik), ezt nekünk manuálisan kellene `nullptr`-re állítanunk. Egy `shared_ptr` mellett ez elég macerás lenne, folyamatosan figyelni, hogy létezik-e még az adott objektum, és explicit kellene a referenciaszámlálót figyelni, hogy mikor kell `nullptr`-re állítani, stb... hibalehetőségek tömkelege. A `weak_ptr` viszont önmagában kezeli ezt: ha a mutatott objektum nem létezik (mert már közben minden rá mutató strong pointer megszűnt, és deallokálódott), akkor a `weak_ptr` null pointerre fog kiértékelődni.

Remélem, hogy sikerült felkeltenem mindenkinek az érdeklődését és hogy ez a kis betekintés sokaknak egy biztos alapot jelenthet ebben a témában. Végül - de nem utolsó sorban - szeretném megköszönni ***Goretity Árpi*** *segítségét, aki szakmailag és formailag is kifogástalan állapotba varázsolta az itt bemutatottakat/leírtakat. A vele való folyamatos konzultálás nélkül nem jöhetett volna létre ez az írás! Köszönet érte!*

Bacsu Attila - 2015. május 3.