Building Your Own Plugin Framework: Part 2

Architecture and design

December 07, 2007
URL:http://www.drdobbs.com/cpp/building-your-own-plugin-framework-part/204702751

Gigi Sayfan specializes in cross-platform object-oriented programming in C/C++/ C#/Python/Java with emphasis on large-scale distributed systems. He is currently trying to build intelligent machines inspired by the brain at Numenta (www.numenta.com).

Editor's Note: This is the second installment of a five-part series by Gigi Sayfan on creating cross-platform plugins in C++. Other installments are: Part 1, Part 3, Part 4, and Part 5.

This article is the second in a series of articles about developing cross-platform plugins in C++. The first article described the problem in detail, explored various solutions, and introduced the plugin framework. In this installment, I describe the architecture and design of a plugin-based system based on the plugin framework, the lifecycle of a plugin, and the internals of the generic plugin framework. Beware! Some code may surface here and there.

The Architecture of a Plugin-Based System

A plugin-based system can be divided into three parts that are loosely coupled: The main system or application with its object model, the plugin manager and the plugins themselves. The plugins conform to the plugin manager's interfaces and protocols and also implement the object model interfaces. Let's illustrate it with a concrete example. The main system is a turn-based game. The game takes place in a battlefield that contains various monsters. The hero fights the monsters until he dies or all the monsters die. Pretty basic yet gratifying. Listing One is the definition of the **Hero** class.

```
#ifndef HERO_H
#define HERO_H

#include <vector>
#include <map>
#include <boost/shared_ptr.hpp>
#include "object_model/object_model.h"

class Hero : public IActor
{
public:
  Hero();
  ~Hero();
  // IActor methods
  virtual void getInitialInfo(ActorInfo * info);
  virtual void play(ITurn * turnInfo);
private:
};
#endif
```

Listing One

The **BattleManager** is the engine that drives the game. It takes care of instantiating the hero and the monsters and populating the battlefield. Then in each turn it calls upon each actor (hero or monsters) to do their worst via the **play()** method.

The hero and the monsters implement the **IActor** interface. The hero is a built-in game object with a predefined behavior. The monsters on the other hand are implemented as plugin objects. This allows the game to be extended with new monsters and decouples the development of new monsters from the development of the main game engine. The **PluginManager**'s job is to abstract away the fact that monsters are spawned from plugins and present them to the **BattleManager** as actors just like the hero. This scheme also allows the game to come with some built-in monsters that are statically linked in and are not implemented in plugins. The **BattleManager** shouldn't even be aware ideally there is such a thing as plugins. It should just operate at the C++ object level. This makes it easier to test too, because you can create mock monster in the test code without having to write a full-fledged plugin.

The **PluginManager** itself can be either generic or specialized. A generic plugin manager is unaware of the specific underlying object model. When a C++ **PluginManager** instantiates a new object implemented in a plugin it must return a generic interface, which the caller must cast to the actual interface from the object model. This is a little ugly, but necessary. A custom PluginManager is aware of your object model and can operate in terms of the underlying object model. For example, a custom **PluginManager** for our game can have a **CreateMonster()** function that returns an **IActor** interface. The **PluginManager** I show is a generic one, but I'll demonstrate how simple it is to put an object model specific layer on top of it. This is standard practice because you don't want your application code to deal with explicit casts.

Plugin System Lifecycle

It's time to figure out the lifecycle of the plugin system. The application, **PluginManager** and the plugins themselves participate in a complicated dance according to a strict protocol. The good news are that the generic plugin framework can mostly orchestrate the process. The application gets access to the plugins when it needs it and the plugins just need to implement a few functions that will be called in due time.

Registration of Static Plugins

Static plugins are plugins that are deployed in static libraries and are linked statically into the application. The registration can be done automatically if the library defines a global registrar object whose constructor is called automatically. Unfortunately it doesn't work on all platforms (e.g., Windows). The alternative is to explicitly tell the **PluginManager** to initialize static plugin by passing it a dedicated init function. Since, all the static plugin are statically linked to the main executable the **init()** function (which must have the signature of **PF_InitPlugin**) of each plugin must have a unique name. A good convention is something like **<Plugin Name>_InitPlugin()**. Here is the prototype of the **init()** function of a static plugin called "StaticPlugin":

```
extern "C" PF_ExitFunc
StaticPlugin_InitPlugin(const PF_PlatformServices * params)
```

The explicit initialization creates a tight coupling between the main application and the static plugins because the main application need to "know" at compile time what plugins are linked to it in order to initialize them. The process can be automated as part of the build if all the static plugins follow some convention that the build process can use to find them and generate the code that initializes each one of them on-the-fly.

Once a static plugin's **init()** function is called it will register all its object types with the **PluginManager**.

Loading of Dynamic Plugins

Dynamic plugins are the more common ones. They should all be deployed in a dedicated directory. The application should invoke the **PluginManager**'s **loadAll()** method and pass the dedicated directory path. The **PluginManager** scans all the files in this directory and load every dynamic library. The application may alternatively call the **load()** method, which loads a single plugin if it wants fine-grained control about what plugins are loaded exactly.

Plugin Initialization

Once a dynamic library has been loaded successfully, the **PluginManager** is looking for a well-known function entry point called **PF_initPlugin**. If such an entry point is found, the **PluginManager** initializes the plugin by calling this function and passing the **PF_PlatformServices struct**. This **struct** contains the **PF_PluginAPI_Version**, which lets the plugin perform some version negotiation and decide if it can function properly. If the application's version is inappropriate the plugin may decide to fail the initialization.The **PluginManager** logs the fact that the plugin wasn't initialized properly and continues to load the next plugin. It is not a fatal error from the point of view of the **PluginManager** if a plugin fails to load or initialize. The application may perform additional checks by enumerating the loaded plugins and verify that no crucial plugin is missing.

Listing Two contains the **PF_initPlugin** function of the C++ plugin (and its **exit** function).

```
#include "cpp_plugin.h"
#include "plugin_framework/plugin.h"
#include "KillerBunny.h"
#include "StationarySatan.h"

extern "C" PLUGIN_API apr_int32_t ExitFunc()
{
  return 0;
}
extern "C" PLUGIN_API PF_ExitFunc PF_initPlugin(const PF_PlatformServices * params)
{
  int res = 0;
  PF_RegisterParams rp;
  rp.version.major = 1;
  rp.version.minor = 0;
  rp.programmingLanguage = PF_ProgrammingLanguage_CPP;
  // Regiater KillerBunny
  rp.createFunc = KillerBunny::create;
  rp.destroyFunc = KillerBunny::destroy;
  res = params->registerObject((const apr_byte_t *)"KillerBunny", &rp);
  if (res < 0)
    return NULL;
  // Regiater StationarySatan
  rp.createFunc = StationarySatan::create;
  rp.destroyFunc = StationarySatan::destroy;
  res = params->registerObject((const apr_byte_t *)"StationarySatan", &rp);
  if (res < 0)
    return NULL;
  return ExitFunc;
}
```

Listing Two

Object Registration

The ball is now in the hands of the plugin itself (inside the **PF_initPlugin** code). If the version negotiation went well, the plugin should register all the object types it supports with the plugin manager. The purpose of the registration is to provide to the application functions like **PF_CreateFunc** and **PF_DestroyFunc** that it can use later on to create and destroy plugin objects. This arrangements allows the plugin to control the actual creation and destruction of objects including any resources they manage (like memory), but lets the application control the number of objects and their lifetime. Of course, a plugin may implement singletons by always returning the same object instance.

The registration is done by preparing for each object type registration record (**PF_RegisterParams**) and calling the **registerObject()** function pointer provided in the **PF_PlatformServices struct** (that was passed as argument to **PF_initPlugin**). The **registerOBject()** function accepts a string that uniquely identifies the object type or a wildcard "*" and the **PF_RegisterParams struct**. I'll explain the purpose of the type string and how it is used in the next section. The reason a type string is necessary is because different plugins may support multiple types of objects.

You can see in Listing Two that the C++ plugin registers two monster types -- "KillerBunny" and "StationarySatan".

Now, the shoe is on the other foot. Once the plugin calls **registerObject()** control goes back to the **PluginManager**. The **PF_RegisterParams** contains also a version and a programming language fields. The version field lets the **PluginManager** make sure it can work with this object type. If there is a version mismatch it will not register the object. It is not a fatal error. This allows fairly flexible negotiations, where the plugin tries to register multiple versions of the same object type in order to take advantage of newer interfaces if they exist and fallback to older interfaces. The programming language field will be explained soon. If the plugin manager is happy with the **PF_RegisterParams struct**, it just stores it in an internal data structure that maps the object type to the **PF_RegisterParams struct**.

After the plugin registered all its object types, it returns a function pointer to a **PF_ExitFunc**. This function is called before the plugin is unloaded and lets the plugin clean up any global resources it acquired during its life time.

If the plugin decides that it can't function properly (can't allocate some resource, crucial object type registration failed, version mismatch) it should cleanup after itself and return NULL. This signals the **PluginManager** that the plugin initialization failed. The **PluginManager** will also remove all registrations performed by the failed plugin.

Plugin Object Creation by the Application

At this point all the dynamic plugins have been loaded and both static and dynamic plugins have been initialized and registered all the object types they support. The application can now create object instances by calling the **PluginManager**'s **createObject()** method. This method accepts an object type string and an **IObjectAdapter** interface. I'll discuss object adaptation in the next section, so let's focus on the object type string. The application needs to know what object types are supported. This knowledge can be hard-coded into the application or it can query the plugin's manager registration map and find out at runtime what object types are currently registered.

If you recall, the type string can be either a unique type identifier or a wildcard "*". When the application calls **createBbject()** with a type string ("*" is an invalid type string) the **PluginManager** looks for an exact match in its registration map. If it find a match it will invoked the registered **PF_CreateFunc** and return the result to the application (possibly after adaptation). If it can't find a match it will go over all the wild card registrations (plugins that registered with "*" type string) and let them try by invoking their registered **PF_CreateFunc**. If any plugin returns a non-NULL result it is returned to the application.

What is the purpose of the wildcard registration? It lets plugins create objects they don't know about at registration time. What? Yes. In Numenta, we used it to allow Python plugins. A single C++ plugin registered with a "*" type string. If the application requested a Python class (the type was the actual qualified import path of a Python class) then the C++ plugin who had an embedded Python interpreter created a special object that held an instance of the Python class and forwarded plugin requests to its internal Python object (via the Python C API). To the application it appeared as a standard C++ object. This allows great flexibility because it is possible to just drop a Python class in the right place even while the system is running and the Python object is immediately available.

Automatic Adaptation of C-based Objects

Again, the plugin framework supports both C and C++ plugins. C and C++ plugin objects implement different interfaces. The main innovation I present in the next installment is how to design and implement a dual C/C++ object model. That unified object model can be transparently accessed and manipulated by both C and C++ objects. However, if the application had to deal with each plugin using its native interface, it would be highly inconvenient. The application code would be peppered with **if** statements and every argument would have to be converted to the proper data type, which is also very inefficient. The plugin framework uses two techniques to overcome these obstacles.

- First, the object model consists of objects that implement both the C and C++.
- Second, C objects are wrapped by a special adapter that exposes a C++ facade that implements the corresponding C++ interface. The end result is that the application can be blissfully ignorant of the fact that there are C plugins at all. It can treat all plugin objects as C++ objects, since they will all implement the C++ interface.

The actual adaptation is done using an object adapter. This is an object provided by the application (just a specialization of the **ObjectAdapter** template provided by the plugin framework) that implements the **IObjectAdapter** interface.

Listing Three contains the **IObjectAdapter** interface and the **ObjectAdapter** template.

```
#ifndef OBJECT_ADAPTER_H
#define OBJECT_ADAPTER_H

#include "plugin_framework/plugin.h"

// This interface is used to adapt C plugin objects to C++ plugin objects.
// It must be passed to the PluginManager::createObject() function.
struct IObjectAdapter
{
  virtual ~IObjectAdapter() {}
  virtual void * adapt(void * object, PF_DestroyFunc df) = 0;
};
// This template should be used if the object model implements the
// dual C/C++ object design pattern. Otherwise you need to provide
// your own object adapter class that implements IObjectAdapter
template<typename T, typename U>
struct ObjectAdapter : public IObjectAdapter
{
  virtual void * adapt(void * object, PF_DestroyFunc df)
  {
    return new T((U *)object, df);
  }
};
#endif // OBJECT_ADAPTER_H
```

Listing Three

The **PluginManager** uses it to adapt a C object to a C++ object. I explain the process in detail when I go over the various components of generic plugin framework later in this article.

The important thing to take home is that the plugin framework provides all the necessary infrastructure necessary to adapt a C object to a C++ object, but it needs the application's help because it doesn't know the types of objects it needs to adapt.

Interaction Between the Application and Plugin Objects

The application simply calls C++ member functions on the C++ interfaces of plugin objects (possibly adapted C objects) it created. In addition to dutifully returning results from their member functions, the plugin objects may also invoke callback functions through the **PF_InvokeService** function of the **PF_PlatformServices struct**. These services can be used for diverse purposes like logging, error reporting, progress notifications of long running operations, and event propagation. Again, these callbacks are part of the protocol between the application and plugins and must be designed as part of the entire application interfaces and object model design.

Destruction of Plugin Objects by the Application

The best practice in managing object lifetime is that the creator is also the destroyer. This is especially important in a language like C++ where you are responsible for memory allocation and deallocation. There are many ways to allocate and deallocate memory: malloc/free, new/delete, array new/delete, OS specific APIs that allocate/deallocate from different heaps, etc. It is often very important to deallocate using the deallocation method that corresponds to the allocation method. The creator is in the best position to know how resources where allocated. In the plugin framework every object type is registered with both a create function and a destroy function (**PF_CreateFunc** and **PF_DestroyFunc**). Plugin objects are created using **PF_CreateFunc** and should be destroyed using **PF_DestroyFunc**. Each plugin is responsible for implementing both properly so all resources are cleaned up properly. The plugin is free to implement any memory scheme it wants. All the plugin objects may be allocated statically and **PF_DestroyFunc** may do nothing or there could be a pool of pre-created instances and **PF_DestroyFunc** may just return an object to the pool. The application just creates objects using **PF_CreateFunc** and releases them when its done with them using **PF_DestroyFunc**. The destructor of C++ plugin objects does the right thing, so the application doesn't have to deal with calling **PF_DestroyFunc** directly and can dispose of plugin objects using the standard delete operator. This works for adapted C objects too, because the object adapter makes sure to call the proper **PF_DestroyFunc** in its destructor.

Plugin System Cleanup When Applications Shut Down

When the application exits it needs destroy all the plugin objects it created and notify all the plugins (both static and dynamic) that it's time to cleanup. The application does it by calling the **PluginManager**'s **shutdown()**PluginManager in turn calls the **PF_ExitFunc** of each plugin (returned from the **PF_initPlugin** function if successful) and unloads all the dynamic plugins. It is important to call the exit function even if the application is about to exit and all the memory the plugins hold will be reclaimed automatically. The reason is that there are other types of resources that are not reclaimed automatically and also because the plugins might have some buffered state they need to commit/flush/send over the network etc. Lucky for the application the **PluginManager** takes care of that.

In some situations the application may also choose to unload only a single plugin. In this case too, the exit function must be called, the plugin itself unloaded (if it's a dynamic plugin) and removed from all the **PluginManager**'s internal data structures.

Plugin System Components

This section describes the main components of the generic plugin framework and what they do. You can find all these components in the **plugin_framework** subdirectory of the source code.

DynamicLibrary

The **DynamicLibrary** component is a simple cross-platform C++ class. It uses the dlopen/dlclose/dlsym system calls on UNIX (including the Mac OS, X) and the LoadLibrary/FreeLibrary/GetProcAddress API calls for Windows.

Listing Four is the header file for **DynamicLibrary**.

```
#ifndef DYNAMIC_LIBRARY_H
#define DYNAMIC_LIBRARY_H

#include <string>

class DynamicLibrary
{
public:
  static DynamicLibrary * load(const std::string & path,
                              std::string &errorString);
  ~DynamicLibrary();
  void * getSymbol(const std::string & name);
private:
  DynamicLibrary();
  DynamicLibrary(void * handle);
  DynamicLibrary(const DynamicLibrary &);
private:
  void * handle_;
};
#endif
```

Listing Four

Each dynamic library is represented by an instance of the **DynamicLibrary** class. Loading a dynamic library involves calling the static **load()** method that returns a **DynamicLibrary** pointer if everything was fine or NULL if it failed. The **errorString** output argument contains the error message if any. The dynamic library will store the platform-specific handle used to represent the loaded library, so it will be available for getting symbols and unloading later.

The **getSymbol()** method is used to get symbols out of loaded library and the destructor unloads the library (just delete the pointer).

There are different ways to load dynamic libraries. For simplicity, **DynamicLibrary** just picks one option on each platform. It is possible to extend it, but due to platform differences the interface will not be simple anymore.

PluginManager

PluginManager is the big Kahuna of the plugin framework. Everything that has to do with plugins goes through the **PluginManager**. Listing Five contains the header file the **PluginManager**.

```
#ifndef PLUGIN_MANAGER_H
```

```
#define PLUGIN_MANAGER_H

#include <vector>
#include <map>
#include <apr-1/apr.h>
#include <boost/shared_ptr.hpp>
#include "plugin_framework/plugin.h"

class DynamicLibrary;
struct IObjectAdapter;

class PluginManager
{
  typedef std::map<std::string, boost::shared_ptr<DynamicLibrary> > DynamicLibraryMap;
  typedef std::vector<PF_ExitFunc> ExitFuncVec;
  typedef std::vector<PF_RegisterParams> RegistrationVec;
public:
  typedef std::map<std::string, PF_RegisterParams> RegistrationMap;
  static PluginManager & getInstance();
  static apr_int32_t initializePlugin(PF_InitFunc initFunc);
  apr_int32_t loadAll(const std::string & pluginDirectory, PF_InvokeServiceFunc func = NULL);
  apr_int32_t loadByPath(const std::string & path);
  void * createObject(const std::string & objectType, IObjectAdapter & adapter);
  apr_int32_t shutdown();
  static apr_int32_t registerObject(const apr_byte_t * nodeType,
                                    const PF_RegisterParams * params);
  const RegistrationMap & getRegistrationMap();
private:
  ~PluginManager();
  PluginManager();
  PluginManager(const PluginManager &);

  DynamicLibrary * loadLibrary(const std::string & path, std::string & errorString);
private:
  bool                inInitializePlugin_;
  PF_PlatformServices platformServices_;
  DynamicLibraryMap   dynamicLibraryMap_;
  ExitFuncVec         exitFuncVec_;

  RegistrationMap     tempExactMatchMap_;   // register exact-match object types
  RegistrationVec     tempWildCardVec_;     // wild card ('*') object types

  RegistrationMap     exactMatchMap_;       // register exact-match object types
  RegistrationVec     wildCardVec_;         // wild card ('*') object types
};
#endif
```

Listing Five

The application initiates the loading of plugins by calling **PluginManager::loadAll()**, passing the directory that contains the plugins. The **PluginManager** loads all the dynamic plugins and initializes them. It stores every dynamic plugin library in the **dynamicLibraryMap_**, every exit function in **exitFuncVec_** (for both dynamic and static plugins) and every registered type in the **exactMatchMap_**. Wildcard registrations are stored in the **wildCardVec_**. The **PluginManager** is now ready to create plugin objects. If there are static plugins they are registered too (either by the application or via auto-registration).

During plugin initialization the **PluginManager** keeps all registrations in temporary data structures that are merged into **exactMatchMap_** and **wildCardVec_** if the initialization is successful and discarded if it fails. This transactional behavior guaranties that all stored registrations come from successfully initialized plugins that are still loaded into memory.

When the application needs to create a new plugin object (either dynamic or static) it calls the **PluginManager::createObject()** and passes an object type and an adaptor. The **PluginManager** creates the object using the registered **PF_CreateFunc** and adapts it from C to C++ if it's a C object (based on the **PF_ProgrammingLanguage** member of the registration struct).

At this point, the **PluginManager** gets out of the picture. The application interacts with the plugin object directly and finally destroys it. The **PluginManager** is blissfully ignorant of all these interactions. It is the responsibility of the application to destroy plugin objects before the plugin is unloaded (or at least not call its methods after its plugin was unloaded).

The **PluginManager** is in charge of unloading the plugins too. The **PluginManager::shutdown()** method should be called the application after it destroyed all the plugin objects it created. The **shutdown()** calls the exit function of all the plugins (both static and dynamic), unloads all the dynamic plugins and clears all the internal data structures. It is possible to "restart" the **PluginManager** by calling its **loadAll()** method. The application may also "restart" static plugins by calling the **PluginManager::initializePlugin()** for each one. Static plugins that used auto-registration will be gone for good.

If the application forgets to call **shutdown()**, the **PluginManager** will call it in its destructor.

ObjectAdapter

The **ObjectAdapter**'s job is to adapt a C plugin object to a C++ plugin object as part of object creation. The **IObjectAdapter** interface is straightforward. It defines a single method (besides the mandatory virtual destructor) -- adapt. The **adapt()** method accepts a void pointer, which will be a C object and **PF_DestroyFunc** function pointer that can destroy the C object. It is supposed to return a C++ object that wraps the C object. It is the responsibility of the application to provide a proper wrapper object. The plugin framework can't do it because this task requires knowledge of the application object model. However, the plugin framework provides the **ObjectAdapter** template that simply performs a **static_cast** of the C object to the wrapper object provided by the application (see Listing Three).

The resulting object can be passed to any context that requires the C++ interface. This will be the main focus of the next article in this series, so don't sweat it if it sounds a little obscure at the moment. The main point here is that the **ObjectAdapter** template provides an implementation of the **IObjectAdapter** interface that the application can specialize to its own C plugin object and its C++ wrapper.

Listing Six contains the **ActorFactory** that subclasses a specialization of the **ObjectAdapter** template. It functions as an adapter from a C object that implements the **C_Actor** to the **ActorAdapter** C++ object that implements the **IActor** interface. **ActorFactory** also provides a static **createActor()** function that calls the **PluginManager**'s **createObject()** with itself as the adapter and casts the resulting void pointer to an **IActor** pointer. This lets the application call a friendly **createActor()** static function with just an actor type and not mess with adapters and casts.

```
#ifndef ACTOR_FACTORY_H
#define ACTOR_FACTORY_H

#include "plugin_framework/PluginManager.h"
#include "plugin_framework/ObjectAdapter.h"
#include "object_model/ActorAdapter.h"

struct ActorFactory : public ObjectAdapter<ActorAdapter, C_Actor>
{
  static ActorFactory & getInstance()
  {
    static ActorFactory instance;

    return instance;
  }
  static IActor * createActor(const std::string & objectType)
  {
    void * actor = PluginManager::getInstance().createObject(objectType, getInstance());
    return (IActor *)actor;
```

Listing Six

```
  }
};
#endif // ACTOR_FACTORY_H
```

PluginRegistrar

The **PluginRegistrar** lets static plugins register their objects automatically with the **PluginManager** without requiring the application to explicitly initialize them. The way it works (when it works) is that the plugin defines a global instance of the **PluginRegistrar** and passes it its initialization function (with a signature that matches **PF_InitFunc**). The **PluginRegistrar** simply calls the **PluginManager::initializePlugin()** method that ignites the static plugin initialization just like with dynamic plugins after loading the dynamic library; see Listing Seven.

```
#ifndef PLUGIN_REGISTRAR_H
#define PLUGIN_REGISTRAR_H
#include "plugin_framework/PluginManager.h"
struct PluginRegistrar
{
  PluginRegistrar(PF_InitFunc initFunc)
  {
    PluginManager::initializePlugin(initFunc);
  }
};
#endif // PLUGIN_REGISTRAR_H
```

Listing Seven

Next Time

That's it for now. In the next installment, I examine the issues that involved in cross-platform development and the dual C/C++ object model, among other topics.