Shared Libraries

Loading shared libraries and classes dynamically at runtime.
Overview

- Shared Libraries
- The Class Loader
Most modern platforms provide facilities to load program modules in the form of shared libraries (dynamic link libraries) at runtime.

Windows provides a `LoadLibrary()` function, most Unix platforms have `dlopen()`.

To use a dynamically loaded shared library, an entry point (address of a function) into the library must be found. The address can then be casted to an appropriate function pointer, and the function can be called.
The SharedLibrary Class

> Poco::SharedLibrary is POCO's interface to the operating system's dynamic linker/loader.

> #include "Poco/SharedLibrary.h"

> Poco::SharedLibrary provides low-level functions for loading a shared library, for looking up the address of a symbol, and for unloading a shared library.
SharedLibrary Functions

- `void load(const std::string& path)`
  loads the shared library from the given path

- `void unload()`
  unloads the shared library

- `bool hasSymbol(const std::string& name)`
  returns true if the library contains a symbol with the given name

- `void* getSymbol(const std::string& name)`
  returns the address of the symbol with the given name. For a function, this is the entry point of the function. To call the function, cast to a function pointer and call through it.
// TestLibrary.cpp

#include <iostream>

#if defined(_WIN32)
    #define LIBRARY_API __declspec(dllexport)
#else
    #define LIBRARY_API
#endif

extern "C" void LIBRARY_API hello();

void hello()
{
    std::cout << "Hello, world!" << std::endl;
}
// LibraryLoaderTest.cpp
#include "Poco/SharedLibrary.h"

using Poco::SharedLibrary;

typedef void (*HelloFunc)(); // function pointer type

int main(int argc, char** argv)
{
    std::string path("TestLibrary");
    path.append(SharedLibrary::suffix()); // adds ".dll" or ".so"
    SharedLibrary library(path); // will also load the library
    HelloFunc func = (HelloFunc) library.getSymbol("hello");
    func();
    library.unload();
    return 0;
}
The Class Loader

- Poco::ClassLoader is POCO's high level interface for loading classes from shared libraries. It is well suited for implementing typical plug-in architectures.

- #include "Poco/ClassLoader.h"

- All classes loaded with a specific class loader must be subclasses of a common base class. Poco::ClassLoader is a class template that must be instantiated for the base class.

- A base class is necessary because the application loading a plugin needs an interface to access it.
A shared library that is used with the class loader can only export classes that have a common base class.

However, this is not really a restriction, because the exported class can be a factory for objects of arbitrary classes.

A shared library used with the class loader exports a Manifest describing all classes exported by the library.

Furthermore, the shared library must export specific functions that are used by the class loader.

POCO provides macros that automate the implementation of these functions.
Manifest and MetaObject

- A library's Manifest maintains a list of all classes contained in a dynamically loadable class library.
- It manages that information as a collection of meta objects.
- A MetaObject manages the lifetime of objects of a given class. It is used to create instances of a class, and to delete them.
- As a special feature, class libraries can export singletons.
The MetaObject Class

- `MetaObject<Class, Base>` is a class template, instantiated with the class it maintains, and its lowest base class.
- `MetaObject<Class, Base>` is derived from `AbstractMetaObject<Base>`.
- A `MetaObject` can be used to create new instances of a class (unless the class is exported as a singleton).
- Like a `AutoReleasePool`, a `MetaObject` can take care of no longer needed objects.
- A `MetaObject` can manage singletons.
The MetaObject Class (cont'd)

- `const char* name()` returns the name of the class
- `Base* create() const` creates a new instance of `Class`, unless it's a singleton.
- `Base& instance() const` returns a reference to the one and only instance of a singleton.
- `bool canCreate() const` returns `true` if new instances can be created (`false` if the class is a singleton).
The MetaObject Class (cont'd)

- `Base* autoDelete(Base* pObject)`
give ownership of `pObject` to the `MetaObject`. The `MetaObject` will delete all object it owns when it's destroyed.

- `bool isAutoDelete(Base* pObject)`
returns `true` if the `MetaObject` owns `pObject`, `false` otherwise.

- `void destroy(Base* pObject)`
if the `MetaObject` owns `pObject`, it will be immediately deleted.
The MetaSingletonClass

> This is a sister class of MetaObject used for managing singletons.

> It has the same interface as MetaObject.
The Manifest Class

- **Poco::Manifest** basically is a collection of meta objects.

- #include "Poco/Manifest.h"

- **Poco::Manifest::Iterator** is used to iterate over its meta objects.

- **Manifest::Iterator find(const std::string& className)** returns an iterator pointing to the meta object for the given class, or an end iterator if the class is not found.

- **Manifest::Iterator begin() const**
  **Manifest::Iterator end() const** return the begin, and end iterator, respectively.
Writing a Class Library

- For a class library to work with the class loader, it must export a manifest.
- The class library must provide a function `bool pocoBuildManifest(ManifestBase* pManifest)` that builds a manifest for the library.
- The `Poco/ClassLibrary.h` header file provides macros to automatically implement this function for a class library.
- Optionally, a class library can export an initialization and a cleanup function.
Writing a Class Library (cont'd)

> These macros are used as follows:
POCO_BEGIN_MANIFEST(MyBaseClass)
    POCO_EXPORT_CLASS(MyFirstClass)
    POCO_EXPORT_CLASS(MySecondClass)
    POCO_EXPORT_SINGLETON(MySingleton)
POCO_END_MANIFEST

> A class library can export a setup and a cleanup function:
void pocoInitializeLibrary()
void pocoUninitializeLibrary()
which will be called by the class loader, if present.
The Class Loader (again)

- A **ClassLoader** maintains a collection of class libraries, as well as their manifests.

- `void loadLibrary(const std::string& path)` loads a class library into memory and runs the set up function, if it's present.

- `void unloadLibrary(const std::string& path)` unloads a class library after running the clean up function.

- Never unload a class library if there are still objects from this library around in memory.
The Class Loader (again, cont'd)

- `const Meta* findClass(const std::string& className) const` looks for the meta object for the given class in all loaded libraries. Returns a pointer to the meta object if found, null otherwise.

- `const Meta& classFor(const std::string& className)` similar to `findClass()`, but throws a `NotFoundException` if the class is not known.

- `Base* create(const std::string& className)` creates a new instance of a class or throws a `NotFoundException` if the class is unknown.
The Class Loader (again, cont'd)

> Base& instance(const std::string& className)
returns a reference to the only instance of a singleton class or
throws a `NotFoundException` if the class is unknown.

> Iterator begin() const
Iterator end() const
return a begin/end iterator for iterating over the manifests of all
loaded libraries. Dereferencing the iterator will return a pointer to
a `std::pair` containing the path of the class library and a pointer to
its manifest.

> Please see the reference documentation for other member
functions.
/ AbstractPlugin.h
//
// This is used both by the class library and by the application.

#ifndef AbstractPlugin_INCLUDED
#define AbstractPlugin_INCLUDED

class AbstractPlugin
{
 public:
  AbstractPlugin();
  virtual ~AbstractPlugin();
  virtual std::string name() const = 0;
};

#endif // AbstractPlugin.h
// AbstractPlugin.cpp
//
// This is used both by the class library and by the application.

#include "AbstractPlugin.h"

AbstractPlugin::AbstractPlugin()
{
}

AbstractPlugin::~AbstractPlugin()
{
}
// PluginLibrary.cpp

#include "AbstractPlugin.h"
#include "Poco/ClassLibrary.h"
#include <iostream>

class PluginA: public AbstractPlugin
{
    public:
        std::string name() const
        {
            return "PluginA";
        }
};

class PluginB: public AbstractPlugin
{
    public:
        std::string name() const
        {
            return "PluginB";
        }
};
POCO_BEGIN_MANIFEST(AbstractPlugin)
    POCO_EXPORT_CLASS(PluginA)
    POCO_EXPORT_CLASS(PluginB)
POCO_END_MANIFEST

// optional set up and clean up functions

void pocoInitializeLibrary()
{
    std::cout << "PluginLibrary initializing" << std::endl;
}

void pocoUninitializeLibrary()
{
    std::cout << "PluginLibrary uninitializing" << std::endl;
}
/ main.cpp

#include "Poco/ClassLoader.h"
#include "Poco/Manifest.h"
#include "AbstractPlugin.h"
#include <iostream>

typedef Poco::ClassLoader<AbstractPlugin> PluginLoader;
typedef Poco::Manifest<AbstractPlugin> PluginManifest;

int main(int argc, char** argv)
{
    PluginLoader loader;

    std::string libName("PluginLibrary");
    libName += Poco::SharedLibrary::suffix(); // append .dll or .so
    loader.loadLibrary(libName);
PluginLoader::Iterator it(loader.begin());
PluginLoader::Iterator end(loader.end());
for (; it != end; ++it)
{
    std::cout << "lib path: " << it->first << std::endl;
    PluginManifest::Iterator itMan(it->second->begin());
    PluginManifest::Iterator endMan(it->second->end());
    for (; itMan != endMan; ++itMan)
        std::cout << itMan->name() << std::endl;
}

AbstractPlugin* pPluginA = loader.create("PluginA");
AbstractPlugin* pPluginB = loader.create("PluginB");

std::cout << pPluginA->name() << std::endl;
std::cout << pPluginB->name() << std::endl;

loader.classFor("PluginA").autoDelete(pPluginA);
delete pPluginB;

loader.unloadLibrary(libName);

return 0;