C++ dlopen mini HOWTO

Aaron Isotton

<aaron@isotton.com>

2006–03–16

Revision History

Revision 1.10	2006–03–16	Revised by: AI
Changed the license from the GFDL to the GPL. Fixed usage of dlerror; thanks to Carmelo Piccione. Using a virtual destructor in the example; thanks to Joerg Knobloch. Added Source Code section. Minor fixes.		
Revision 1.03	2003–08–12	Revised by: AI
Added reference to the GLib Dynamic Module Loader. Thanks to G. V. Sriraam for the pointer.		
Revision 1.02	2002–12–08	Revised by: AI
Added FAQ. Minor changes		
Revision 1.01	2002–06–30	Revised by: AI
Updated virtual destructor explanation. Minor changes.		
Revision 1.00	2002–06–19	Revised by: AI
Moved copyright and license section to the beginning. Added terms section. Minor changes.		
Revision 0.97	2002–06–19	Revised by: JYG
Entered minor grammar and sentence level changes.		
Revision 0.96	2002–06–12	Revised by: AI
Added bibliography. Corrected explanation of extern functions and variables.		
Revision 0.95	2002–06–11	Revised by: AI
Minor improvements.		

How to dynamically load C++ functions and classes using the `dlopen` API.

Table of Contents

1. Introduction

A question which frequently arises among Unix C++ programmers is how to load C++ functions and classes dynamically using the `dlopen` API.

In fact, that is not always simple and needs some explanation. That's what this mini HOWTO does.

An average understanding of the C and C++ programming language and of the `dlopen` API is necessary to understand this document.

This HOWTO's master location is http://www.isotton.com/howtos/C++−dlopen−mini−HOWTO/.

1.1. Copyright and License

1.2. Disclaimer

No liability for the contents of this document can be accepted. Use the concepts, examples and information at your own risk. There may be errors and inaccuracies, that could be damaging to your system. Proceed with caution, and although this is highly unlikely, the author(s) do not take any responsibility.

All copyrights are held by their by their respective owners, unless specifically noted otherwise. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark. Naming of particular products or brands should not be seen as endorsements.

1.3. Credits / Contributors

In this document, I have the pleasure of acknowledging (in alphabetic order):

- Joy Y Goodreau <joyg (at) us.ibm.com> for her editing.
- D. Stimitis <stimitis (at) idcomm.com> for pointing out a few issues with the formatting and the name mangling, as well as pointing out a few subtleties of `extern "C"`.

  Many unnamed others pointing out errors or giving tips to improve this howto. You know who you are!

1.4. Feedback

Feedback is most certainly welcome for this document. Send your additions, comments and criticisms to the following email address: <aaron@isotton.com>.

1.5. Terms Used in this Document

`dlopen` API
: The `dlclose`, `dlerror`, `dlopen` and `dlsym` functions as described in the `dlopen(3)` man page.

  Notice that we use "`dlopen`" to refer to the individual `dlopen` *function*, and "`dlopen` API" to refer to the *entire API*.

---

2. The Problem

At some time you might have to load a library (and use its functions) at runtime; this happens most often when you are writing some kind of plug−in or module architecture for your program.

In the C language, loading a library is very simple (calling `dlopen`, `dlsym` and `dlclose` is enough), with C++ this is a bit more complicated. The difficulties of loading a C++ library dynamically are partially due to name mangling, and partially due to the fact that the `dlopen` API was written with C in mind, thus not offering a suitable way to load classes.

Before explaining how to load libraries in C++, let's better analyze the problem by looking at name mangling in more detail. I recommend you read the explanation of name mangling, even if you're not interested in it because it will help you understanding why problems occur and how to solve them.

2.1. Name Mangling

In every C++ program (or library, or object file), all non−static functions are represented in the binary file as *symbols*. These symbols are special text strings that uniquely identify a function in the program, library, or object file.

In C, the symbol name is the same as the function name: the symbol of `strcpy` will be `strcpy`, and so on. This is possible because in C no two non−static functions can have the same name.

Because C++ allows overloading (different functions with the same name but different arguments) and has many features C does not   like classes, member functions, exception specifications   it is not possible to simply use the function name as the symbol name. To solve that, C++ uses so−called *name mangling*, which transforms the function name and all the necessary information (like the number and size of the arguments) into some weird−looking string which only the compiler knows about. The mangled name of `foo` might look like `foo@4%6^`, for example. Or it might not even contain the word "foo".

One of the problems with name mangling is that the C++ standard (currently [ISO14882]) does not define how names have to be mangled; thus every compiler mangles names in its own way. Some compilers even change their name mangling algorithm between different versions (notably g++ 2.x and 3.x). Even if you worked out how your particular compiler mangles names (and would thus be able to load functions via `dlsym`), this would most probably work with your compiler only, and might already be broken with the next version.

2.2. Classes

Another problem with the `dlopen` API is the fact that it only supports loading *functions*. But in C++ a library often exposes a class which you would like to use in your program. Obviously, to use that class you need to create an instance of it, but that cannot be easily done.

3. The Solution

3.1. `extern "C"`

C++ has a special keyword to declare a function with C bindings: `extern "C"`. A function declared as `extern "C"` uses the function name as symbol name, just as a C function. For that reason, only non−member functions can be declared as `extern "C"`, and they cannot be overloaded.

Although there are severe limitations, `extern "C"` functions are very useful because they can be dynamically loaded using `dlopen` just like a C function.

This does *not* mean that functions qualified as `extern "C"` cannot contain C++ code. Such a function is a full−featured C++ function which can use C++ features and take any type of argument.

3.2. Loading Functions

In C++ functions are loaded just like in C, with `dlsym`. The functions you want to load must be qualified as `extern "C"` to avoid the symbol name being mangled.

Example 1. Loading a Function

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';
}
```

The function `hello` is defined in `hello.cpp`as `extern "C"`; it is loaded in `main.cpp` with the `dlsym` call. The function must be qualified as `extern "C"` because otherwise we wouldn't know its symbol name.

There are two different forms of the `extern "C"` declaration: `extern "C"` as used above, and `extern "C" { & }` with the declarations between the braces. The first (inline) form is a declaration with extern linkage and with C language linkage; the second only affects language linkage. The following two declarations are thus equivalent:

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

and

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

As there is no difference between an `extern` and a non−`extern` *function* declaration, this is no problem as long as you are not declaring any variables. If you declare *variables*, keep in mind that

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

and

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

are *not* the same thing.

For further clarifications, refer to [ISO14882], 7.5, with special attention to paragraph 7, or to [STR2000], paragraph 9.2.4.

Before doing fancy things with extern variables, peruse the documents listed in the see also section.

3.3. Loading Classes

Loading classes is a bit more difficult because we need an *instance* of a class, not just a pointer to a function.

We cannot create the instance of the class using `new` because the class is not defined in the executable, and because (under some circumstances) we don't even know its name.

The solution is achieved through polymorphism. We define a base, *interface* class with virtual members *in the executable*, and a derived, *implementation* class *in the module*. Generally the interface class is abstract (a class is abstract if it has pure virtual functions).

As dynamic loading of classes is generally used for plug−ins which must expose a clearly defined interface we would have had to define an interface and derived implementation classes anyway.

Next, while still in the module, we define two additional helper functions, known as *class factory functions*. One of these functions creates an instance of the class and returns a pointer to it. The other function takes a pointer to a class created by the factory and destroys it. These two functions are qualified as `extern "C"`.

To use the class from the module, load the two factory functions using `dlsym` just as we loaded the the hello function; then, we can create and destroy as many instances as we wish.

Example 2. Loading a Class

Here we use a generic `polygon` class as interface and the derived class `triangle` as implementation.

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;
}
```

There are a few things to note when loading classes:

- You must provide *both* a creation and a destruction function; you must *not* destroy the instances using `delete` from inside the executable, but always pass it back to the module. This is due to the fact that in C++ the operators `new` and `delete` may be overloaded; this would cause a non−matching `new` and `delete` to be called, which could cause anything from nothing to memory leaks and segmentation faults. The same is true if different standard libraries are used to link the module and the executable.
- The destructor of the interface class should be virtual in any case. There *might* be very rare cases where that would not be necessary, but it is not worth the risk, because the additional overhead can generally be ignored.

  If your base class needs no destructor, define an empty (and `virtual`) one anyway; otherwise you *will have problems* sooner or later; I can guarantee you that. You can read more about this problem in the comp.lang.c++ FAQ at http://www.parashift.com/c++−faq−lite/, in section 20.

4. Source Code

You can download all the source code presented in this howto as an archive: examples.tar.gz.

5. Frequently Asked Questions

5.1. I'm using Windows and I can't find the `dlfcn.h` header file! What's the problem?
5.2. Is there some kind of `dlopen`−compatible wrapper for the Windows `LoadLibrary` API?

5.1. I'm using Windows and I can't find the `dlfcn.h` header file! What's the problem?

The problem is that Windows doesn't have the `dlopen` API, and thus there is no `dlfcn.h` header. There is a similar API around the `LoadLibrary` function, and most of what is written here applies to it, too. Please refer to the Microsoft Developer Network Website for more information.

5.2. Is there some kind of `dlopen`−compatible wrapper for the Windows `LoadLibrary` API?

I don't know of any, and I don't think there'll ever be one supporting all of `dlopen`'s options.

There are alternatives though: libtltdl (a part of libtool), which wraps a variety of different dynamic loading APIs, among others `dlopen` and `LoadLibrary`. Another one is the Dynamic Module Loading functionality of GLib. You can use one of these to ensure better possible cross−platform compatibility. I've never used any of them, so I can't tell you how stable they are and whether they really work.

You should also read section 4, "Dynamically Loaded (DL) Libraries", of the Program Library HOWTO for more techniques to load libraries and create classes independently of your platform.

6. See Also

- The `dlopen(3)` man page. It explains the purpose and the use of the `dlopen` API.
- The article *Dynamic Class Loading for C++ on Linux* by James Norton published on the Linux Journal.
- Your favorite C++ reference about `extern "C"`, inheritance, virtual functions, `new` and `delete`. I recommend [STR2000].
- [ISO14882]
- The Program Library HOWTO, which tells you most things you'll ever need about static, shared and dynamically loaded libraries and how to create them. Highly recommended.
- The Linux GCC HOWTO to learn more about how to create libraries with GCC.

Bibliography

ISO14482 *ISO/IEC 14482−1998 The C++ Programming Language*. Available as PDF and as printed book from http://webstore.ansi.org/.

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