LOADING LIBRARY FILES IN C++

This article demonstrates how to load shared or dynamic library files in programs written in C++, which is not as straightforward as in C.

Device drivers and library files have always been associated with the C programming language, and dominated by C programmers, because of the straightforward symbols of C's libraries, which are directly related to the function name. Loading a library in C is simpler than in C++, mainly due to the issue of name mangling, which we will examine later. Another problem of using `dlopen` in C++ is that the `dlopen` API supports loading of functions, but in C++, to use the methods of a class, normally, you need to instantiate it.

Name mangling

In any executable or shared library, all non-static functions are represented by a symbol, which is usually the same as the function name, and represents the start address of the function in memory. In C, the symbol is the same as the function name — e.g., the symbol for the `init`function will be `init`, since no two functions can have the same name.
However, in C++, because of overloading and polymorphism in classes, it is possible to have the same name for two functions in a program. Hence, it's not possible to use the function name as the symbol. To solve this problem of having two functions with the same name, C++ compilers use name-mangling techniques, in which they change the symbol names (you can read more about this on Wikipedia). Name mangling makes it difficult for programmers to access a specific symbol in the compiled shared library file, even if they know the original function name.

The solution to this issue is to use the special keyword extern “C” before the function implementation (i.e., `extern "C" void function_name()`). This tells the C++ compiler to compile the function in C style — to keep the symbol name the same as the function name. We can use this keyword to define a function that returns an instance of a class, which also solves the must-instantiate-class problem mentioned earlier.
Now, let’s look at an example of building a library, and then loading it. First, let’s make the interface for the shared library, which we can use to reference the shared library and our main programs:

```
1
2   #ifndef TESTVIR H
3   #define TESTVIR_H
4
5   class TestVir
6   {
7   public:
8     virtual void init()=0;
9   };
10
11  #endif
```

Let’s start by making a sample shared library, `testLib.h`:

```
1
2   #ifndef TESTLIB_H
3   #define TESTLIB_H
4
5   class TestLib
6   {
7    public:
8        void init();
9   };
10
11  #endif
```

In the above header file, we declared a class and the `init` method as public, as we need to access it later. The next code is `testLib.cpp`:

```
1   #include <iostream>
```

```
2  #include "testVir.h"
3  #include "testLib.h"
4
5  using namespace std;
6  void TestLib::init()
7  {
8    cout<<"TestLib::init: Hello World!! "<<endl ;
9  }
10
11 //Define functions with C symbols (create/destroy TestLib instance).
12 extern "C" TestLib* create()
13 {
14    return new TestLib;
15 }
16 extern "C" void destroy(TestLib* Tl)
17 {
18   delete Tl ;
19 }
```

Our two-class helper functions with `extern "C"` and will be used to create and destroy an instance of the shared library class, and serve as access points (entry and exit points) of the library.

Let us compile and link the above class with the shared and `fPIC` options to `g++`:

```
g++ -shared -fPIC testLib.cpp -o testLib.so
```

Now, let's write a program to access this library — `main.cpp`:

```
1  #include<iostream>
2  #include<dlfcn.h>
3  #include "testVir.h"
4
5  using namespace std;
6
7  int main()
8  {
9     void *handle;
10    handle = dlopen("./testLib.so", RTLD_NOW);
```

```
9       if (!handle)
10      {
11          printf("The error is %s", dlerror());
12      }
13
14      typedef TestVir* create_t();
15      typedef void destroy_t(TestVir*);
16
17      create_t* creat=(create_t*)dlsym(handle,"create");
18      destroy_t* destroy=(destroy_t*)dlsym(handle,"destroy");
19      if (!creat)
20      {
21          cout<<"The error is %s"<<dlerror();
22      }
23      if (!destroy)
24      {
25          cout<<"The error is %s"<<dlerror();
26      }
27      TestVir* tst = creat();
28      tst->init();
29      destroy(tst);
30      return 0 ;
31  }
32
33
```

In the above program, we used `dlopen` to load the library, then retrieved references to the symbols for our two access functions with `dlsym`, and then via these, first invoked the create function to get an instance of the shared object, invoked its `init` method, and then destroyed the object used to invoke the methods of the C++ class.

Next, compile and link the program with `g++ -ldl main.cpp -o test`. The option `-ldl` is used to link it with `libdl.so` to use the methods in the `dlfcn` header file.

The above scenario is basically used to build a plugin framework, where the plugins are dynamically loaded into the `main` component.

Source : http://www.opensourceforu.com/2011/12/loading-library-files-in-cpp/