C++ dlopen mini-HOGYAN

Aaron Isotton

aaron@isotton.com

C++ függvények és osztályok betöltése a `dlopen` API segítségével.

Tartalomjegyzék

1. Bevezető

UNIX C++ programozókban felmerülő gyakori kérdés, hogyan töltsenek be dinamikusan C++ függvényeket és osztályokat a `dlopen` használatával.

Tény, hogy nem minden esetben egyszerű ez, és némi magyarázatot igényel. Ez van leírva ebben a mini-HOGYANban.

Egy átlagos C és C++ programozási nyelv ismeret valamint a `dlopen` API ismerete szükséges ahhoz, hogy megérthesd ezt a dokumentumot.

Ez a HOGYAN elsődleges a http://www.isotton.com/howtos/C++-dlopen-mini-HOWTO/ webhelyen található meg.

1.1. Szerzői jog és licenc

1.2. A felelősség teljes elhárítása

A dokumentum tartalmáért nincs felelősségvállalás. Az elgondolásokat, példákat és információkat a saját felelősségedre használd. Előfordulhatnak hibák és pontatlanságok, amelyek a rendszered sérülését okozhatják. Minden óvatosság ellenére bármily hihetetlen, a szerző(k) semmilyen felelősséget nem vállal(nak).

Minden szerzői jog fenntartva az eredeti tulajdonosának, amennyiben másként nincs jelölve. A dokumentumban használt szakkifejezések semmilyen párhuzamot nem képviselnek védjegyekre, szervíz márkákra vonatkozólag. Egyedi alkotások vagy védjegyek nevesítése nem hozzájárulások.

1.3. Közreműködők

Örömmel mondok köszönetet az alábbi személyeknek (abc sorrendben):

- Joy Y Goodreau <`joyg (at) us.ibm.com`> a szerkesztésért.
- D. Stimitis <`stimitis (at) idcomm.com`> rámutatott néhány kérdésre a formázással és a név szétszedéssel kapcsolatban valamit az `extern "C"`-vel kapcsolatban.

1.4. Visszajelzés

Visszajelzést szívesen fogadok. A megjegyzéseidet, kritikádat és a hozzájárulásaidat a <`aaron@isotton.com`> címre küldheted.

1.5. A dokumentumban használt kifejezések

`dlopen` API

> A `dlclose`, `dlerror`, `dlopen` és `dlsym` függvények, amik leírása a `dlopen(3)` kézikönyv oldalon található.
>
> Megjegyezzük, hogy mi a "`dlopen`" kifejezést a `dlopen` *függvényre* magára, és a "`dlopen` API" kifejezést az *egész API*-ra használjuk.

1.6. Magyar fordítás

A magyar fordítást Szalai Ferenc (mailto: szferi[kukac]einstein.ki.iif[pont]hu) készítette (2004.04.17). A lektorálást Daczi László (mailto:dacas@freemail.hu_NO_SPAM) végezte el (2004.05.04). Utoljára javítva 2004.05.05.-én (r2). A dokumentum legfrissebb változata megtalálható a Magyar Linux Dokumentációs Projekt (http://tldp.fsf.hu/index.html) honlapján.

2. A probléma

Néha futásidőben kellene betölteni programkönyvtárakat (és használni a függvényeiket). Ez leginkább akkor szükséges, ha valamilyen plug-in vagy modul architektúrájú programot írsz.

A C nyelvben a program könyvtárak betöltése igen egyszerű (`dlopen`, `dlsym` és `dlclose` meghívása elegendő). C++-al ez egy kicsit bonyolultabb. A C++ program könyvtárak betöltésének nehézséget részint a „nevek szétszedése", részben pedig az a tény okozza, hogy a `dlopen` API C-ben lett írva, így nem teszi lehetővé osztályok egyszerű betöltését.

Mielőtt bemutatnánk a programkönyvtárak betöltését, C++-ban megvizsgáljuk a „név szétszedési" problémát egy kicsit alaposabban. Azt ajánlom akkor is olvasd el ezt a részt, ha nem érdekel, mert segít megérteni mi is a probléma és mi a megoldása.

2.1. „Név szétszedése"

Minden C++ programban (vagy programkönyvtárban vagy tárgykód állományban) minden nem statikus függvény a bináris állományban *szimbólumokkal* van reprezentálva. Ezek a szimbólumok speciális karaktersorozatok, amik egyértelműen azonosítják a függvényt a programban, programkönyvtárban vagy tárgykód állományban.

C-ben a szimbólum nevek megegyeznek a függvények neveivel: az `strcpy` függvény szimbóluma `strcpy` és így tovább. Ez azért lehetséges, mert C-ben két nem statikus függvénynek nem lehet azonos a neve.

Mivel a C++ engedélyezi az átdefiniálást (overloading - különböző függvények azonos névvel, de különböző argumentumokkal), valamint számos új tulajdonsága van, ami a C-nek nincs — mint osztályok, tagfüggvények, kivétel kezelés — ezért nem lehetséges a függvények nevét egyszerűen szimbólumnévnek használni. A C++ ezt az problémát az úgynevezett *„név szétszedéssel"* (mangling) oldja meg. Ez úgy működik, hogy a a függvények és egyéb szükséges információk (mint az argumentumok száma és mérete) alapján létrehoz egy csak a fordító számára értelmes karaktersorozatot, amit az szimbólum névnek tud használni. A `foo` függvény ilyen módon előállított neve így nézhet ki például: `foo@4%6^`. Vagy nem is feltétlen kell tartalmaznia a "foo" szót magát.

Az egyik probléma ezzel az eljárással az, hogy a C++ standard (jelenleg [ISO14882]) nem definiálja ennek a menetét. Így minden fordító a saját módszerét használja. Néhány fordító meg is változtatja az algoritmust verzióról verzióra (különösen a g++ 2.x és 3.x között). Ezért ha ki is találtad, hogy a te fordítód hogyan is működik e tekintetben (és így be fogod tudni tölteni a függvényeidet a `dlsym` segítségével) ez valószínűleg csak a te fordítóddal fog működni és használhatatlan lesz annak következő verziójával.

2.2. Osztályok

A másik probléma a `dlopen` API-val, az, hogy csak *függvények* betöltését támogatja. Általában azonban egy C++ programkönyvtárban egy osztályt publikálsz, amit a programodban használni szeretnél. Ezen osztály használatához egy példányt kell belőle készítened, de ez nem is olyan könnyű.

3. A megoldás

3.1. `extern "C"`

A C++-nak van egy speciális kulcsszava arra, hogy függvényeket C kötéssel definiáljuk. Ez az `extern "C"`. Az a függvény ami `extern "C"`-ként lett definiálva annak függvényneve szimbólumként használható akárcsak egy C függvénynek. Ezért csak nem-tagfüggvények deklarálhatók `extern "C"` segítségével, és ezeket nem lehet átdefiniálni.

Habár van néhány megkötés az `extern "C"` függvényekre, mégis igen hasznosak, mivel dinamikusan betölthetőek a `dlopen` segítségével akárcsak a C függvények.

Ez *nem* jelenti azt, hogy az `extern "C"`-vel definiált függvények nem tartalmazhatnak C++ kódot. Az ilyen függvények teljes értékű C++ függvények, kihasználhatják a C++ lehetőségeit és bármilyen típusú argumentummal rendelkezhetnek.

3.2. Függvények betöltése

C++ a függvények úgy tölthetőek be mint C-ben; a `dlsym` segítségével. A betölteni kívánt függvényeket `extern "C"`-vel kell jelölnöd, hogy a C-szerű szimbólum névképzést kikényszerítsd.

Példa 1. Egy függvény betöltése

main.cpp:

```
#include <iostream>
#include <dlfcn.h>


int main() {
    using std::cout;
    using std::cerr;
```

```
    cout << "C++ dlopen demo\n\n";

    // open the library
    cout << "Opening hello.so...\n";
    void* handle = dlopen("./hello.so", RTLD_LAZY);

    if (!handle) {
        cerr << "Cannot open library: " << dlerror() << '\n';
        return 1;
    }

    // load the symbol
    cout << "Loading symbol hello...\n";
    typedef void (*hello_t)();
    hello_t hello = (hello_t) dlsym(handle, "hello");
    if (!hello) {
        cerr << "Cannot load symbol 'hello': " << dlerror() <<
            '\n';
        dlclose(handle);
        return 1;
    }

    // use it to do the calculation
    cout << "Calling hello...\n";
    hello();

    // close the library
    cout << "Closing library...\n";
    dlclose(handle);
}
```

hello.cpp:

```
#include <iostream>

extern "C" void hello() {
    std::cout << "hello" << '\n';
}
```

A `hello` függvény a `hello.cpp` állományban van definiálva, mint `extern "C"`. A `main.cpp`-ben töltődik be a `dlsym` hívással. A függvényt `extern "C"`-vel kell megjelölni, mert különben nem tudjuk biztosan a hozzá tartozó szimbólumnevet.

Figyelem

Két típusa létezik az `extern "C"` deklarációnak: `extern "C"` ahogy fent is használtuk, és `extern "C" { ... }` a deklaráció kapcsos zárójelek között. Az első (inline) forma egy deklaráció ami egyszerre extern és C nyelvű kiértékelést ír elő, míg a második csak a nyelvi előírást befolyásolja. Az alábbi két deklaráció ekvivalens:

```
extern "C" int foo;
extern "C" void bar();
```

és

```
extern "C" {
     extern int foo;
     extern void bar();
 }
```

Ahogy nincs különbség `extern` és a nem-`extern` *függvény* függvénydeklarációk között sem. Ez mindaddig nem jelent problémát amíg nem deklarálsz változókat. Ha *változókat* deklarálsz tartsd észben, hogy

```
extern "C" int foo;
```

és

```
extern "C" {
    int foo;
}
```

nem ugyanaz a dolog.

További részleteket találsz a [ISO14882], 7.5 fejezetében, különös tekintettel a 7. bekezdésre vagy a [STR2000], 9.2.4. paragrafusában.

Mielőtt bármi extra dolgot csinálnál az extern változókkal, ajánlott elolvasni a „További információ” fejezetben felsorolt dokumentumokat.

3.3. Osztályok betöltése

Az osztályok betöltése egy kicsit komplikáltabb, mert nekünk az osztály egy *példányára* van szükségünk, nem csak egy függvényre mutató mutatóra.

Nem tudjuk létrehozni az osztály egy példányát a `new` operátor segítségével, mert az osztály nincs definiálva a futtatható állományban, és mert nem tudjuk a nevét.

A megoldás a polimorfizmus segítségével adódik. Egy alap *interfész* osztályt definiálunk a *futtatható állományban* virtuális tagfüggvényekkel, és egy származtatott *implementációs* osztályt *a modulban*. Általában az interfész absztrakt osztály (egy osztály absztrakt, ha minden függvénye virtuális).

A dinamikus osztálybetöltést általában plug-in-okban használják — Ezeknek egy világosan definiált interfészt kell használniuk — Egy interfészt és az azt implementáló osztályokat kell definiálnunk.

Ezek után - még mindig a modulban - definiálunk két további segédfüggvényt (úgynevezett *class factory functions*). Az egyik függvény ezek közül elkészíti egy példányát az osztálynak, és egy arra irányított mutatót ad vissza. Míg a másik egy osztályra irányított mutatót kap (amit a factory készített) és felszabadítja azt. Ezt a két függvényt `extern "C"` direktívával jelöljük meg.

Ahhoz, hogy osztályt tölts be modulból csak a két factory függvényt kell betöltened a `dlsym` segítségével. Szerkeszteni (link) ugyanúgy kell, mint ahogy azt ebben részben tettük a hello függvénnyel. Ezek után már annyi példányt tudsz létrehozni és felszabadítani az osztályból, amennyit csak akarsz.

Példa 2. Egy osztály betöltése

Itt mi most egy általános `polygon` osztályt használunk, mint interfész és egy származtatott `triangle` osztályt, mint implementációt.

main.cpp:

```
#include "polygon.hpp"
#include <iostream>
#include <dlfcn.h>

int main() {
    using std::cout;
    using std::cerr;

    // load the triangle library
    void* triangle = dlopen("./triangle.so", RTLD_LAZY);
    if (!triangle) {
        cerr << "Cannot load library: " << dlerror() << '\n';
        return 1;
    }

    // load the symbols
    create_t* create_triangle = (create_t*) dlsym(triangle, "create");
    destroy_t* destroy_triangle = (destroy_t*) dlsym(triangle, "destroy");
    if (!create_triangle || !destroy_triangle) {
        cerr << "Cannot load symbols: " << dlerror() << '\n';
        return 1;
    }

    // create an instance of the class
    polygon* poly = create_triangle();

    // use the class
    poly->set_side_length(7);
        cout << "The area is: " << poly->area() << '\n';

    // destroy the class
    destroy_triangle(poly);

    // unload the triangle library
    dlclose(triangle);
}
```

polygon.hpp:

```
#ifndef POLYGON_HPP
#define POLYGON_HPP

class polygon {
protected:
    double side_length_;

public:
    polygon()
        : side_length_(0) {}

    void set_side_length(double side_length) {
        side_length_ = side_length;
    }

    virtual double area() const = 0;
};

// the types of the class factories
typedef polygon* create_t();
typedef void destroy_t(polygon*);

#endif
```

triangle.cpp:

```
#include "polygon.hpp"
#include <cmath>

class triangle : public polygon {
public:
    virtual double area() const {
        return side_length_ * side_length_ * sqrt(3) / 2;
    }
};


// the class factories

extern "C" polygon* create() {
    return new triangle;
}

extern "C" void destroy(polygon* p) {
    delete p;
}
```

Néhány dolgot meg kell jegyeznünk az osztályok betöltésével kapcsolatban:

- Az osztályt létrehozó és felszabadító függvényeket *meg kell írnod*. Soha *ne* szabadítsd fel a példányokat a `delete` operátorral a futtatható állományon belül. Mindig add vissza azokat a modulnak. Ez azért szükséges, mert a `new` és a `delete` C++ operátorok használata nem feltétlenül konzekvens. Ezért lehetséges, hogy egy pár nélküli `new`

vagy `delete` hívás az oka a memória-szivárgásnak vagy segmentation fault-nak. Ugyanez igaz akkor is, ha különböző standard programkönyvtárakat használsz a modulban és futtatható állományban.

- Az interfész osztály dekonstruktorának virtuálisnak kell lennie szinte minden esetben. *Lehetséges* egy meglehetősen ritka eset, amikor ez nem feltétlen szükséges. Ez a megkötés nem okoz problémát, mert az általa keletkező többletterhelés (overhead) elhanyagolható.

  Ha az alap osztályodnak nincs szükséges dekonstruktorra akkor is definiálj egy üreset (és `virtual`-t), különben előbb vagy utóbb *problémáid lesznek*. Ezt garantálom. Többet tudhatsz meg erről a problémáról a C++ FAQ lite (http://www.parashift.com/c++-faq-lite/) webhelyen található comp.lang.c++ GYIK 20. fejezetéből.

4. Gyakran Ismételt Kérdések

1. Windowst használok és nem találom a `dlfcn.h` header állományt a PC-men! Mi a probléma?

A probléma, mint mindig a Windows. Nincs `dlfcn.h` header Windows-on és nincs `dlopen` API sem. Van egy hasonló API a `LoadLibrary` függvénnyel. A legtöbb itt leírt dolog alkalmazható erre is. Továbbá használhatod a libltdl (a libtool része) programkönyvtárat, hogy “emuláld” a `dlopen`-t számos platformon.

Olvasd el a Programkönyvtár HOGYAN (http://tldp.fsf.hu/HOWTO/Program-Library-HOWTO-hu/index.html) (Program Library HOWTO (http://www.dwheeler.com/program-library)) 4. fejezetét (Dinamikusan betölthető (Dynamically Loaded; DL) programkönyvtárak (http://tldp.fsf.hu/HOWTO/Program-Library-HOWTO-hu/dl-libraries.html); Dynamically Loaded (DL) Libraries (http://www.dwheeler.com/program-library/Program-Library-HOWTO/dl-libraries.html)). Ez további információkkal szolgál olyan technikákról, amelyekkel platformfüggetlenül tölthetsz be programkönyvtárakat és készíthetsz osztályokat.

2. Létezik bármilyen `dlopen`-kompatibilis illesztőfelület a Windows `LoadLibrary` API-jához?

Nem tudok róla és nem hiszem, hogy valaha is lesz olyan, ami a `dlopen` összes lehetőségét támogatni fogja.

Vannak alternatív megoldások: libtltdl (a libtool része), ami a különböző dinamikus betöltő API-khoz nyújt egységes felületet, köztük a `dlopen` és a `LoadLibrary` API-khoz is. Egy másik lehetőség a Dynamic Loading of Modules (http://developer.gnome.org/doc/API/glib/glib-dynamic-loading-of-modules.html) (A GLib dinamikus modul betöltés). Használd ezeket a jobb platformfüggetlenség biztosítása érdekében. Én soha nem használtam őket, így nem tudom megmondani neked mennyire stabilak és hogyan működnek.

5. További információ

- A `dlopen(3)` kézikönyv oldalai. Ez kifejti a `dlopen` API célját és a használatát.

- A *Dynamic Class Loading for C++ on Linux* (http://www.linuxjournal.com/article.php?sid=3687) cikk James Norton tollából a Linux Journal (http://www.linuxjournal.com/)-on.
- A kedvenc C++ referenciád a `extern "C"`-ról, öröklődésről, virtuális függvényekről, `new` és `delete` operátorokról. A [STR2000] ajánlott.
- [ISO14882]
- A Programkönyvtár HOGYAN (http://tldp.fsf.hu/HOWTO/Program-Library-HOWTO-hu/index.html) (Program Library HOWTO (http://www.dwheeler.com/program-library)) mintent tartalmaz, amire valaha szükséged lesz statikus, megosztott és dinamikusan betölthető programkönyvtárakkal kapcsolatban. Melegen ajánlott.
- A Linux GCC HOWTO (http://tldp.org/HOWTO/GCC-HOWTO/index.html)-ból többet tudhatsz meg arról, hogyan készíthetsz programkönyvtárakat GCC-vel.

Irodalomjegyzék

ISO14482 *ISO/IEC 14482-1998 — The C++ Programming Language*. PDF és nyomtatott könyv formájában is elérhető a http://webstore.ansi.org/ webhelyen.

STR2000 Bjarne Stroustrup *The C++ Programming Language*, Special Edition. ISBN 0-201-70073-5. Addison-Wesley.