C++ dlopen mini HOWTO

Aaron Isotton

aaron@isotton.com

2006-03-16

Diario delle Revisioni

Revisione 1.10 2006-03-16 Revisionato da: AI
Cambio di licenza da GFDL a GPL. Corretta la spiegazione dell'uso di dlerror, al riguardo si ringrazia Carme
Revisione 1.03 2003-08-12 Revisionato da: AI
Inclusa menzione del Dynamic Module Loader GLib. Si ringrazia G. V. Sriraam per il suggerimento.
Revisione 1.02 2002-12-08 Revisionato da: AI
Aggiunta una FAQ. Altre modifiche minori.
Revisione 1.01 2002-06-30 Revisionato da: AI
Aggiornamento del materiale sui virtual destructor. Altre modifiche minori.
Revisione 1.00 2002-06-19 Revisionato da: AI
Spostato il copyright e la licenza all'inizio del documento. Aggiunta la sezione riguardante i termini impiegati
Revisione 0.97 2002-06-19 Revisionato da: JYG
Modifiche minori riguardanti la grammatica e la struttura delle frasi.
Revisione 0.96 2002-06-12 Revisionato da: AI
Aggiunta la bibliografia. Corretta la spiegazione delle funzioni e variabili extern.
Revisione 0.95 2002-06-11 Revisionato da: AI
Modifiche minori..

Come caricare dinamicamente funzioni e classi C++ per mezzo dell'interfaccia `dlopen`.
Traduzione a cura di Federico Lucifredi <lucifred@fas.harvard.edu>, Harvard University.

1. Introduzione

Una domanda spesso posta da programmatori C++ che sviluppano sotto Unix riguarda il come si proceda al caricamento di funzioni e classi C++ dinamicamente per mezzo dell'API `dlopen`.

In pratica, far ciò non è sempre semplice e qualche spiegazione in merito sembra dovuta - questo è l'obiettivo di questo documento.

Una conoscenza intermedia dei linguaggi C e C++ e dell'API `dlopen` sono necessari per poter comprendere appieno il documento.

La versione originale di questo documento può essere consultata online al seguente URL:

http://www.isotton.com/howtos/C++-dlopen-mini-HOWTO/.

1.1. Copyright e Licenza (in originale inglese)

1.2. Liberatoria

Non si può accettare responsabilità per i contenuti di questo documento. L'uso di concetti, casi di esempio e di informazione ivi contenuti è interamente a rischio dell'utente. Questo documento può contenere errori o informazioni inaccurate che possono causar danno al sistema. Si proceda con cautela, e nonostante tutto ciò sia estremamente improbabile, l'autore non si assume alcuna responsabilità.

Tutti i copyright sono mantenuti dai rispettivi proprietari, eccezion fatta per i casi in cui si indica diversamente. L'uso di un termine in questo documento non deve essere considerato come avente effetto sulla validità di qualsiasi marchio registrato. La menzione di particolari prodotti o marche non deve essere considerato come una indicazione di preferenza dell'autore per tal prodotto.

1.3. Ringraziamenti a Coloro Che Hanno Contribuito

In questo documento, l'autore ha il piacere di riconoscere il contributo ricevuto da (in ordine strettamente alfabetico) :

- Joy Y Goodreau `<joyg (at) us.ibm.com>` per il suo lavoro di revisione editoriale.
- D. Stimitis `<stimitis (at) idcomm.com>` per aver portato all'attenzione dell'autore alcune sottigliezze riguardanti il processo di decorazione dei simboli (name mangling) e della clausola `extern "C"`.
- Numerosi altri hanno indicato errori o dato suggerimenti per migliorare questo documento. Voi sapete chi siete!

1.4. Commenti

L'autore apprezza i commenti del lettore, che possono essere inviati al seguente indirizzo e-mail: `<aaron@isotton.com>`.

1.5. Termini Usati in Questo Documento

`dlopen` API

L'insieme delle funzioni `dlclose`, `dlerror`, `dlopen` e `dlsym` come descritte nella pagina `dlopen(3)` di man.

Si osservi l'uso del termine "`dlopen`" con riferimento individuale alla *funzione* `dlopen`, e dei termini "`dlopen` API" e interfaccia `dlopen` con riferimento all'*intera API*.

2. Il Problema

Talvolta uno sviluppatore può trovarsi nella posizione di dover caricare una libreria (ed usare le funzioni ivi contenute) a runtime; questo accade il più delle volte quando si sta sviluppando un plug-in di qualche tipo o un programma disegnato con una architettura modulare.

In C caricare una libreria è estremamente semplice (è sufficiente invocare le funzioni `dlopen`, `dlsym` e `dlclose`), mentre in C++ l'operazione è leggermente più complessa. Le difficoltà insite nel caricare dinamicamente una libreria C++ sono in parte dovute al processo di decorazione dei simboli di linking comunemente chiamato name mangling, ed in parte dovute al fatto che l'interfaccia di `dlopen` è stata implementata pensando in C e, conseguentemente, non offre una maniera appropriata di caricare classi.

Prima di illustrare come si caricano dinamicamente le librerie in C++, è appropriato analizzare il processo di name mangling in maggior dettaglio. Si continui la lettura, la spiegazione del processo di decorazione dei simboli è importante nel comprendere le ragioni del problema e come risolverle.

2.1. Name Mangling

In ogni programma C++ (o libreria, o object file), tutte le funzioni non dichiarate come static sono rappresentate nel file binario da *simboli*. Questi simboli sono speciali stringhe di testo che identificano unicamente una funzione nel programma, la libreria o l'object file.

In C il simbolo e il nome della funzione corrispondono: il simbolo della funzione `strcpy` è appunto `strcpy`, e così per ogni altro simbolo. Questo è possibile perché in C due funzioni non dichiarate come static *devono* necessariamente avere nomi distinti.

Siccome il linguaggio C++ permette l'overloading di una funzione (diverse versioni di una funzione avente il medesimo nome ma una differente lista di argomenti) ed include svariate funzionalità non presenti in C - come classi, funzioni che fanno parte di dette classi, la specifica di eccezioni - non è semplicemente possibile impiegare il solo nome della funzione come unico simbolo. Per risolvere questo problema, il C++ utilizza il cosiddetto meccanismo di *name mangling*, che trasforma il nome della funzione e tutte le informazioni necessarie (come il numero e la dimensione degli argomenti) in una stringa di testo così ingarbugliata che solo il compiler riesce a comprenderla. Il simbolo risultante per la funzione `foo` potrebbe ad esempio essere `foo@4%6^` - o potrebbe non includere nemmeno la parola "foo".

Uno dei problemi risultanti dal processo di name mangling è che lo standard C++ (currently [ISO14882]) non definisce come i simboli debbano essere codificati. Di conseguenza, ogni compiler "decora" i simboli in maniera diversa, alcuni compiler hanno addirittura cambiato l'algoritmo di name mangling tra diverse release (famoso il caso di g++ 2.x e 3.x). Anche se aveste studiato i dettagli del processo di name mangling del vostro compiler (e quindi sareste in grado di caricare funzioni per mezzo di `dlsym`), questa soluzione risulterebbe funzionare solamente con quel particolare compiler, ed il meccanismo potrebbe non funzionare già con la prossima versione del medesimo.

2.2. Classi

Un altro problema dell'interfaccia definita da `dlopen` è la limitazione inerente nel fatto che la libreria supporta solamente il caricamento di *funzioni*. Ma in C++ una libreria spesso espone la definizione di classi da utilizzare nei nostri programmi. Ovviamente, per poter utilizzare queste ultime si deve essere in grado di istanziare le medesime, ma questo non è facilmente fattibile.

3. La Soluzione

3.1. `extern "C"`

In C++, una keyword indica che una funzione deve impiegare la convenzione di linking (il termine comunemente usato è "binding" - N.d.T.) usata in C: `extern "C"`. Una funzione dichiarata `extern "C"` viene rappresentata da un simbolo coincidente con il nome della funzione, esattamente come in C. Per questa ragione, solo funzioni che non sono metodi di una classe possono essere dichiarate `extern "C"`, e non possono essere overloaded.

Nonostante le serie limitazioni, funzioni dichiarate `extern "C"` sono estremamente utili nel nostro scenario perché possono essere caricate dinamicamente utilizzando `dlopen` esattamente come una funzione C.

Questo *non* significa che una funzione dichiarata come `extern "C"` non può contenere codice C++. Tale funzione è una funzione C++ a tutti gli effetti, che può utilizzare le funzionalità del linguaggio C++ e ricevere qualunque tipo di argomento il programmatore desideri.

3.2. Caricare Funzioni

In C++ le funzioni vengono caricate *esattamente* come in C, con una chiamata a `dlsym`, ma le funzioni che saranno caricate devono essere qualificate con `extern "C"` per evitare che i simboli vengano decorati.

Esempio 1. Caricare una Funzione

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)();

    // reset errors
    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;
    }

    // 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';
}
```

La funzione `hello`, definita in `hello.cpp` come `extern "C"`, viene caricata in `main.cpp` con la chiamata a `dlsym`. La funzione deve essere dichiarata come `extern "C"` perché altrimenti non sapremmo come ricostruire il simbolo corrispondente al suo nome.

Avvertimento

Esistono due versioni differenti della dichiarazione `extern "C"`: la summenzionata versione `extern "C"`, e la corrispondente `extern "C" { ... }` con le dichiarazioni in parentesi graffe. La prima versione, detta inline, indica linking esterno secondo la convenzione C, mentre la seconda versione ha effetto solo sul linking del linguaggio. Le due dichiarazioni seguenti sono quindi equivalenti:

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

e

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

Dato che non c'è differenza tra una funzione dichiarata con `extern` ed una dichiarata senza, la differenza tra i due stili non è rilevante, purché non si stiano dichiarando variabili. Se si dichiarano delle *variabili*, non dimenticare che

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

e

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

non sono la stessa cosa.

Per ulteriori chiarimenti, si faccia riferimento a [ISO14882] 7.5, prestando particolare attenzione al paragrafo 7, oppure si consulti [STR2000] al paragrafo 9.2.4.

Prima di far cose troppo fantasiose con variabili extern, dare una scorsa ai documenti elencati nella sezione approfondimento.

3.3. Caricare Classi

Caricare classi è un compito più complesso perché non ci serve solo un puntatore a funzione, ci serve un'*istanza* della classe.

Non possiamo istanziare la classe utilizzando `new` perché la classe non è definita nell'eseguibile, e perché, in certe circostanze, non ne conosciamo nemmeno il nome.

La soluzione è nell'usare le proprietà del polimorfismo: definiamo una classe *interfaccia* di base con metodi dichiarati virtuali nell'eseguibile e una classe di *implementazione* derivata da questa nel modulo che vogliamo caricare. Solitamente la classe che dichiara l'interfaccia è astratta (una classe è astratta se tutti i suoi metodi sono dichiarati virtuali).

Siccome il caricamento dinamico di classi viene generalmente usato per la creazione di plug-in - che devono definire una interfaccia chiaramente definita - si sarebbe dovuto definire una classe interfaccia ed una classe di implementazione in ogni caso.

A questo punto, si devono includere nel modulo altre due funzioni che ci assistano nel caricamento, dette funzioni *class factory* (letteralmente "fabbrica di classi" - N.d.T.). La prima di queste funzioni crea un istanza della classe e ritorna un puntatore a tale istanza, mentre l'altra si occupa di distruggere le classi che le vengono passate. Queste due funzioni sono dichiarate `extern "C"`.

Per utilizzare la classe definita nel modulo, si carichino le due funzioni factory per mezzo di `dlsym` esattamente come si è già illustrato nel precedente esempio. Con questo meccanismo, si possono istanziare e distruggere tante istanze della classe quante si desideri avere a disposizione.

Esempio 2. Caricare una Classe

In questo esempio una generica classe `polygon` dichiara l'interfaccia che la classe derivata `triangle` a sua volta implementa.

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;
    }

    // reset errors
    dlerror();

    // load the symbols
```

```
    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;
    }

    // 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) {}

    virtual ~polygon() {}

    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;
}
```

Alcune osservazioni sul processo di caricamento di classi:

- È necessario definire sia una funzione factory per creare istanze della classe che per distruggerle: si deve evitare l'uso dell'operatore `delete` nell'eseguibile e *sempre* affidarsi al modulo per distruggere la classe. Questo è dovuto al fatto che in C++ gli operatori `new` e `delete` possono essere overloaded, il che può condurre ad una situazione in cui le chiamate a `new` e `delete` invocano versioni reciprocamente non corrispondenti, risultando in problemi come memory leak e segmentation fault (o in assolutamente nessun problema in certe circostanze). Lo stesso problema può verificarsi se diverse versioni delle librerie standard vengono usate per il linking del modulo e dell'eseguibile.
- Il destructor della classe interfaccia deve solitamente essere virtuale. In rari casi questo può non essere necessario, ma il vantaggio che risulta dalla riduzione dell'overhead è così ridotto da non meritare il rischio.

  Se la propria interfaccia di base non necessita di un destructor, se ne definisca comunque una non contenente istruzioni (e dichiararla `virtual`), altrimenti prima o poi si avranno problemi, posso garantirvelo. Ulteriori informazioni possono essere ottenute nella sezione 20 della FAQ della newsgroup comp.lang.c++ (http://www.parashift.com/c++-faq-lite/).

4. Sorgenti

Tutto il codice sorgente contenuto in questo documento è disponibile in un singolo archivio: examples.tar.gz.

5. FAQ

1. Sto sviluppando sotto Windows e non riesco a trovare l'header `dlfcn.h`! Qual'è il problema?

Il problema è che l'header `dlfcn.h` non esiste sotto Windows, come non esiste una interfaccia `dlopen`. Una API con funzionalità similare è centrata sulla funzione `LoadLibrary`, e la maggior parte delle osservazioni contenute in questo documento è applicabile a questa interfaccia. Si faccia riferimento al sito del Microsoft Developer Network (http://msdn.microsoft.com/) per ulteriori informazioni.

2. Esiste un wrapper per la Windows API `LoadLibrary` che riproduca l'interfaccia di `dlopen`?

Non sono a conoscenza di nessuna API del genere, ed è probabile che un wrapper in grado di supportare tutte le opzioni di `dlopen` non diverrà mai disponibile.

Esistono tuttavia delle alternative: libtltdl (parte di libtool) incapsula una varietà di API per il caricamento dinamico, ivi incluse `dlopen` e `LoadLibrary`. Un'altra alternativa è la funzionalità di caricamento dinamico di moduli di Glib (http://developer.gnome.org/doc/API/glib/glib-dynamic-loading-of-modules.html). Si può utilizzare una di queste API per garantire una miglior portabilità tra piattaforme. Io non le ho mai usate, qundi non posso dire quanto stabili o funzionanti queste soluzioni risultino essere.

Si faccia anche riferimento alla sezione 4, "Dynamically Loaded (DL) Libraries", della Program Library HOWTO (http://www.dwheeler.com/program-library) per altre tecniche per caricare librerie e istanziare classi indipendentemente dalla piattaforma.

6. Approfondimento

- La pagina man per la funzione `dlopen(3)` illustra lo scopo e l'uso dell'API `dlopen`.
- L'articolo *Dynamic Class Loading for C++ on Linux* (http://www.linuxjournal.com/article.php?sid=3687) di James Norton pubblicato sul Linux Journal (http://www.linuxjournal.com/).
- Il proprio materiale di riferimento preferito sull'uso di `extern "C"`, ereditarietà, funzioni virtuali, `new` e `delete`. L'autore raccomanda [STR2000].
- [ISO14882]

- La Program Library HOWTO (http://www.dwheeler.com/program-library), che include la maggior parte di ciò che uno sviluppatore avrà mai bisogno di sapere in materia di librerie statiche, condivise e caricate dinamicamente e come crearle. Altamente raccomandato.
- La Linux GCC HOWTO (http://tldp.org/HOWTO/GCC-HOWTO/index.html) illustra come creare librerie con GCC.

Bibliografia

ISO14482 *ISO/IEC 14482-1998 — The C++ Programming Language*. Disponibile sia in PDF che rilegata in un libro http://webstore.ansi.org/.

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