

Python interfész kialakítása C++-alapú függvénykönyvtárhoz Boost.Python segítségével

A tudományos életben, valamint az alkalmazott kutatás területén is egyre fontosabb az ún. time-to-market, azaz a piacra kerülési idő, ebből kifolyólag pedig egyre nagyobb jelentősége van a gyors prototípustervezésre alkalmas programnyelveknek, mint amilyen a Guido van Rossum által jegyzett Python is. A rendelkezésre álló nagy mennyiségű, alacsonyabb szintű programnyelveken megírt könyvtárak portolása azonban óriási feladat lenne. Ráadásul az olyan, nagyfokú flexibilitást jelentő nyelvek, mint például amilyen a Python programozási nyelv is, teljesítményproblémáktól szenvednek az alacsonyabb szintű, általában C/C++ felhasználásával megírt programok ellenében.

Ebből kifolyólag számos olyan keretrendszer látott napvilágot, melyek lehetővé teszik, hogy a hatékony implementáció (például C++ felhasználásával) köré olyan programrendszer (ún. wrapper) kerüljön kialakításra, amelynek segítségével Python alól lehetővé válik C++-függvények végrehajtása, objektumok manipulálása.

Az alábbi cikk célja bemutatni és összehasonlítani a programtervező mérnökök számára rendelkezésre álló lehetőségeket, valamint egy, a magdeburgi Fraunhofer Institut für Fabrikbetrieb und Automatisierung által fejlesztett képfeldolgozási függvény könyvtárhoz elkészített konvertermodul implementációján keresztül kitérni a választott interfész sajátosságaira.

Hangsúlyozandó, hogy jelen cikk elsődlegesen azt a célt tűzte ki, hogy a Boost.Python függvénykönyvtár használatba vételét megkönnyítse azáltal, hogy az implementáció során tapasztalt speciális eseteket elemzi, ennek segítségével pedig hozzájárul a hivatalos dokumentációhoz segítve ezzel a Boost.Python könyvtárat alkalmazó közösség munkáját.

Python és C++ - összehasonlítás

A vázolt feladat megoldásához mérlegelni kell a kiindulási, illetve a célnyelv sajátosságait, ugyanis ezek nagy mértékben befolyásolják a kiválasztandó technológiát a két nyelv összekapcsolására.

	C++	Python
Nyelv típusa	Fordított (compiler)	Interpretált (interpreter)
Típusok kezelése	Erősen tipizált	Gyengén tipizált
Adattípus-hozzárendelés	Fordítási idő	Futási idő
Objektumok attribútumai	Fordítási időben meghatározott	Futási időben bővíthető
Adatreprezentáció a felhasználó felé	Egységbe zárás, csak a szükséges attribútumok érhetőek el	Minden (elvileg) elérhető
Argumentumok átadása	Érték, pointer, referencia	Referencia (érték szerint)
Függvények visszatérési értéke	0 vagy 1	0 vagy több
Objektumok nyilvántartása	Smart pointerek esetét leszámítva manuális memóriafelszabadítás szükséges dinamikusan allokkált objektumok esetében.	Automatikus referencia számláló és memóriafelszabadítás
Objektum másolása (alapértelmezett esetben)	Másoló konstruktor (copy constructor)	Referencia számláló növelése (ún. shallow copy)
Modularitás	Névterek	Modulok
Numerikus adattípusok	Natív	Objektum

1. Táblázat: A C++ és Python programnyelvek főbb különbségeinek összehasonlítása

Az 1. táblázat tartalmaz számos, igen fontos különbséget a két programnyelv között - Pythonból a 3-as verzió képezi a vizsgálat tárgyát, külön megjegyzés hiányában a továbbiakban is minden megállapítás

erre a verzióra vonatkozik. Az említett Python főverzió részletes dokumentációja megtalálható van Rossum és Drake összefoglaló munkájában (2011).

C++ tekintetében pedig a nyelv megalkotójának, Stroustrupnak a munkája (2013) képezi az összehasonlításhoz alapul vett programverziót.

A legfontosabb a különbségek közül az, hogy míg C++ esetében fordított, addig Python tekintve interpretált nyelvről beszélünk (a teljesen korrekt megfogalmazás a következő: a lefordított Python bytekódot interpretálja az interpreter), ez azonban szerencsére a két nyelv közötti átjárhatóságot egyáltalán nem észrevehető módon befolyásolja, a rendelkezésre álló megoldások ezt a háttérben kezelik.

Kritikus szempont azonban az objektumok, illetve azok kezelésének témaköre, kiindulva akárcsak abból a tényből, hogy a két programnyelv alapvetően máshogy viszonyul az adattípusokhoz, ez pedig a C++-interfész-Python hármasság esetében a változók, objektumok konverzióját problémássá teszi, vagyis valamilyen módon szükség lesz majd ennek explicit megjelölésére.

A típusok kezelésének flexibilitása abban is jelentkezik, hogy a C++-ban ismert, rendkívül praktikus virtuális függvények (Stroustrup, 2013, 67-68) alapvetően nem léteznek Pythonban. Ugyanis a változók által létesített referens viszony, vagyis annak ténye, hogy a változó neve lényegében egy, az *object* osztályból leszármazott osztály példányára mutató referenciát tárol, az allokált memória felszabadítása pedig akkor történik meg, amikor a referenciaszámláló értéke eléri a nullát, (van Rossum-Drake, 2011, p.24) társítva a dinamikus típushozzárendeléssel, a virtuális függvényeket lényegükénél fogva értelmetlen konstrukciókká teszik (Pythonban nem létezik heterogén kollekció), vagyis az interfésznek különösen fontos ezen funkcionalitás robusztus kezelése.

A választandó interfésznek továbbá alkalmasnak kell lennie arra, hogy az objektumok által elfoglalt memóriaterületet felszabadítsa, figyelembe véve annak tényét, hogy a C++-ban megszokott referencia, ill. pointer formájában kezelt objektumok Python esetében alapvetően ismeretlen konstrukciók. Ismeretlenek, de csak a felhasználó számára, például a legnépszerűbb Python implementáció – Cpython, további implementációkért lásd (van Rossum-Drake, 2011, p.3)-, ill. a Python nyelv C API-ja esetében is gyakran találkozni *PyObject* pointerekkel (Python Software Foundation, 2001b) .

Keretrendszerek C++ és Python összekapcsolására

Már a hivatalos Python dokumentáció is számtalan lehetőséget kínál számunkra arra az esetre, amennyiben más programnyelven írt kódot kívánunk felhasználni Python programunkban (Python Software Foundation, 2001a).

Mielőtt ezek közül a fontosabbakra részletesebben kitérnénk, érdemes még egyszer kiemelni a korábban már megemlített tényt, vagyis hogy létezik a Python nyelvnek egy C nyelven keresztül elérhető API-ja (*Application Programming Interface, alkalmazás programozási interfész*), melyen keresztül lehetőségünk adódik alacsony szintű modulok írására Pythonhoz.

Ennek azonban éppen az alacsony szintű konstrukciója a nagy hátránya, ugyanis szinte mindenről a modul fejlesztőjének kell gondoskodnia, beleértve az objektumok referenciaszámlálójának növelését, ill. csökkentését is, ami jelentősen komplexebbé, s így nehezebben karbantarthatóvá teszi a programot.

Tehát olyan keretrendszerre, interfészre van szükségünk, ami az alapszintű, mechanikusan elvégzendő, ill. elvégezhető műveleteket a lehető legnagyobb mértékben automatizálja.

Az egyik ilyen lehetőség a Beazley cikkében (2003) bemutatott SWIG (*Simple Wrapper Interface Generator*), ami egy kódgenerátor segítségével lehetővé teszi a szükséges átalakításokat, ráadásul más szkriptnyelveket is támogat (Perl, Ruby, Tcl). Kódgenerátor lévén a portolni kívánt kód változásait automatikusan tükrözi a generált fájl, azonban az esetlegesen (a generátor számára) nem egyértelmű esetek bonyolult felismerhető hibajelenségekhez vezethetnek. Ugyanis a kódgenerátorok általában komplex szabályrendszer segítségével analizálják a programkódot (amihez egy ún. parszerre is szükség van). További probléma a SWIG-gel kapcsolatban, hogy a C++ és a Python réteg közé további temporális rétegek kerülnek generálásra, ami a futási idő számára jelent kedvezőtlen híreket.

Másik lehetőség a Boost könyvtárcsomag Python moduljának használata, ami teljesen más megközelítésmódot alkalmaz. A Boost.Python modul lehetőségeit remekül mutatja be Grosse-Kunstleve cikke (2003), teszi mindezt pedig számos példán keresztül. A Boost.Python könyvtár ugyanis egy interfészkönyvtár, ami többek között a felhasználó számára a következőket jelenti: minden esetleges változást a C++ interfészben manuálisan követni kell a Boost modul kódjában. Ez természetesen a már korábban említett time-to-market mérlegelésénél időbeli veszteségeket jelent.

Azonban azt is figyelembe kell vennünk, hogy mivel a leképezést explicit módon meg kell adnunk, ez nagy mértékben hozzájárul ahhoz, hogy már a fordítás során hibaüzenetet generáljon a hibásan megadott wrapper (amely természetesebben jóval könnyebben kezelhető, mint a futási idejű hiba). Azonban még fontosabb, hogy ez az explicit írásmód azt is jelenti, hogy nem egyértelmű esetek előfordulásakor a programfejlesztő kezében van a döntés, nem áll fenn annak a veszélye, hogy egy automatizmus rossz döntést hozzon egy fel nem ismert speciális szituáció tekintetében.

	Kódgenerátor	Interfészkönyvtár
C++ interfész változása	Automatikusan követi a generált kód	Manuális beavatkozást igényel
Hibakeresés	A kódgenerátor belső állapota kifelé nem ismert (fekete doboz), nehezebb	A végeredmény C++ kód, hagyományos módon
Kód	Kevesebb	Több
Tesztelés	Több	Kevesebb

2. Táblázat: A kódgenerátor és az interfészkönyvtár elvi összehasonlítása Jim Bosch (2011) alapján

Az interfészkönyvtár alapú megoldás, ez esetben a Boost.Python mellett szólt továbbá a Boost könyvtárak mögött álló óriási közösség, illetve annak ténye, hogy számos megoldás került be a C++ standardba az eredetileg Boost könyvtárakban megtalálható megoldások közül.

Az ismertetett két megoldás mindegyike rendelkezik a maga előnyeivel, míg a kódgenerátor alapú megoldások maga az implementáció során csökkentik a szükséges kód mennyiségét, addig az interfészkönyvtár alapú megoldások talán legnagyobb előnye a szokásos hibakeresési módszerek biztosítása a felhasználó számára.

Mindkét megoldás csak részleteit tekintve alkalmaz automatizmusokat a feladat végrehajtására, ezért is érdekes az AutoWIG könyvtár megjelenése (az eszköz Fernique és Pradal 2017-es cikkében került bemutatásra), amely Boost.Python alapokon ígér teljesen automatizált kódportolást Python alá C++ függvénykönyvtárak számára. Sajnálatos módon azonban a rendelkezésre álló, felettébb hiányos dokumentáció nem teszi lehetővé az eszköz ipari környezetben történő alkalmazását.

Egyre nagyobb népszerűségnek örvendenek az olyan megoldások is, mint a Numba (Lam et al., 2015), mely egy úgynevezett típus-specializáló *Just-in-Time* fordító függvények számára. Vagyis arról van szó, hogy adott adattípussal történő függvényhívás esetében az adott adattípusra létrehozza a függvény specializált változatát. Ebből kifolyólag pedig lehetőség van különböző optimalizálási stratégiák kihasználására. A Numba alapvetően a számításintenzív függvények esetében képes jelentős gyorsulást eredményezni. A Numba-hoz hasonló megoldások előnye, hogy mindösszesen specifikus dekorátorokra van szükség – hangsúlyozandó, hogy a kód megírása teljes mértékben Pythonban történik. Ezáltal pedig a külön fejlesztett és karbantartott gyors implementációt biztosító C/C++, valamint a gyorsabb fejlesztést lehetővé tevő Python kód-adatbázisok egyesíthetők, kis mértékű teljesítményvesztés mellett. Természetesen az ilyen jellegű megoldások alkalmazását már az adott könyvtár tervezési fázisában figyelembe kell venni.

Boost.Python

Ebben a szakaszban röviden összefoglalásra kerül a Boost.Python interfészkönyvtár mögött rejlő tervezési paradigma, ebben elsősorban a Boost könyvtárak honlapján található, a Boost.Python modult ismertető leírást követem (Abrahams-Seefeld, 2002).

A Boost.Python könyvtár alapvetően speciális preprocesszor makrók, illetve kulcsszavak segítségével teszi lehetővé az interfész létrehozását a C++-kód és a Python programnyelv között.

Ezen eszközök segítségével válik lehetővé viszonylag egyszerűen és rendkívül strukturáltan az interoperabilitás biztosítása, mégpedig úgy, hogy a Boost.Python könyvtár lényegében a C++-kódot a Python C API-nak megfelelően teszi C nyelven elérhetővé.

Vagyis többek között például a minden Boost.Python wrapper részét képező `BOOST_PYTHON_MODULE(name)` makró az interfész kód köré egy *extern "C"* blokkot hoz létre, aminek hatására nem a C++-fordító által használt konvenció kerül alkalmazásra a fordítás során generált szimbólumok esetében (*name mangling/decoration*, *névdekoráció*), hanem annak C-konform változata, így lehetővé téve C++-függvények hívását a Python C API-n keresztül.

Természetesen ez csak egy kis, ámár rendkívüli fontosságú része az interfész létrehozásának. Ezen felül többek között arról kell gondoskodnia az interfészkönyvtárnak, hogy a C számára ismeretlen, elsősorban a C++-nyelv objektumorientált felépítéséből eredő konstrukciók is elérhetővé váljanak az interfész segítségével.

Ezt pedig a Boost.Python könyvtár a felhasználó számára felettebb követhető és az esetek döntő többségében nem intruzív módon valósítja meg, vagyis a wrapper elkészítése nem igényli a portolni kívánt forráskód közvetlen módosítását. Jelentősebb kivételt jelentenek a virtuális, illetve tisztán virtuális függvények, ahol bár az eredeti kód továbbra is változatlan marad, azonban az interfész elkészítéséhez szükséges wrapper-függvények manuális implementációja, vagyis ebben az esetben egy szinttel mélyebb programstruktúrát kapunk.

Kiegészítések a hivatalos Boost.Python dokumentációhoz

A következőkben összefoglalásra kerülnek a Boost.Python könyvtár alkalmazása során felmerülő, kevésbé vagy egyáltalán nem dokumentált korlátozások, illetőleg hasonló okokból egyes, speciális C++-konstrukciók esetében a megfelelő, a sikeres portolást lehetővé tevő szemantikai, illetőleg szintaktikai struktúrák.

Move szemantika

A C++11-ben bemutatott move szemantika (Stroustrup, 2013, p.514-517) lehetővé teszi objektumok tulajdonlásának (*ownership*) átadását másolat létrehozása nélkül. Az ezt lehetővé tevő jobbérték-referencia (*T&&*) kifejezetten hasznos konstrukció temporális objektumok, vagy a mindösszesen egy tulajdonost (*owner*) engedélyező *unique_ptr* esetében.

Azonban a Boost.Python könyvtár jelenleg nem támogatja a move szemantikát, az erre irányuló törekvések egyelőre eredménytelennek bizonyultak – más megoldások, például a SWIG esetében ez a funkció már elérhető (SWIG-3.0 Documentation, 2018, p.102-103).

Pointerváltozók egyszerű adattípusokra

Habár ez esetben nem kifejezetten Boost.Python specifikus a probléma, azonban a fordító által generált hibaüzenet mindenképpen analízisre érdemes.

Amennyiben ugyanis egyszerű adattípusra (*POD*, *plain old data*) mutató pointer egy portolni kívánt függvényünk visszatérési értéke, abban az esetben *specify_a_return_value_policy_to_wrap_functions_returning<T*>* hibát jelez a fordító. Ennek az oka abban rejlik, hogy bár Pythonban az egyszerű adattípusok is az *object* osztály leszármazottjai (vagyis objektumok), a Python filozófia szerint nem megváltoztatható (*immutable*) objektumnak minősülnek (van Rossum-Drake, 2011, p. 17.), tehát az egyes példányokra mutató pointerváltozók nem definiáltak.

Template osztályok portolása

Mivel a template-paradigma a Python nyelv számára annak dinamikus tipizált voltából adódóan ismeretlen, ezért template osztályok, illetőleg függvények esetében példányosítani kell az adott template-konstrukciót, praktikusán a portolást elvégző C++projekthez tartozó megfelelő header-fájlban. Megjegyzendő, hogy a SWIG is hasonlóképpen oldja meg a template osztályok portolását, abban az esetben – mivel más paradigmáról van szó – egy speciális kulcsszó (*%template*) segítségével történik

gyakorlatilag a definíció lemásolása. Ennek során a megadott adattípus kerül behelyettesítésre a template típus helyére – vagyis lényegében a %template kulcsszó az adattípus behelyettesítésének manuális átírását automatizálja.

```
/*x.h*/
template <typename T>
class X
{
public:
    X(T value) : m_x(value){}
    T m_x;
}

/*x_to_python.h*/
#include "x.h"
template class X<int>;
```

1. Ábra: A szükséges példányosítás template-konstrukciók portolása esetén

Függvénynevek túlterhelése statikus esetben

Interfészkönyvtár voltából adódóan a Boost.Python által biztosított API megköveteli a leképezések explicit megadását, kiváltképpen a kérdéses esetekben.

Ebből kifolyólag túlterhelt függvények regisztrálásakor az adott függvény teljes aláírását specifikálni kell. Statikus függvények esetében ez annyiban módosul, hogy az aláírás megadásakor a függvénypointernél nem kell megadni az osztály névterét, annak ellenére, hogy a tagfüggvény címét hozzárendelő utasítás esetében igen.

Amennyiben összevetjük az alábbiakban részletezett megoldást más rendszerekkel, akkor például SWIG esetében azt tapasztalhatjuk, hogy mivel a wrapper kód generálása automatikusan történik, a dokumentáció szerint statikus függvények esetében a generált függvény pusztán meghívja azt (SWIG-3.0 Documentation, 2018, p.66.). Vagyis az automatizálásnak köszönhetően a jelen szakaszban vázolt probléma nem fordul alapvetően elő, azonban ez azzal a kompromisszummal jár, hogy bizonyos feltételek mellett a kódgenerálás sikertelen (SWIG-3.0 Documentation, 2018, p.75-80.).

```
/* method */
.def<void(class_namespace::*)(int)>("method", &class_namespace::method)
.def<void(class_namespace::*)(int, int)>("method", &class_namespace::method)

/* method - static */
.def<void(*)>("method", &class_namespace::method)
.def<void(*)>("method", &class_namespace::method)
.staticmethod("method")
```

2. Ábra: Statikus és nem statikus tagfüggvények regisztrálása közötti különbségek bemutatása

Argumentumok megadása virtuális függvények esetében

Virtuális függvények esetében a korábban említett módon wrapper implementációja szükséges, amihez a Boost.Python könyvtár alapvetően két, némileg különböző szintaxissal rendelkező megoldást kínál. Ennek oka alapvetően a régebbi verziójú Microsoft Visual Studio fordítójában keresendő.

Holott a használt Microsoft Visual Studio 2012 esetében már használható lenne az új szintaxis, a felhasználói táboron belül felmerülő, frissebb verziókat is érintő problémák miatt a régebbi szintaxis szerint történt a wrapperek megvalósítása.

Ebben az esetben pedig az argumentumok átadása kevésbé egyértelmű, ugyanis a Boost.Python virtuális függvény hívását végző nyelvi struktúrájának kell átadni a virtuális függvény paramétereit, holott az új szintaxis esetében ez közvetlenül a virtuális függvénynek történik.

```
/* virtual - new syntax */
void virtual_f_new_syntax(int x)
{
    this->get_override("virtual_f_new_syntax")(x);
}

/* virtual - old syntax */
void virtual_f_old_syntax(int x)
{
    return call<void>(this->get_override("virtual_f_old_syntax").ptr(), x);
}
```

3. Ábra: A Paraméterátadás különbségei az új és a régi Boost.Python szintaxis szerint

Összefoglalás

Jelen cikkben bemutatásra kerültek a jelenleg elérhető megoldások C++-alapú könyvtárak elérhetővé tételére Python programnyelven, valamint a Boost.Python interfészkönyvtár sajátosságai, illetve annak működésmódja. Az analízis részét képezte a C++ és a Python programnyelvek szisztematikus összehasonlítása, elsődlegesen a két nyelv különbségeire, mint a portolást megnehezítő tényezőkre kitérve.

Ezen korlátok külön tárgyalásra kerültek a Boost.Python könyvtár esetében, valamint egyes speciális szituációk hatékony kezelésére megoldási javaslatok kerültek megfogalmazásra, kiegészítéséként szolgálva ezzel a hivatalos dokumentációhoz, továbbá elősegítve a könyvtárat használó közösség tevékenységét.

A jelen cikkben található kódrészletek bővített, kipróbálást megkönnyítő változata szabadon elérhető az interneten (Reizinger, 2018).

Irodalomjegyzék

Abrahams D., Seefeld S. (2002) *Boost.Python*. [online] (Utoljára frissítve: 2018.08.01.) Elérhetőség: https://www.boost.org/doc/libs/1_68_0/libs/python/doc/html/index.html [Megtekintve: 2018.08.30.]

Beazley D.M. (2003). *Automated Scientific Software Scripting with SWIG*. Future Generation Computer Systems, 19(5), pp 599-609. Bosch J. (2011). *SWIG vs. Boost.Python*. [online] (Utoljára frissítve: 2013.10.08.) Elérhetőség: <https://dev.lsstcorp.org/trac/wiki/SwigVsBoostPython> [Megtekintve: 2018.08.30.]

Grosse-Kunstleve, R. W. (2003). *Building hybrid systems with Boost.Python*. C/C++ Users Journal, 21(7), pp. 29-36. - Report Number: LBNL-53142

Fernique, P. és Pradal C. (2017) *AutoWIG: Automatic Generation of Python Bindings for C++ Libraries*. arXiv:1705.11000v1 [cs.SE]

Lam et al. (2015). *Numba: A LLVM-based Python JIT-compiler*. Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, 7, pp. 1-6.

Reizinger, P. (2018) *boost_python.cpp*. [online] (Utoljára frissítve: 2018.11.11.) Elérhetőség: <https://gist.github.com/rpatrik96/cd7eba98aa3255b2fb20dfc1fd2172a3> [Megtekintve: 2018.11.11.]

Stroustrup, B. (2013). *The C++ Programming Language*. 4. Kiadás. Reading Mass., USA: Addison-Wesley

SWIG-3.0 Documentation. (2018). [ebook] p.653. Elérhető: <http://www.swig.org/Doc3.0/SWIGDocumentation.pdf> [Megtekintve: 30 Aug. 2018]

van Rossum, G. és Drake F.L. (2011) *The Python Language Reference Manual (version 3.2)*, UK: Network Theory Ltd.

Python Software Foundation (2001). *Extending and Embedding the Python Interpreter*. [online] (Utoljára frissítve: 2018.11.11.) Elérhetőség: <https://docs.python.org/3.6/extending/index.html> [Megtekintve: 2018.08.30.]

Python Software Foundation (2001). *Python/C API Reference Manual*. [online] (Utoljára frissítve: 2018.11.11.) Elérhetőség: <https://docs.python.org/3.6/c-api/index.html> [Megtekintve: 2018.08.30.]