Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Building Your Own Plugin Framework: Part 3

Cross-platform development, misc topics, and the dual C/C++ object model

December 26, 2007
URL:http://www.drdobbs.com/cpp/building-your-own-plugin-framework-part/205203129

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 third installment of a five-part series by Gigi Sayfan on creating cross-platform plugins in C++. Other installments are: Part 1, Part 2, Part 4, and Part 5.

This is the third article in a series of articles entitled "Building Your Own Plugin Framework" that is about developing cross-platform plugins in C++. The first article described the problem in detail, explored various solutions and introduced the plugin framework. The second article explored the architecture and design of a plugin-based systems based on the plugin framework, the lifecycle of a plugin and the internals of the generic plugin framework. This article covers cross-platform development, miscellaneous topics like platform services provided to plugins by the system, error handling, and design and implementation of a dual C/C++ object model.

Cross-Platform Development

Cross-platform development in C/C++ is hard. Really hard. There are data type differences, compiler differences and OS API differences. The key to cross-platform development is to encapsulate platform differences so your main application code can concentrate on your application's logic. If your application code is bogged down in platform-specific code and has lots of **#ifdef OS_THIS** and **#ifdef OS_THAT**, it's a sure sign you need some refactoring. A good practice is to completely isolate all platform-specific code into a separate library or set of libraries. The ideal is that if you need to support a totally new platform you will need to modify only the code of the platform support library.

Understand your Target Platforms

The first order of business when targeting multiple platforms is to understand and be aware of the differences. If you target 32-bit and 64-bit platforms, you need to understand the ramifications. If you target Windows, you need to be aware of ANSI/MBCS versus Unicode/Wide string. If you target a mobile device with a stripped down OS, you need to know what subset is available for you.

Use a Good Cross-platform Library

The next order of business is to pick a good cross-platform library. There are several good libraries. Most of them focus on UI. I chose to use the Apache Portable Runtime (APR) for the plugin framework. APR is the foundation of the Apache web server, the subversion server and a few other projects.

But APR might not be right for you. It is fully documented, but the documentation is not stellar. There isn't a big thriving community. There are no books and relatively a small number of projects use it. To top it off it's a C library and you might not like the naming conventions. However, it is very portable and robust (at least the parts used by Apache and Subversion) and you know it can be used to implement high-performance systems.

Consider writing a custom wrapper to your cross-platform library (just the parts you use). There are several benefits to this approach:

- You can modify the interface to match your needs exactly
- The naming conventions will match the rest of your code
- It will make it much easier to switch to a different library or even upgrade to a new version of the same library.

There are also a disadvantages:

- You have to invest time and resources in writing and maintaining your wrapper
- Debugging could be a little more difficult because you have to go through another layer (not to mention that your wrapper may be buggy).

I chose to write a wrapper for APR because it is a C library that require that you release resources explicitly, deal with memory pools for efficient memory allocation and I didn't like the naming conventions and the entire feel. I also used a relatively small subset (just directory and file APIs), but it was still a significant work to do it, test it and debug it. You can see the Path and Directory classes in the plugin_framework subdirectory. Note, that I use the APR typedefs for basic types as is without wrapping them in my own typedefs. This is just out of laziness and not a recommended practice. You can see the APR types all over the place in structs and interfaces.

Data Type Differences

The integral types C++ inherited from C are a cross-platform hazard. int, long and friends have different sizes on different platforms (32-bit and 64-bit on

today's systems, maybe 128-bit later). For some applications it might seem irrelevant because they never approach the 32-bit limit (or rather 31-bit if you use unsigned integers), but if you serialize your objects on a 64-bit system and deserialize on a 32-bit system you might be unpleasantly surprised. Again, no easy breaks here. You need to understand the issues and do the right thing, which is make sure the sending/saving side and the receiving/loading side agree on the number of bytes of each value. File format or network protocol designers must be smart about it.

APR provides a set of typedefs for basic types that might be different on different platforms. These typedefs provide a guaranteed size and avoid the fuzzy built-in types. However, for some applications (mostly numerical) it is sometimes important to use the native machine word size (typically what **int** stands for) to achieve maximal performance.

Wrap Platform-specific Components In Cross-platform Wrappers

Sometimes you must write larger chunks of platform-specific code. For example, APR's support for dynamic libraries wasn't adequate for the plugin framework needs. I simply implemented the **DynamicLibrary** class, which is a cross-platform abstraction of the dynamic library concept with a neutral interface. It is not always possible to do it without losing expressiveness or performance. You need to make your own trade-offs and expose to the application what you must. In the case of **DynamicLibrary** I preferred a minimal interface that doesn't allow the application to specify any flags to **dlopen()** on Unix and I used the simpler **LoadLibrary()** on Windows in lieu of the more flexible **LoadLibraryEx()**.

Organize All Third-party Dependencies

Today, it's common practice to reuse third-party code. There are many good libraries with liberal licenses. Your project is likely to use a few of them too. In cross-platform environment it is important to select the third-party libraries you use wisely. In addition to standard selection criteria like robustness, performance, ease of use, documentation and support, you need to pay attention to the way the library is developed and maintained. Some libraries have not been developed in a cross-platform fashion from the beginning. It is common to see libraries that are used mostly on one platform and ported as an afterthought to other platforms. If the code base is not unified, it is a red flag. If there is a small number of users on a particular platform it is a red flag. If the core developers or maintainers don't don't package the library for all platforms it is a red flag. If when new version comes out, some platforms lag behind it is a red flag. A red flag doesn't mean you shouldn't use the library. It just means that all other things being equal you should prefer a library without red flags.

Invest in the Build System

A good automated build system is crucial for the development of non-trivial software systems. If you throw in several flavors of your system (Standard, Pro, Enterprise), a couple of platforms (Windows, Linux, Mac OS X) and a few build variants (Debug, Release) you get an exponential explosion of artifacts. The build system must be fully automated and support the full build lifecycle -- get the source code from the source control system, do any pre-processing, compile, link, run unit tests and integration tests, package and distribute, possibly run full system tests, and report the results to all the stakeholders.

I'm a little fanatic about build systems and automation, but you won't be sorry for investing in your build system.

Platform Services

Platform services are services provided to the plugins by your system or application. I call them "platform services" because the generic plugin framework can service as a platform for plugin-based systems. The **PF_PlatformServices struct** contains the version, the **registerObject** function pointer and the **invokeService** function pointer; see Example 1.

'

```
typedef apr_int32_t (*PF_RegisterFunc)(const apr_byte_t * nodeType, const PF_RegisterParams * params);
typedef apr_int32_t (*PF_InvokeServiceFunc)(const apr_byte_t * serviceName, void * serviceParams);
typedef struct PF_PlatformServices
{
  PF_PluginAPI_Version version;
  PF_RegisterFunc registerObject;
  PF_InvokeServiceFunc invokeService;
} PF_PlatformServices;
```

Example 1

The version lets the plugin know the version of **PluginManager** it is hosted in. It lets the plugin make version-specific decisions like register different types of objects depending on the version of the host.

The **registerObject()** function is a **PluginManager** service through which the plugin registers its objects (without it there will be no plugin system).

The **invokeService** function is for application-specific services. The normal interaction between the application and the plugins (once the plugin objects have been created) is that the application invokes plugin object methods through the object model interfaces (e.g., **IActor::play()**). But, it is often desirable for a plugin to request some service from the application. It happens a lot when the application provides some managed execution environment where objects log progress, report errors or allocate memory in a centralized way. The application typically provides a standard logger, error reporting function, and memory allocator and all the objects use them. These service objects are often singletons or static functions and methods. Dynamic plugins can't access them directly. Unlike the **registerObject** service, the application-specific can't be defined in the generic **PF_PlatformServices struct**, because they are not known and to the generic plugin framework and they will be different for different applications. The application may wrap all these service access points and define a big struct that contains all of them and pass it to each plugin object through an object model interface method (e.g., **initObject()**) that every plugin object must implement. This is not always convenient in the presence of C plugins. The service objects are most likely implemented in C++ and accept and return C++ objects as arguments, maybe they are templated and maybe they throw exceptions. It is possible to provide a C compatible wrapper for each one. However, often it is easier to funnel all the service requests through a single sink that handles all the interaction with the plugins.

This is what **invokeService()** is all about. The signature is very simple -- a string for the service name and a void pointer to arbitrary struct. This is a weakly typed interface, but it provides absolute flexibility. The plugin and the application must coordinate the available services and what the **params struct** should be. Example 2 demonstrates a logging service that accepts a filename, line number, and message and logs.

```
LogServiceParams.h
==================
typedef struct LogServiceParams
{
  const apr_byte_t * filename;
  apr_uint32_t       line;
  const apr_byte_t * message;
} LogServiceParams;


Some Application File...
========================

#include "LogServiceParams.h"

apr_int32_t InvokeService(const apr_byte_t * serviceName,
                          void * serviceParams)
{
  if (::strcmp(serviceName, "log") == 0)
  {
        LogServiceParams * lsp = (LogServiceParams *)serviceParams;
        Logger::log(lsp->filename, lsp->line, lsp->message);
  }
}
```

Example 2

The **LogServiceParams struct** is defined in a header file that both the plugin and the application **#include**. This provides the logging protocol between them. The plugin packs the current filename, line number and log message in the structs and calls the **invokeService()** function with "log" as the service name and a pointer to the struct (as a **void ***). The implementation of the **invokeService()** function on the application side gets the pointer to the struct as a void pointer casts it to **LogServiceParams struct** and then calls the **Logger::log()** method with the information. If the plugin doesn't send a proper **LogServiceParams struct** the behavior is undefined (but definitely bad). The **invokeService()** can be used to handle multiple service requests and can let the plugin know by returning -1 that it failed. If the application needs to return a result to the plugin, output variable can be added to the service **params struct**. Each service may have its own **params struct**.

For example if the application wants to control memory allocation because it uses a custom memory allocation scheme it can provide an "allocate" and "deallocate" service. Whenever the plugin needs to allocate memory instead of doing **malloc** or **new** on its own it will call the "allocate" service and the **AllocateServiceParams struct** will contain the requested size and an output **void *** (or **char ***) for the allocated buffer.

Error Handling

Error handling is a little different with C++ plugin-based systems. You can't just throw exceptions in your plugin and expect them to be handled by the application. This is part of the binary compatibility problem I discussed in the first article in the series. It may work if the plugin was built using the same compiler exactly as the application, but it's not a restriction I am willing to force on plugin developers. You can always stoop down to C-style error return codes but that goes against the grain of the C++ plugin framework. One of the main design goals of the plugin framework is to allow both plugin developers and the application developers to program in C++ even if under the covers they communicate in C-style functions across the dynamic library boundary.

So, what's needed is a way to intercept exceptions thrown in the plugin, transmit them across the dynamic library boundary in a safe and compiler-agnostic manner to the application and then throwing the exception again on the application side.

The solution I use is to wrap (on the plugin side) every method in a **try-except** block. When an exception is thrown on the plugin side I extract some information and report it to the application through a special **invokeService()** call. On the application side, when the **reportError** service is invoked I store the error information and when the current plugin object method returns I throw the stored exception.

This delayed serialized exception throwing mechanism is not very conventional, but it achieves the semantics of a regular C++ exception if the plugin method doesn't do anything else after invoking the **reportError** service.

Implementing a Dual C/C++ Object Model

This section is maybe the most complicated and the most innovative part of the C++ plugin framework. The dual object model is what allows both C and C++ plugins to coexist and be hosted by the same application and also allow the application itself to be totally unaware of the duality and treat all objects as C++ objects. Unfortunately, the generic plugin framework can't do it automatically for you. I present the design patterns and dive into the dual object model of the sample game and you will have to do it for your application object model.

The basic idea of the dual object model is that every object should be usable through the C interfaces and the C++ interfaces of the application. These interfaces should be almost identical other than language differences. The object model for the game includes the objects:

- **ActorInfo**
- **ActorInfoContainer**
- **Turn**
- **Actor**

ActorInfo is the simplest because it is just a passive **struct** that contains information about actor; see Example 3. The same struct exactly is used by the C

and C++ and it is fully defined in the c_object_model.h file. The other objects are not so simple.

```
typedef struct C_ActorInfo_
{
  apr_uint32_t id;
  apr_byte_t   name[MAX_STR];
  apr_uint32_t location_x;
  apr_uint32_t location_y;
  apr_uint32_t health;
  apr_uint32_t attack;
  apr_uint32_t defense;
  apr_uint32_t damage;
  apr_uint32_t movement;
} C_ActorInfo;
```

Example 3

The **ActorInfoContainer** is a full-fledged dual C/C++ object; see Listing One.

```
#ifndef ACTOR_INFO_CONTAINER_H
#define ACTOR_INFO_CONTAINER_H
#include "object_model.h"
#include <vector>
struct ActorInfoContainer :
  IActorInfoIterator,
  C_ActorInfoIterator
{
  static void reset_(C_ActorInfoIteratorHandle handle)
  {
    ActorInfoContainer * aic = reinterpret_cast<ActorInfoContainer *>(handle);
    aic->reset();
  }
  static C_ActorInfo * next_(C_ActorInfoIteratorHandle handle)
  {
    ActorInfoContainer * aic = reinterpret_cast<ActorInfoContainer *>(handle);
    return aic->next();
  }
  ActorInfoContainer() : index(0)
  {
    C_ActorInfoIterator::handle = (C_ActorInfoIteratorHandle)this;
    C_ActorInfoIterator::reset = reset_;
    C_ActorInfoIterator::next = next_;
  }
  void reset()
  {
    index = 0;
  }
  ActorInfo * next()
  {
    if(index >= vec.size())
      return NULL;
    return vec[index++];
  }
  apr_uint32_t index;
  std::vector<ActorInfo *> vec;
};
#endif
```

Listing One

I will now dissect it almost line by line so pay attention. Its job is pretty simple. It provides forward-only iteration over a collection (**std::vector**) of immutable **ActorInfo** objects. It also allows resetting it's internal pointer to the beginning of the collection, so multiple passes are possible. The interface has a **next**() method that returns a pointer to the current object (**ActorInfo**) and advances the internal pointer to the next object or NULL if it has already returned the last object. The first call will return the first object or NULL if the collection empty. The semantics are different than STL iterators where the iterator just advances and you need to explicitly dereference it to get to the underlying object. Also STL iterators can point to value objects and the indicator of end of collection is if the iterator equals to the **end**() iterator of the collection. There are several reasons I use a different iteration interface than the STL interface. STL iterators support many more styles of iteration with differences nuances than just forward iteration over an immutable collection. As a result, they are more complicated to use and require more code. The main reason however is that the iteration interface of plugin objects should support C interfaces (see Example 4). Finally, I like the fact that NULL result indicates end of collection and that I don't have to dereference the iterator to get to the object. It lets me write really compact code to iterate over collections.

Back to **ActorInfoContainer**, it subclasses both **IActorInfoIterator** and **C_ActorInfoIterator** and that's what makes it a dual C/C++ object; see Example 4.

```
struct ActorInfoContainer :
  IActorInfoIterator,
  C_ActorInfoIterator
{
 ...
};
```

Example 4

It needs to implement both interfaces of course. The C++ interface (see Example 5) is your typical ABC (abstract base class) where all the member functions (**next**() and **reset**()) are pure virtual.

```
struct IActorInfoIterator
{
  virtual void reset() = 0;
  virtual ActorInfo * next() = 0;
};
```

Example 5

The C interface (see Example 6) has an opaque handle, which is just a pointer to a dummy **struct** that contains a single character and it has two function pointers for **next**() and **release**() that accepts as a first argument the handle.

```
typedef struct C_ActorInfoIteratorHandle_ { char c; } * C_ActorInfoIteratorHandle;
typedef struct C_ActorInfoIterator_
{
  void (*reset)(C_ActorInfoIteratorHandle handle);
  C_ActorInfo * (*next)(C_ActorInfoIteratorHandle handle);
  C_ActorInfoIteratorHandle handle;
} C_ActorInfoIterator;
```

Example 6

ActorInfoContainer manages a vector of **ActorInfo** objects and it implements the C++ **IActorInfoIterator** interface by keeping an index into its vector of ActorInfo objects; see Example 7. When **next()** is called it returns the object in the current index in the array or NULL if the index is greater than the vector size. When **reset()** is called it simply sets the index to 0. The index is initialized to 0, of course.

```
struct ActorInfoContainer :
  IActorInfoIterator,
  C_ActorInfoIterator
{
  ...
  ActorInfoContainer() : index(0)
  {
    ...
  }
  void reset()
  {
    index = 0;
  }
  ActorInfo * next()
  {
    if (index %gt;= vec.size())
      return NULL;
    return vec[index++];
  }
  apr_uint32_t index;
  std::vector<ActorInfo *> vec;
};
```

Example 7

It implements the C interface by populating **C_ActorInfoIterator struct**. In the constructor it assigns the **reset_()** and **next_()** static methods to the **reset** and **next** function pointers in the **C_ActorInfoIterator** base **struct**. It also assigns the **this** pointer to the handle; see Example 8.

```
static void reset_(C_ActorInfoIteratorHandle handle)
  {
    ActorInfoContainer * aic = reinterpret_cast<ActorInfoContainer *>(handle);
    aic->reset();
  }
  static C_ActorInfo * next_(C_ActorInfoIteratorHandle handle)
  {
    ActorInfoContainer * aic = reinterpret_cast<ActorInfoContainer *>(handle);
    return aic->next();
  }
  ActorInfoContainer() : index(0)
  {
    C_ActorInfoIterator::handle = (C_ActorInfoIteratorHandle)this;
    C_ActorInfoIterator::reset = reset_;
    C_ActorInfoIterator::next = next_;
  }
```

Example 8

This is the time to unveil the inner workings of the dual object. The actual implementation of each dual object is always in the C++ part of each dual object. The C function pointers always point to static methods of the C++ object that delegate the work to the corresponding methods of the C++ interface. This is the tricky part. Although the C and the C++ interfaces are the "parents" of **ActorInfoContainer** there is no portable way in C++ to get from one base class to another base class. To do that the static C functions need an access to the **ActorInfoContainer** instance (the "child"). This is where the handle comes in handy (pun intended). Each static C method casts the handle to **ActorInfoContainer** pointer (using **reinterpret_cast**) and calls the corresponding C++ method. The **reset()** method accepts no arguments and return nothing. The **next()** method accepts no arguments and returns **ActorInfo** pointer, which is the same return type for the C and C++ interfaces.

The situation is a little more complicated when it comes to the Turn dual object. This object implements the **ITurn** C++ interface (see Example 9) and the **C_Turn** C interface (see Example 10).

```
struct ITurn
{
  virtual ActorInfo * getSelfInfo() = 0;
  virtual IActorInfoIterator * getFriends() = 0;
  virtual IActorInfoIterator * getFoes() = 0;
  virtual void move(apr_uint32_t x, apr_uint32_t y) = 0;
  virtual void attack(apr_uint32_t id) = 0;
};
```

Example 9

```
typedef struct C_TurnHandle_ { char c; } * C_TurnHandle;
typedef struct C_Turn_
{
  C_ActorInfo * (*getSelfInfo)(C_TurnHandle handle);
  C_ActorInfoIterator * (*getFriends)(C_TurnHandle handle);
  C_ActorInfoIterator * (*getFoes)(C_TurnHandle handle);
  void (*move)(C_TurnHandle handle, apr_uint32_t x, apr_uint32_t y);
  void (*attack)(C_TurnHandle handle, apr_uint32_t id);
  C_TurnHandle handle;
} C_Turn;
```

Example 10

The **Turn** object follows in the footsteps of **ActorInfoContainer** and has static C methods that are hooked up to the function pointers of the **C_Turn** interface in the constructor and delegate the work to the C++ methods. Let's focus on the **getFriends()** method. This method is supposed to return **IActorInfoIterator** from the C++ **ITurn** interface and **C_ActorInfoIterator** from the **C_Turn** interface. A different return value type. What a conundrum! The static **getFriends_()** can't just return the result of calling **getFriends()**, which is **IActorInfoIterator** pointer and it can't just do **reinterpret_cast** or C cast to **C_ActorInfoIterator** because the offset of the **C_Turn** base struct is different. The solution is to use a little inside information. The result of **ITurn::getFriends()** is indeed **IActorInfoIterator**, but actually it returns **ActorInfoContainer** dual object, which implements both **IActorInfoIterator** and **C_ActorInfoIterator**.

In order to get from **IActorInfoIterator** to **C_ActorInfoIterator**, **getFriends_()** performs up-casting to the **ActorInfoContainer** dual C/C++ object (using **static_cast<ActorInfoContainer>**). Once, it has an **ActorInfoContainer** instance it can serve as **C_ActorInfoIterator**. It's okay to take a sip of water or something stronger now. You earned it.

The core idea is that the entire object model is implemented in terms of dual C/C++ objects that can be used through either the C or C++ interfaces (with some nudging and casting). It's also okay to use basic types and C structs like **ActorInfo** that can be used trivially in C and C++. Note that all this mind-boggling stuff is safely entombed inside the application object model implementation. The rest of the application code and the plugin code don't have to deal with multi-language multiple inheritance, nasty casts and switching from one interface to another through the derived class. This design pattern/idiom is admittedly convoluted, but once you the get a grip on it you can see it simply repeats all over the object model.

We are not done yet. I lied. Just two sentences ago I said that this weird dual C/C++ pattern repeats all over the place. Well, almost. When it comes to the **IActor** (see Example 11) and **C_Actor** (see Example 12) interfaces this is not the case. These interfaces represent the actual plugin objects that are created using the **PF_CreateFunc**. There is no dual **Actor** object that implements both **IActor** and **C_Actor**.

```
struct IActor
{
  virtual ~IActor() {}
  virtual void getInitialInfo(ActorInfo * info) = 0;
  virtual void play(ITurn * turnInfo) = 0;
};
```

Example 11

```
typedef struct C_ActorHandle_ { char c; } * C_ActorHandle;
typedef struct C_Actor_
{
  void (*getInitialInfo)(C_ActorHandle handle, C_ActorInfo * info);
  void (*play)(C_ActorHandle handle, C_Turn * turn);
  C_ActorHandle handle;
} C_Actor;
```

Example 12

The objects that implement **IActor** and **C_Actor** come from the plugins. They are not part of the application object model. they are users of the application object model. Their interfaces are just defined in the application's object model header files (object_model.h and c_object_model.h). Each plugin object implements either the C++ **IActor** interface or the C **C_Actor** interface (and they were registered accordingly with the **PluginManager**). The **PluginManager** will adapt C objects that implement the **C_Actor** interface to **IActor**-based adapted C++ object and the application will remain ignorant.

In the next installment I will discuss writing plugins. I'll go over the sample application and its plugins. I'll also give a quick tour of source code (there's a lot of it).