C++ dlopen mini NASIL

Yazan:
Aaron Isotton
<aaron (at) isotton.com>

Çeviren:
A. Kemal Ökmen
<kemal (at) comu.edu.tr>

22 Mart 2006

Özet

C++'da `dlopen` API kullanarak işlevler ve sınıflar çalışma anında nasıl yüklenir.

Konu Başlıkları

Bu çevirinin sürüm bilgileri:

1.0	Mart 2006	akö
İlk çeviri		

Özgün belgenin sürüm bilgileri:

1.10	2006-03-22	AI
Lisans GFDL'den GPL'e değiştirildi. dlerror kullanımı düzeltildi; Teşekkürler Carmelo Piccione. Örnekte sanal yıkıcı kullanımı; Teşekkürler Joerg Knobloch. Kaynak Kod bölümü eklendi. Ufak düzeltmeler.		
1.03	2003-08-12	AI
GLib Dynamic Module Loader referansı eklendi. Teşekkürler G. V. Sriraam.		
1.02	2002-12-08	AI
SSS eklendi. Küçük değişiklikler		
1.01	2002-06-30	AI
Sanal yıkıcı açıklaması güncellendi. Küçük değişiklikler.		
1.00	2002-06-19	AI
Telif ve Lisans bölümleri başa getirildi. Terimler bölümü eklendi. Küçük değişiklikler.		
0.97	2002-06-19	JYG
Küçük gramer ve cümle değişiklikleri girildi.		
0.96	2002-06-12	AI
Kaynakça eklendi. Extern işlev ve değişkenlerin açıklaması düzeltildi.		
0.95	2002-06-11	AI
Küçük geliştirmeler.		

Yasal Açıklamalar

1. Giriş

Unix C++ programcıları arasında `dlopen` Uygulama Programlama Arayüzü – UPA (API) kullanılarak çalışma anında işlev ve sınıf yüklemenin nasıl yapıldığı sorusu sıklıkla ortaya çıkar.

Aslında, herzaman kolay değildir ve bazı açıklamalara ihtiyaç duyulur. Bu mini NASIL belgesi bunu yapar.

Bu belgeyi anlamak için ortalama C, C++ ve `dlopen` API bilgisi gereklidir.

Bu NASIL belgesinin özgün sürümü `http://www.isotton.com/howtos/C++-dlopen-mini-HOWTO/` adresinde bulunmaktadır.

1.1. Teşekkürler/Katkıda Bulunanlar

Bu belgenin hazırlanmasında katkıları olanlara teşekkür ederim:

- Düzenlemeleri için Joy Y Goodreau <joyg (at) us.ibm.com>.
- Biçimleme ve isim cendereleme ile ilgili birkaç husus ve ayrıca `extern "C"` hakkında birkaç incelik için D. Stimitis <stimitis (at) idcomm.com>.

  Bu NASIL belgesinde hataları gösteren ve ipuçları veren isimsiz diğer kişiler. Siz kendinizi biliyorsunuz!

1.2. Geri Bildirim

Bu belge için geribildirim memnuniyetle karşılanır. Eklemelerinizi, yorumlarınızı ve eleştirilerinizi eposta ile gönderebilirsiniz: < aaron (at) isotton.com>.

1.3. Belgede Kullanılan Terimler

`dlopen` API
: `dlclose`, `dlerror`, `dlopen` ve `dlsym` işlevleri `dlopen(3)` man sayfasında açıklanmıştır.

  Tek bir `dlopen` *işlevi*nden bahsederken "`dlopen`" ve *tüm* uygulama geliştirme arayüzünden bahsederken "`dlopen` API" terimi kullanıldı.

isim cendereleme
: İngilizce *name mangling* teriminin Türkçe karşılığı olarak kullanıldı.

soyut sınıf
: İngilizce *abstract class* teriminin Türkçe karşılığı olarak kullanıldı. Bazı kaynaklarda *özet sınıf* olarak da geçer.

2. Sorun

Bazen bir kütüphaneyi (ve onun işlevlerini) çalışma zamanında yüklemek zorunda olabilirsiniz; bu çoğunlukla programınız için bir çeşit uyumlu–ek (plug–in) veya modül yazdığınızda ortaya çıkar.

C programlama dilinde, kütüphane yükleme çok sadedir (`dlopen`, `dlsym` ve `dlclose` işlevlerini çağırmak yeterli), C++ ile bu biraz karmaşıktır. C++ kütüphanelerini dinamik olarak yüklemenin zorluğu kısmen *isim cendereleme* (sayfa: 5) (name mangling) işleminden ve `dlopen` API'nin C ile yazılmasından ileri gelir. Bu yüzden C++ sınıflarını yüklemek için elverişli bir yöntem sunmaz.

C++'da sınıfların nasıl yüüklendiğini açıklamadan önce, dilerseniz isim cendereleme özelliğine bakarak sorunu analiz edelim. Sorunun nasıl oluştuğunu ve nasıl çözüleceğini anlamak için ilgilenmeseniz bile isim cendereleme işleminin açıklamasını okumanızı tavsiye ediyorum.

2.1. İsim Cendereleme

Her C++ programında (veya kütüphanesinde veya nesne dosyasında), statik olmayan tüm işlevler ikili dosya içinde *sembol*lerle temsil edilir. Bu semboller bir işlevin programda, kütüphanede veya nesne dosyasında tek ve diğerlerinden farklı olarak gösterildiği bir dizgedir.

C'de sembol isimleri ile işlev isimleri aynıdır: `strcpy`'nin sembolü de `strcpy` olacaktır, vb. C'de bu şekilde kullanım mümkündür çünkü statik olmayan aynı isimli iki işlev yoktur.

C++ aşırı yüklemeye (farklı işlevlerin değişik argümanlarla aynı isme sahip olması) izin verdiği için ve C'nin izin vermediği bir çok özelliği olduğu için –sınıflar, eleman işlevler, istisnai durum belirtimleri gibi– işlev isimlerini basitçe sembol ismi olarak kullanmak mümkün değildir. Bunu çözmek için C++ isim cendereleme denen, işlev isimlerinin ve tüm gerekli bilgilerin (argümanların sayısı ve boyutu gibi) sadece derleyicinin bildiği bazı garip görünümlü dizgelere dönüştürüldüğü yöntemi kullanır. Örneğin cenderelenen `foo` ismi `foo@4%6^` gibi gözükebilir. Hatta "foo" kelimesini bile içermeyebilir.

Cendereleme ile ilgili başka bir sorun da C++ standardının (şu anda [ISO14882]) isimlerin nasıl cendereleneceğini tanımlamamasıdır. Bu yüzden her derleyici cendereleme işlemini kendine göre yapar. Hatta bazı derleyicilerin farklı sürümleri arasında bile isim cendereleme algoritmaları değişiklik göstermektedir (özellikle g++2.x ve 3.x). Belirli bir derleyicinin isim cendereleme işlemini nasıl yaptığını çözseniz bile (ve böylece işlevleri `dlsym` ile yükleyebilirsiniz) bu muhtemelen sadece sizin derleyicinizde çalışacak ve bir sonraki sürüme uymayacaktır.

2.2. Sınıflar

`dlopen` API ile ilgili diğer bir sorun sadece yükleme *işlevlerini* desteklemesidir. Fakat C++ sıklıkla programınızda kullanmak isteyeceğiniz bir sınıf ortaya çıkarır. Açıktır ki bu sınıfı kullanmak için o sınıfın bir örneğini (nesne) oluşturmanız gerekir, fakat bu kolay bir şekilde yapılamaz.

3. Çözüm

3.1. `extern "C"`

C++, C bağlamaları (bindings) ile işlev bildirimi için özel bir anahtar kelimeye sahiptir: `extern "C"`. `extern "C"` olarak bildirilen bir işlev, C işlevinde olduğu gibi, işlev ismini sembol ismi olarak kullanır. Bu yüzden sadece üye olmayan işlevler `extern` "C" olarak bildirilebilir ve aşırı yüklenemez.

Pek çok kısıtlamanın olmasının yanında `extern` "C" işlevleri çok kullanışlıdır çünkü bir C işlevi gibi `dlopen` kullanarak çalışma anında yüklenebilir.

Bu, `extern` "C" olarak sınıflandırılan işlevlerin C++ kodu içermeyecekleri anlamına *gelmez*.

3.2. İşlevlerin Yüklenmesi

C++'da işlevler C'deki gibi `dlsym` ile yüklenir. Sembol isimlerinin cenderelenmesini engellemek için yüklemek istediğiniz işlevler `extern "C"` olarak sınıflandırılmalıdır.

Örnek 1. Bir işlevin yüklenmesi

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;
    }

    // sembolü yükle
    cout << "Loading symbol hello...\n";
    typedef void (*hello_t)();

    // hataları resetle
    dlerror();
    hello_t hello = (hello_t) dlsym(handle, "hello");
    const char *dlsym_error = dlerror();
    if (dlsym_error) {
        cerr << "Cannot load symbol 'hello': " << dlsym_error <<
            '\n';
        dlclose(handle);
        return 1;
    }

    // hesaplama yapmak için kullan
    cout << "Calling hello...\n";
    hello();

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

hello.cpp:

```
#include <iostream>

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

`hello` işlevi `hello.cpp` içinde `extern "C"` tanımlanmıştır; `main.cpp` içinde `dlsym`'in çağırılması ile yüklenmiştir. İşlev `extern "C"` olarak sınıflandırılmalıdır çünkü aksi taktirde sembol ismini bilemeyecektik.

Uyarı

`extern "C"` tanımlamanın iki değişik biçimi vardır: yukarıdaki gibi `extern "C"` kullanımı ve küme parantezleri arasında `extern "C" { ... }` biçiminde. İlki (satıriçi biçimi) harici ilintilemeli (extern linkage) ve C dili ilintilemeli bildirimdir; ikincisi sadece dil ilintilemesini etkiler. Bu yüzden aşağıdaki iki bildirim eşdeğerdir:

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

ve

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

`extern` ve `extern` olmayan *işlev* bildirimi arasında fark olmadığından değişken bildirimi yapmadığınız sürece sorun yoktur. *Değişken* tanımlarsanız şunu aklınızdan çıkarmayın

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

ile

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

aynı şey *değildir*.

Daha ileri açıklamalar için 7. bölüme özellikle dikkat ederek [ISO14882] 7.5 bölümüne veya [STR2000], 9.2.4'e bakınız.

Harici (extern) değişkenlerle daha ileri işlemler için *Ayrıca bakınız* (sayfa: 10) bölümündeki belgeleri inceleyiniz.

3.3. Sınıfların Yüklenmesi

Bir sınıfı yüklemek biraz daha karmaşıktır çünkü sadece bir işlev göstericisi değil, sınıfın bir *örneği* gereklidir.

`new` kullanarak bir sınıfın örneğini yaratamayız çünkü sınıf çalıştırılabilir kod içinde tanımlı değildir ve (bazı durumlarda) ismini bile bilmeyiz.

Çözüm çokbiçimlilik ile gerçekleştirilir. Bir baz –*çalıştırılabilir kod* içinde sanal üyeler ile *arayüz* sınıfı– ve bir türemiş –*modül* içinde *gerçekleştirim* sınıfı– tanımlarız. Genel olarak arayüz sınıfı soyuttur (abstract – bir sınıf eğer saf sanal işlevi varsa soyuttur).

Sınıfların dinamik yüklenmesi genellikle uyumlu–eklentiler (plug–in) için kullanıldığından –ki açıkça tanımlanmış arayüzleri ortaya çıkarması gerekir– bir arayüz ve türemiş gerçekleştirim sınıfı tanımlamak zorunda oluruz.

Sonra, hala modül içindeyken, sınıf üretim işlevleri (*class factory functions*) olarak bilinen iki ek yardımcı işlev tanımlarız. Bu işlevlerden bir tanesi sınıfın bir örneğini yaratır ve bunun göstericisini geri döndürür. Diğer işlev üretim tarafından oluşturulmuş bir sınıf göstericisi alır ve onu yokeder. Bu iki işlev `extern "C"` olarak sınıflandırılır.

Sınıfı modülden kullanmak için `dlsym` kullanarak *hello işlevini yüklediğimiz gibi* (sayfa: 5) iki üretim işlevini yükleriz; sonra, istediğimiz kadar örnek oluşturur ve yokederiz.

Örnek 2. Bir Sınıfın Yüklenmesi

Burada arayüz olarak çok bilinen `polygon` sınıfını ve gerçekleştirim olarak türemiş `triangle` sınıfını kullanıyoruz.

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;
    }

    // hataları resetle
    dlerror();

    // sembolleri yükle
    create_t* create_triangle = (create_t*) dlsym(triangle, "create");
    const char* dlsym_error = dlerror();
    if (dlsym_error) {
        cerr << "Cannot load symbol create: " << dlsym_error << '\n';
        return 1;
    }

    destroy_t* destroy_triangle = (destroy_t*) dlsym(triangle, "destroy");
    dlsym_error = dlerror();
    if (dlsym_error) {
        cerr << "Cannot load symbol destroy: " << dlsym_error << '\n';
        return 1;
    }

    // sınıfın bir örneğini yarat
    polygon* poly = create_triangle();

    // sınıfı kullan
    poly->set_side_length(7);
        cout << "The area is: " << poly->area() << '\n';

    // sınıfı yoket
    destroy_triangle(poly);

    // triangle kütüphanesini boşalt (unload)
    dlclose(triangle);
}
```

polygon.hpp:

```
#ifndef POLYGON_HPP
#define POLYGON_HPP

class polygon {
```

```
protected:
    double side_length_;

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

    virtual ~polygon() {}

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

    virtual double area() const = 0;
};

// sınıf üretiminin türleri
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;
    }
};


// sınıf üretimi

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

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

Sınıfları yüklerken dikkat edilmesi gereken bir kaç şey vardır:

- *Hem* oluşturma *hem de* yoketme işlevi yapmalısınız; örneği çalıştırılabilir kodun içinden `delete` kullanarak yoketmemeli, herzaman onu modüle geri göndermelisiniz. Bunun sebebi C++'da `new` ve `delete` operasyonlarının aşırı yüklü olabilmesidir; bu, bellek sızıntısı ve parçalama arızasına (segmentation fault) yolaçan eşleşmeyen `new` ve `delete` çağırımına sebep olabilecektir. Modül ve çalıştırılabilir kodun bağlanması için farklı standart kütüphaneleri kullanılırsa yine aynısı geçerlidir.
- Arayüz sınıfının yıkıcısı her durumda sanal olmalıdır. Bunun zorunlu olmadığı çok nadir durumlar olabilir, fakat risk almaya değmez, çünkü ek yük genellikle ihmal edilebilir.

  Eğer baz sınıfın hiçbir yıkıcıya ihtiyacı yoksa yine de boş (ve sanal) bir yıkıcı tanımlayın; aksi taktirde

er ya da geç sorunlarla karşılaşırsınız; bunu garanti ederim. Bu sorunlar hakkında daha fazla bilgi için comp.lang.c++ SSS http://www.parashift.com/c++-«faq-«lite/, bölüm 20'yi okuyabilirsiniz.

4. Kaynak Kod

Saklamak için bu nasıl belgesindeki tüm kaynak kodları buradan[B7] indirebilirsiniz.

5. Sıkça Sorulan Sorular

5.1. Windows kullanıyorum ve `dlfcn.h` başlık dosyasını bulamıyorum! Sorun nedir?

Sorun Windows'da `dlopen` API'nin olmamasıdır ve dolayısıyla `dlfcn.h` başlık dosyası yoktur. Benzer bir API olan `LoadLibrary` işlevi vardır ve burada yazılanların bir çoğu ona da uygulanabilir. Daha fazla bilgi için Microsoft Developer Network[B8] adresine başvurunuz.

5.2. Acaba Windows `LoadLibrary` API için `dlopen`–uyumlu bir çeşit sarmalayıcı (wrapper) var mıdır?

Olduğunu bilmiyorum ve tüm `dlopen` seçeneklerini destekleyen bir sarmalayıcının olacağını düşünmüyorum.

Bunun yanında alternatifler mevcut: çeşitli farklı dinamik yükleme API'leri taşıyan libtltdl (libtool'un bir parçası), diğerleri arasında `dlopen` ve `LoadLibrary`. Başka bir tanesi ise GLib'in Dinamik Modül Yükleme işlevselliği (Dynamic Module Loading functionality of GLib)[B9]. Daha iyi bir çapraz–platform uyumluluğu sağlamak için bunlardan birini kullanabilirsiniz. Bunlardan birini hiç kullanmadım, bu yüzden ne kadar kararlı oldukları hakkında veya gerçekten çalışıp çalışmadıkları hakkında bir şey söyleyemem.

Ayrıca platformdan bağımsız olarak kütüphane yükleme ve sınıf oluştuma hakkında daha teknik bilgiler için Program Kütüphanesi NASIL[B10] belgesinin "Dinamik Yüklü Kütüphaneler – Dynamically Loaded (DL) Libraries" başlıklı 4. bölümünü okumalısınız.

6. Ayrıca bakınız

- `dlopen(3)` man sayfası. `dlopen` API'nin amacını ve kullanımını açıklar.
- Linux Journal[B11]'da yayınlanan James Norton tarafından yazılan Dynamic Class Loading for C++ on Linux[B12] makalesi.
- `extern "C"`, kalıtım, sanal işlevler, `new` ve `delete` hakkında gözde C++ referans belgeniz. [STR2000]'i tavsiye ederim.
- [ISO14882]
- Statik, paylaşımlı ve dinamik yüklenen kütüphaneler ve bunların nasıl oluşturulduğu hakkında ihtiyacınız olan herşeyi bulabileceğiniz Program Library HOWTO[B13] belgesi. Bilhassa önerilir.
- GCC ile kütüphane yapımını öğrenmek için Linux GCC HOWTO[B14] belgesi.

A. Kaynakça

[ISO14482]
ISO/IEC 14482–1998 — The C++ Programming Language -- , http://webstore.ansi.org/ adresinde basılı kitap ve PDF olarak mevcut. --

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

Notlar

Belge içinde dipnotlar ve dış bağlantılar varsa, bunlarla ilgili bilgiler bulundukları sayfanın sonunda dipnot olarak verilmeyip, hepsi toplu olarak burada listelenmiş olacaktır.

(*B7*) ../indirin/examples.tar.gz

(*B8*) http://msdn.microsoft.com/

(*B9*) http://developer.gnome.org/doc/API/glib/glib-“dynamic-“loading-“of-“modules.html

(*B10*) http://www.dwheeler.com/program-“library

(*B11*) http://www.linuxjournal.com/

(*B12*) http://www.linuxjournal.com/article.php?sid=3687

(*B13*) http://www.dwheeler.com/program-“library

(*B14*) http://tldp.org/HOWTO/GCC-“HOWTO/index.html

Bu dosya (c–dlopen–mini–howto.pdf), belgenin XML biçiminin TEXLive ve belgeler-xsl paketlerindeki araçlar kullanılarak PDF biçimine dönüştürülmesiyle elde edilmiştir.

5 Şubat 2007