# Rekurzivitás 2. Rész

## A rekurzivitásban rejlő veszélyek. Óvintézkedések

Rögtön az elején tisztázni szeretnénk egy apró félreértést. A cikksorozat első részében (lásd CodeX 2004.szeptemberi szám „Rekurzivitás elegancia veszélyekkel") említett könyvtárbejáró program forrásának bemutatására a cikksorozat utolsó részében kerül sor. Elnézést a félreérthető fogalmazásért mindazoktól, akik az előző számban keresték a forrást.

Az előző részben ismertettük a rekurzivitás fogalmát, valamint a függvény hívásakor zajló háttértevékenységeket. Itt az ideje, hogy megvizsgáljuk a rekurzivitás gyakorlati hasznát továbbá a benne rejlő veszélyeket, illetve a hibák elkerülésének egy lehetséges megvalósítását.

A rekurzivitás mélysége, más szóval a rekurzív hívások száma, egyenesen arányos az „elfogyasztott" veremterülettel. A verem mindaddig nem szabadul fel, míg a legbelső függvényből vissza nem térünk. Gyakorlatilag arról van szó, hogy a függvény lokális veremterületének visszafejtése nem történik meg. Emlékeztetőül: a cikksorozat előző részében felsoroltuk azokat a háttértevékenységeket, amelyek egy függvény hívásakor végrehajtódnak. Ezek közül a legutolsó, a 7. pont a verem visszafejtése, amely kizárólag akkor hajtódik végre, ha a legbelső függvényből visszatérünk. Az ezt megelőző tevékenységek azonban minden egyes függvényhíváskor végrehajtódnak, ami a veremmemória szabad kapacitásának folyamatos csökkenésével jár.

A veremmemória a program számára allokált memória része, amely véges, így előbb-utóbb betelik. A jelenséget veremtúlcsordulásnak (angolul „stack overflow") nevezik, és kellemetlen hibajelenségek forrása. A verem alulcsordulása is bekövetkezhet (angolul „stack underflow"), ha a veremről többet emelünk le, mint amennyit ténylegesen ráhelyeztünk. Utóbbi ritka jelenség, mivel a verem karbantartása amint azt már említettük nem a programozó feladata. Azonban némi assembly tudással lehetőségünk van „belepiszkálni" a verembe, a megfelelő regiszterek módosítgatásával. Másrészt a verem tartalma a PUSH / POP utasítások segítségével is módosítható. Ilyesmire viszonylag ritkán van szükség, leginkább elhivatott bitfaragók vagy kísérletező kedvű programozók kezelik ilyen alacsony szinten a vermet. Amennyiben valóban elkerülhetetlen a verem tartalmának illetve regisztereinek közvetlen módosítása, legyünk nagyon körültekintőek, hiszen igen érzékeny területen „garázdálkodunk".

Most, hogy a bemutatott problémák kellőképpen elvették a kedvünket a rekurzivitás alkalmazásától, vizsgáljuk meg mégis miért nem száműzték ezt a módszert. A rekurzivitás nagyon elegáns megoldásokhoz vezethet, hiszen általában sokkal rövidebb kódot eredményez, mint a klasszikus megoldás. Idézet a napokban megjelent Gyakorlati C++ című könyvem, a „Rekurzív függvények. A veremtúlcsordulás" című szövegrészből:

Ha egy szám faktoriálisát szeretnénk kiszámítani, a legegyszerűbben egy rekurzív függvénnyel tudjuk ezt megvalósítani. Egy pozitív egész szám faktoriálisát a következőképpen számítjuk ki:

n! = n * (n-1) * (n-2) * (n-3) ... * 1
Azaz például 7! = 7*6*5*4*3*2*1

Megfigyelhető, hogy egy ismétlődő műveletsorról van szó dekrementálás és szorzás -, és ezt a megfigyelést ültetjük át a gyakorlatba rekurzív függvényünkön keresztül:

```
unsigned long fakt ( unsigned long long_in )
{
        if ( long_in > 1 )
        {
                return long_in * fakt( long_in - 1
);;
        }
        else // kisebb, mint 1? -> 0! = 1
        {
                return 1;
        }
}
```

Megjegyzés:

A fenti függvény gyengéjét viszonylag gyorsan észre fogjuk venni, ugyanis a unsigned long int felső határát (4.294.967.295) elég gyorsan elérjük: 12! = 5.748.019.200.

A rekurzív függvények veszélyessé válhatnak, különösképpen, ha működésük bonyolult, nehezen átlátható, és a kilépés nincs minden esetben biztosítva. Ezenfelül, a rekurzív függvények végrehajtását lassítja a függvényverem felállítása, a függvényparaméterek és lokális változók lemásolása ami minden egyes függvényhíváskor megtörténik (nehogy egy újabb függvényhívás felülírja az előző lokális változóit, paramétereit).

Ezáltal minden egyes függvényhíváskor csökken a veremmemória szabadon felhasználható területe, és amikor végképp elfogy, az ún. veremtúlcsordulással (stack overflow) van dolgunk. Ennek az ellenkezője a verem alulcsordulása (stack underflow), ez olyankor szokott bekövetkezni, amikor több elemet emelünk le a veremről, mint amennyit ténylegesen elhelyeztünk benne.
A verem túlcsordulása gyakoribb az alulcsordulásnál, mivel nagy számú rekurzív függvényhívással aránylag gyorsan be lehet telíteni a veremmemóriát főleg ha az adott függvénynek méretes lokális objektumai vagy érték szerint átadott paraméterei vannak.

Ha egy rekurzív függvény kilépési pontja nem teljesen egyértelmű, biztonsági intézkedésként deklarálhatunk egy statikus számlálót, amit minden függvényhívás előtt növelünk, illetve utána csökkentünk. Ha elértünk egy megszabott felső határt, kilépünk a függvényből:

```
#define MAX_CALL_DEPTH 10

void recursive()
{
        static int cnt;
        if ( cnt >= MAX_CALL_DEPTH )
        {
                cout << endl << "Kritikus hiba!
Maximális függvényhívás- mélység! " << endl;
                return;
        }

        ++cnt; // rekurzív hívás előtt
inkrementáljuk
        cout << cnt << " ";
        recursive();
        --cnt; // rekurzív hívás után
dekrementáljuk
        cout << cnt << " ";
}
// főprogramint main()
{
        recursive();
        return 0;
}
```

A függvényhívások örvénylését megállítjuk a 10. hívás után:

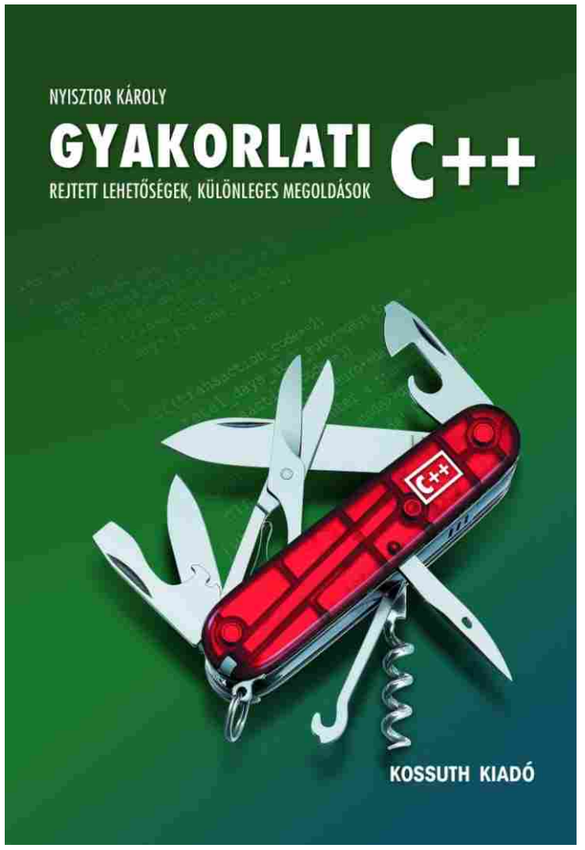
```
1 2 3 4 5 6 7 8 9 10
Kritikus hiba! Maximális függvényhívás-mélység!
9 8 7 6 5 4 3 2 1 0
```

MAX_CALL_DEPTH értéke elég nagy kell legyen ahhoz, hogy a függvény elvégezze feladatát, ugyanakkor meg kell akadályozza az eltúlzott rekurzióból származó veremtúlcsordulást - ha például függvényünk rekurzív módszerrel állományokat keres, értelmetlen az adott operációs rendszer által biztosított maximális könyvtár-mélységnél mélyebbre ásni)."

A rekurzivitásról szóló cikksorozat következő számában a függvényhívási konvenciókról lesz szó. A szabványos __cdecl C függvényhívási konvención kívül bemutatásra kerül még a Microsoft-specifikus __stdcall ( WINAPI), a __fastcall illetve a thiscall is.

Addig is tiszta vermet és kevés „elszállást"! ;)

Nyisztor Károly
nyisztor.karoly@evosoft.hu




**Nyisztor Károly** számos cikkét olvashattuk már a CodeXben. Szerzőnk egy 400 oldalas könyvvel lepett meg minket, amely átfogó képet szeretne nyújtani a C++-ról, elsősorban gyakorlati szempontokból elemezve a nyelv elemeit. A Gyakorlati C++ című könyv októberben jelent meg a Kossuth kiadó gondozásában.

Következzen egy rövid összefoglaló a könyvről: „Könyvünk a C++ nyelv alapjait és működésének elvontabb, finomabb aspektusait egyaránt feldolgozza az előfeldolgozástól a kivételkezelésig.

Olyan gyakorlati problémákra kapunk választ, mint például az állományok befordításakor fellépő végtelen rekurzió, a számok gépi ábrázolása és az ebből eredő alul-, illetve felülcsordulási gondok, valamint a véges pontosság jelentése.

Konkrét megoldásokkal találkozhatunk a lebegőpontos számok bináris ábrázolására, egy dinamikus tömbosztály megvalósítására. Fény derül a nyelv egyik alapeleme, a pointerek belső működésére és a velük kapcsolatos veszélyekre. Betekintést nyerhetünk a függvényhívások rejtelmeibe, világossá válnak a rekurzív függvényhívásokban rejlő kockázatok, nyilvánvalóvá az okok. A könyv külön figyelmet fordít az objektumorientáltsággal felmerülő fogalmak tisztázására, az osztályok, struktúrák, virtuális függvények helyes használatára és optimális kiaknázására. Fontosságának megfelelő helyet kap a kivételkezelés. Természetesen nem maradhat el a nyelv legújabb vívmánya, az STL - a konténerek, iterátorok és algoritmusok - bemutatása sem."

A könyvről további részleteket a szerző weboldalán Olvashatunk: http://nkari.uw.hu