

Building Your Own Plugin Framework

A cross-platform plugin framework for C/C++

Dr. Dobb's - November 25, 2007

Gigi Sayfan

Author

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).

URL

<https://www.drdobbs.com/cpp/building-your-own-plugin-framework-part/204202899>

Part 1

This article is the first in a series of articles that discuss the architecture, development, and deployment of a C/C++ cross-platform plugin framework. This first installment explores the terrain, surveys (briefly) several existing plugin/component libraries, delves into the binary compatibility problem, and presents some desirable properties of the solution.

Subsequent articles showcase an industrial strength plugin framework that you can use on Windows, Linux, Mac OS X, and easily port to other operating systems. The plugin framework has some unique properties compared to other alternatives and it is designed to be flexible, efficient, easy for application developers, easy for plugin writers, supports both C and C++, and provides multiple deployment options (as dynamic or static libraries).

I will develop as a sample application a simple role-playing game that lets you add non-player characters (NPCs) plugins. The game engine loads the

plugins and seamlessly integrates their contents. The game demonstrates the concepts and shows concrete running code.

Who Needs Plugins?

Plugins are the way to go if you want to develop a successful and dynamic system. Plugin-based extensibility is the current best practice to extend and evolve systems in a safe manner. Plugins let third-party developers add value to systems and let in-house developers add functionality without risk of destabilizing the core functionality. Plugins promote separation of concerns, guaranty implementation details hiding, isolated testing, and many other best practices.

Platforms like Eclipse are actually bare-bones frameworks where all the functionality is provided by plugins. The Eclipse IDE itself (including the UI and the Java Development Environment) is just a set of plugins hooked into the core framework.

Why C++?

C++ is notoriously non-accommodating when it comes to plugins. C++ is extremely platform-specific and compiler-specific. The C++ standard doesn't specify an Application Binary Interface (ABI), which means that C++ libraries from different compilers or even different versions of the same compiler are incompatible. Add to that the fact that C++ has no concept of dynamic loading and each platform provide its own solution (incompatible with others) and you get the picture. There are a few heavyweight solutions that try to address more than just plugins and rely on some additional runtime support. Still, C/C++ is often the only practical option when it comes to high-performance systems.

What's Out There?

Before embarking on a brand new framework, it's worth checking out existing libraries or frameworks. I found that there are either heavyweight solutions like Microsoft's COM and Mozilla's XPCOM (Cross-platform COM), or pretty basic offerings like Qt's plugins and a few articles about creating C++ plugins. One interesting library is [DynObj](#) that claims to solve the binary compatibility problem (with some restrictions). There is also a proposal for adding plugins to C++ as a native concept by [Daveed Vandervoorde](#). It's an interesting read, but it feels strange.

None of the basic solutions address the myriad issues associated with creating an industrial strength plugin-based system like error handling, data types, versioning, separation between framework code, and application code. Before diving into the solution, let's understand the problem.

The Binary Compatibility Problem

Again, there is no standard C++ ABI. Different compilers (and even different versions of the same compiler) produce different object files and libraries. The most obvious manifestation of this problem is the different name mangling algorithms implemented by different compilers. This means that in general you can only link C++ object files and libraries that were compiled using exactly the same compiler (brand and version). Many compilers don't even implement standard C++ features from the C++98

There are some partial solutions to this problem. For example, if you access a C++ object only through a virtual pointer and call only its virtual methods you sidestep the name mangling issue. However, it is not guaranteed that even the virtual table layout in memory is identical between compilers, although it is more stable.

If you try to load C++ code dynamically you face another issue -- there is no direct way to load and instantiate C++ classes from a dynamic library under Linux or Mac OS X (Visual C++ supports it under Windows).

The solution to this issue is to use a function with C linkage (not name mangled by the compiler) as a factory function that returns an opaque handle to the caller. The caller then casts the handle to the appropriate class (usually a pure abstract base class). This requires some coordination, of course, and works only if the library and the application were compiled with compilers that have a matching vtable layout in memory.

The ultimate in compatibility is to just forget about C++ and expose a pure C API. C is compatible in practice between all compiler implementations. Later I'll show how to achieve C++ programming model on top of C compatibility.

Plugin-Based System Architecture

A plugin-based system can be divided into three parts:

- The domain-specific system.
- A plugin manager.
- The plugins.

The domain-specific system loads the plugins and creates plugin objects via the plugin manager. Once a plugin object is created and the main system has some pointer/reference to it, it can be used just like any other object. Usually, there are some special destruction/cleanup requirements as we shall see.

The plugin manager is a pretty generic piece of code. It manages the life-cycle of the plugins and exposes them to the main system. It can find and load plugins, initialize them, register factory functions and be able to unload plugins. It should also let the main system iterate over loaded plugins or registered plugin objects.

The plugins themselves should conform to the plugin manager protocol and provide objects that conform to the main system expectations.

In practice, you rarely see such a clean separation (in C++-based plugin systems, anyway). The plugin manager is often tightly coupled with the domain-specific system. There is a good reason for that. Plugin managers need to provide eventually instances of plugin objects with certain type. Moreover, the initialization of the plugin often requires passing domain-specific information and/or callback functions/services. This can't be done easily with a generic plugin manager.

Plugin Deployment Models

Plugins are usually deployed as dynamic libraries. Dynamic libraries allow many of the advantages of plugins such as hot swapping (reloading a new implementation without shutting the system), safe extension by third-party developers (additional functionality without modifying the core system), and shorter link times. However, there are situations where static libraries are the best vehicle for plugins. For example, some systems simply don't support dynamic libraries (many embedded systems). In other cases, security concerns don't allow loading foreign code. Sometimes, the core system comes with pre-loaded with some plugins and it is more robust to statically link them to the main executable (so the users can't delete them by accident).

The bottom line is that a good plugin system should support both dynamic and static plugins. This lets you deploy the same plugin-based system in different environments with different constraints.

Plugin Programming Interface

Plugins are all about interfaces. The basic notion of plugin-based system is that there is some central system that loads plugins it knows nothing about and communicates with them through well-defined interfaces and protocols.

The naive approach is to define a set of functions as the interface that the plugin exports (either dynamic or static library). This approach is technically possible but conceptually it is flawed. The reason is that there are two kinds of interfaces a plugin should support and can be only a single set of functions exported from a plugin. This means that both kinds of interface will be mixed together.

The first interface (and protocol) is the generic plugin interface. It lets the central system initialize the plugin, and lets the plugin register with the central system various functions for creating and destroying objects as well as global cleanup function. The generic plugin interface is not domain-specific and can be specified and implemented as a reusable library. The second interface is the functional interface implemented by the plugin objects. This interface is domain-specific and must be carefully designed and implemented by the actual plugins. The central system should be aware of this interface and interact with the plugin objects through it.

Listing One is the header file that specifies the generic plugin interface. Without delving into the details and explaining anything just yet let's just see what it offers.

```
#ifndef PF_PLUGIN_H
#define PF_PLUGIN_H

#include <apr-1/apr_general.h>

#ifdef __cplusplus
extern "C" {
#endif

typedef enum PF_ProgrammingLanguage
{
```

```
PF_ProgrammingLanguage_C,  
  
PF_ProgrammingLanguage_CPP  
} PF_ProgrammingLanguage;  
  
struct PF_PlatformServices_  
  
typedef struct PF_ObjectParams  
{  
  
    const apr_byte_t * objectType;  
  
    const struct PF_PlatformServices_ * platformServices;  
} PF_ObjectParams;  
  
typedef struct PF_PluginAPI_Version  
{  
  
    apr_int32_t major;  
  
    apr_int32_t minor;  
} PF_PluginAPI_Version;  
  
typedef void * (*PF_CreateFunc) (PF_ObjectParams *);  
  
typedef apr_int32_t (*PF_DestroyFunc) (void *);  
  
typedef struct PF_RegisterParams  
{  
  
    PF_PluginAPI_Version version;
```

```

PF_CreateFunc createFunc;

PF_DestroyFunc destroyFunc;

PF_ProgrammingLanguage programmingLanguage;

} PF_RegisterParams;


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;


typedef apr_int32_t (*PF_ExitFunc)();

typedef PF_ExitFunc (*PF_InitFunc)(const PF_PlatformServices *);


#ifndef PLUGIN_API

#ifdef WIN32

#define PLUGIN_API __declspec(dllimport)

#else

#define PLUGIN_API

#endif

```

```

#endif

#endif

extern

#ifdef __cplusplus

"C"

#endif

PLUGIN_API PF_ExitFunc PF_initPlugin(const PF_PlatformServices * params);

#ifdef __cplusplus

}

#endif

#endif /* PF_PLUGIN_H */

```

Listing One

The first thing you should notice is that it is a C file. This allows the plugin framework to be compiled and used by pure C systems and to write pure C plugins. But, it is not limited to C and is actually designed to be used mostly from C++.

The **PF_ProgrammingLanguage** enum allows plugins to declare to the plugin manager if they are implemented in C or C++.

The **PF_ObjectParams** is an abstract struct that is passed to created plugin objects.

The **PF_PluginAPI_Version** is used to negotiate versioning and make sure that the plugin manager loads only plugins with compatible version.

The functions pointer definitions **PF_CreateFunc** and **PF_DestroyFunc** (implemented by the plugin) allow the plugin manager to create and destroy plugin objects (each plugin registers such functions with the plugin manager)

The **PF_RegisterParams** struct contains all the information that a plugin must provide to the plugin manager upon initialization (version, create/destroy functions, and programming language).

The **PF_RegisterFunc** (implemented by the plugin manager) allows each plugin to register a **PF_RegisterParams** struct for each object type it supports. Note that this scheme allows a plugin to register different versions of an object and multiple object types.

The **PF_InvokeService** function pointer definition is a generic function that plugins can use to invoke services of the main system like logging, event notification and error reporting. The signature includes the service name and an opaque pointer to a parameters struct. The plugins should know about available services and how to invoke them (or you can implement service discovery if you wish using **PF_InvokeService**).

The **PF_PlatformServices** struct aggregates all the services I just mentioned that the platform provides to plugin (version, registering objects and the invoke service function). This struct is passed to each plugin at initialization time.

The **PF_ExitFunc** is the definition of the plugin exit function pointer (implemented by the plugin).

The **PF_InitFunc** is the definition of the plugin initialization function pointer.

The **PF_initPlugin** is the actual signature of the plugin initialization function of dynamic plugins (plugins deployed in dynamically linked libraries/shared libraries). It is exported by name from dynamic plugins, so the plugin manager will be able to call it when loading the plugin. It accepts a pointer to a **PF_PlatformServices** struct, so all the services are immediately available upon initialization (this is the right time to register objects) and it returns a pointer to an exit function.

Note that static plugins (plugins implemented in static libraries and linked directly to the main executable) should implement an **init** function with C linkage too, but MUST NOT name it **PF_initPlugin**. The reason is that if there are multiple static plugins, they will all have a function with the same name and your compiler will hate it.

Static plugins initialization is different. They must be initialized explicitly by the main executable that will call their initialization function with the **PF_InitFunc** signature. This is unfortunate because it means the main executable needs to be modified whenever a new static plugin is

added/removed and also the names of the various **init** functions must be coordinated.

There is a technique called "auto-registration" that attempts to solve the problem. Auto-registration is accomplished by a global object in the static library. This object is supposed to be constructed before the **main()** even starts. This global object can request the plugin manager to initialize the static plugin (passing the plugin's **init()** function pointer). Unfortunately, this scheme doesn't work in Visual C++.

Writing a Plugin

What does it mean to write a plugin? The plugin framework is very generic and doesn't provide any tangible objects your application can interact with. You must build your application object model on top of the plugin framework. This means that your application (that loads the plugins) and the plugins themselves will have to agree about and coordinate their interaction model. Usually it means that the application expects the plugin to provide certain types of objects that expose some specific API. The plugin framework will provide all the infrastructure necessary to register, enumerate and load those objects. Example 1 is a definition of a C++ interface called **IActor**. It has two operations -- **getInitialInfo()** and **play()**. Note that this interface is not sufficient because **getInitialInfo()** expects a pointer to a struct called **ActorInfo** and **play()** expects a pointer to yet another interface called **ITurn**. This is usually the case and you must design and specify a whole object model.

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

Example 1

Each plugin can register multiple types that implement the **IActor** interface. When the application decides to instantiate an object registered by a plugin, it invokes the registered **PF_CreateFunc** implemented by the plugin. The plugin is responsible to create a corresponding object and return it to the application. The return type is **void *** because the object creation operation is part of the generic plugin framework that knows nothing about the specific **IActor** interface. The application then casts the **void *** to the an **IActor *** and can work with it through the interface as if it was a regular object. When the application is done with the **IActor** object it invokes the

registered **PF_DestroyFunc** implemented by the plugin and the plugin destroys the actor object. Pay no attention to the virtual destructor behind the curtain. I'll discuss it in the next installment.

Programming Language Support

In the binary compatibility section I explained that you can have C++ vtable-level compatibility if you use compilers with matching vtable layouts for the application and the plugins or you can use C-level compatibility and then you can use different compilers to build the application and the plugins, but you are limited to C interaction. Your application object model must be C-based. you can't use a nice C++ interface like `<b<iactor< b="">in` Example 1, but you must devise a similar C interface.`</b<iactor<>`

Pure C

In pure C programming model you simply develop your plugin in C. When you implement the **PF_CreateFunc** function you return a C object that interacts with further C object in your application C object model. What is all this talk about C objects and C object models. Everybody knows C is a procedural language and has no concept of objects. This is correct and still C has enough abstraction mechanism to implement objects including polymorphism (which is necessary in this case) and support object-oriented programming style. In fact, the original C++ compiler was actually a front-end to a C compiler. It produced C code from the C++ code that was later compiled using a plain C compiler. It's name [Cfront](#) is more than telling.

The ticket is to use structs that contain function pointers. The signature of each function should accept its own struct as first argument. The struct may also contain other data members. This simple idiom corresponds to a C++ class and provides encapsulation (state and behavior in one place), inheritance (by using the first data member for a base struct), and polymorphism (by setting different function pointers).

C doesn't support destructors, function, and operators overloading and namespaces so you have fewer options when defining interfaces. That may be a blessing in disguise because interfaces are supposed to be used by other people who may master a different subset of the C++ language. Reducing the scope of language construct in interfaces may improve the simplicity and usability of your interfaces.

I will explore object-oriented C in the context of the plugin framework in the follow up articles. Listing Two contains the C object model of the sample game that accompanies this article series (just to whet your appetite).

If you take a quick look you can see that it even supports a form of collections and iterators beyond plain objects.

```
#ifndef C_OBJECT_MODEL
#define C_OBJECT_MODEL

#include <apr-1/apr.h>

#define MAX_STR 64 /* max string length of string fields */

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;

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;


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;


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;

#endif

```

Listing Two

Pure C++

In pure C++ programming model you simply develop your plugin in C++. The plugin programming interface functions can be implemented as static member functions or as plain static/global functions (C++ is mostly a superset of C after all). The object model can be your garden variety C++ object model. Listing Three contains the C++ object model of the sample game. It is almost identical to the C object model of Listing Two.

```

#ifndef OBJECT_MODEL
#define OBJECT_MODEL

#include "c_object_model.h"

typedef C_ActorInfo ActorInfo;

struct IActorInfoIterator
{
    virtual void reset() = 0;

    virtual ActorInfo * next() = 0;
};

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;

};


struct IActor

{

    virtual ~IActor() {}

    virtual void getInitialInfo(ActorInfo * info) = 0;

    virtual void play( ITurn * turnInfo) = 0;

};


#endif

```

Listing Three

Dual C/C++

In the dual C/C++ programming model you can develop your plugin in either C or C++. When you register your objects you specify if they are C or C++ object. This is useful if you create a platform and you want to provide third-party developers ultimate freedom to choose their programming language and programming model and mix and match C and C++ plugins.

The plugin framework supports it, but the real work is in devising a dual C/C++ object model to your application. Each object type needs to implement both C interface and the C++ interface. This means that you will have a C++ class with a standard vtable and also a bunch of function

pointers that correspond to the methods of the virtual table. The mechanics are not trivial and I'll demonstrate it in the context of the sample game.

Note that from the point of view of a plugin developer the dual C/C++ model doesn't introduce any additional complexity. The plugin developer always develop either a C or a C++ plugin using C interfaces or the C++ interfaces.

Hybrid C/C++

In hybrid C/C++ programming models, you develop your plugin in C++ but under the covers the C object model is used. This involves creating C++ wrapper classes that implement the C++ object model and wrap corresponding C objects. The plugin developers programs against this layer that translates every call, parameter and return value back and forth between C and C++. This requires additional work when implementing your application object model, but is very straight forward usually. The benefit is a nice C++ programming model for the plugin developer with a full C-level compatibility. I don't demonstrate it in the context of the sample game.

Language-Linkage Matrix

Figure 1 shows the various pros and cons of different combinations of deployment models (static vs. dynamic libraries) and programming language choice (C vs. C++).

Language Linkage	C++	C
Static	<ul style="list-style-type: none"> * Best performance * Easy API * Easy debugging * Can use any modern C++ compiler * Only option on systems with no dynamic libraries * Avoid all DLL hell pitfalls * Requires source compilation of the entire system (possibly with obfuscated source code) 	<ul style="list-style-type: none"> * Unused – may be used on systems that support only a reliable C compiler and no dynamic libraries
Dynamic	<ul style="list-style-type: none"> * Good performance * Easy API * Easy debugging * Must use compiler with compatible vtable layout (not full ABI compatibility) * Compatibility risk if new version of application migrates to a compiler with different vtable layout * Requires dynamic library support in target platform 	<ul style="list-style-type: none"> * C/C++ marshaling overhead * Cumbersome API * Cumbersome debugging * Can use any compiler for development * Not sensitive to compiler changes in new versions * Requires dynamic library support in target platform

Figure 1

For the sake of this discussion the dual C/C++ model has the prerequisites and limitations of C++ if using C++ plugins and the prerequisites and limitations of C if using C plugins. Also, the hybrid C/C++ model is just a C model because the C++ layer is hidden behind the plugin implementation. This can all be confusing, but the bottom line is that you have options and the plugin framework allows you to make choices and pick the tradeoffs you feel appropriate to your situation. It doesn't force you to use a specific model and it doesn't aim to lowest common denominator.

Part 2

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;
    // Register KillerBunny
    rp.createFunc = KillerBunny::create;
    rp.destroyFunc = KillerBunny::destroy;
    res = params->registerObject((const apr_byte_t *)"KillerBunny", &rp);
    if (res < 0)
        return NULL;
    // Register 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 arrangement 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 were 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 to 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;

}

};

#endif // ACTOR_FACTORY_H

```

Listing Six

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.

Part 3

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 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 for 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 too. 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 >= 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 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).

Part 4

This is the fourth article in a series about developing cross-platform plugins in C++. In the previous articles -- [Part 1](#), [Part 2](#), and [Part 3](#) -- I examined the difficulties of working with C++ plugins in portable way due to the binary compatibility problem. I then introduced the plugin framework, its design and implementation, explored the life cycle of a plugin, covered cross-platform development and dived into designing object models for use in plugin-based systems (with special emphasis on dual C/C++ objects).

In this installment, I demonstrate how to create hybrid C/C++ plugins where the plugin communicates with the application through a C interface for absolute compatibility, but the developer programs against a C++ interface.

Finally, I introduce the RPG (Role Playing Game) that serves (faithfully) as a sample application that use the plugin framework and hosts plugins. I explain the concept of the game, how and why its interfaces were designed, and finally explore the application object model.

C++ Facade for Plugin Developers

As you recall, the plugin framework supports both C and C++ plugins. C plugins are very portable, but not so easy to work with. The good news for plugin developers is that it is possible to have a C++ programming model with C compatibility. This is going to cost you though. The transition from C to C++ and back is not free. Whenever a plugin method is invoked by the application it arrives as a C function call through the C interface. An elaborate set of C++ wrapper classes (provided by the application for use by plugin developers) will encapsulate the C plugin, wrap every C argument with non-primitive data type in a corresponding C++ type and invoke the C++ method implementation provided by the plugin object. Then the return value (if any) must be converted back from C++ to C and sent through the C interface to the application. This sounds kinda familiar.

Isn't the dual C/C++ model with the automatically adapted C object exactly the same? In a word, "No." The situation here is totally different. The objects in question always come from the application object model. The application instantiated them and it may convert a dual object between its C and C++ interfaces. On the plugin side, you get the C interface and you have no knowledge about the dual object. This knowledge is necessary to cast from one interface to the other. In addition, even if the plugin knew the type of the dual object it wouldn't be enough because the application and the plugin might have been built using different compilers, different memory models, different calling conventions, etc. The physical layout in memory of the same object might be very different. If you can guarantee that the application and the plugins are vtable-compatible just use the direct C++ interface.

C++ Object Model Wrappers

This is a little weird but necessary. You take a perfectly good dual C/C++ object that you have access only to its C interface and then you wrap it in a C++ wrapper with the same interface. The wrapper can be lean or fat, especially when it comes to iterators. The wrapper can keep the C iterator and call it in response to **next()** and **reset()** calls or it can copy the entire collection.

For the sample game I chose the second approach. It is a little more expansive at call time, but if you use the same data again and again then it can actually be faster because you don't have to wrap the result of each iteration (if you iterate multiple times).

Listing One presents the object model wrappers for the demo game.

```
#ifndef OBJECT_MODEL_WRAPPERS_H
#define OBJECT_MODEL_WRAPPERS_H

#include <string>

#include <vector>

#include <map>

#include "object_model.h"

#include "c_object_model.h"

struct ActorInfoIteratorWrapper : public IActorInfoIterator
{
    ActorInfoIteratorWrapper(C_ActorInfoIterator * iter) : index_(0)
    {
        iter->reset(iter->handle);

        // Create an internal vector of ActorInfo objects

        const ActorInfo * ai = NULL;

        while ((ai = iter->next(iter->handle)))

            vec_.push_back(*ai);
    }

    // IActorInfoIteraotr methods

    virtual void reset()
```

```

{

    index_ = 0;

}

virtual ActorInfo * next()

{

    if (index_ == vec_.size())

        return NULL;

    return &vec_[index_++];

}

private:

    apr_uint32_t index_;

    std::vector<ActorInfo> vec_;

};

struct TurnWrapper : public ITurn

{

    TurnWrapper(C_Turn * turn) :

        turn_(turn),

        friends_(turn->getFriends(turn->handle)),

        foes_(turn->getFoes(turn->handle))

    {

    }

    // ITurn methods

    virtual ActorInfo * getSelfInfo()

```

```
{

    return turn_->getSelfInfo(turn_->handle);

}

virtual IActorInfoIterator * getFriends()

{

    return &friends_;

}

virtual IActorInfoIterator * getFoes()

{

    return &foes_;

}

virtual void move(apr_uint32_t x, apr_uint32_t y)

{

    turn_->move(turn_->handle, x, y);

}

virtual void attack(apr_uint32_t id)

{

    turn_->attack(turn_->handle, id);

}

private:

    C_Turn * turn_;

    ActorInfoIteratorWrapper friends_;

    ActorInfoIteratorWrapper foes_;
```

```
};  
  
#endif // OBJECT_MODEL_WRAPPERS_H
```

Listing One

Note that I need to wrap the C interfaces of any object passed to the main interface **C_Actor**, as well as any object passed to its arguments recursively. Luckily (or by design), there aren't too many objects that need to be wrapped. The **ActorInfo** struct is common to both the C and C++ interfaces and needs no wrapping. The other objects are the **C_Turn** object and the **C_ActorInfoIterator** objects. These objects are wrapped by the **ActorInfoIteratorWrapper** and **TurnWrapper** correspondingly. The implementation of wrapper objects is usually pretty simple, but if you have a large number of them it can be tiresome and a maintenance headache. Each wrapper derives from the C++ interface and accepts the corresponding C interface pointer in its constructor. For example, the **TurnWrapper** object derives from the C++ **ITurn** interface and accepts the a **C_Turn** pointer in its constructor. Wrapper objects store their C interface pointer and in the implementation of their methods they usually forward the call to wrapped object via the stored C interface pointer and wrap the result on-the-fly if necessary. In this case **ActorInfoIteratorWrapper** takes a different approach. In its constructor it iterates over the passed in **C_ActorInfoIterator** and stores the **ActorInfo** objects in an internal vector. Later in its **next()** and **reset()** methods it just works with its populated vector. That wouldn't work, of course, if the collection the iterator works with can be modified after construction. This is fine because all the **ActorInfo** collection passed in are immutable. But, it is something to consider and you need to understand your object model and how it is supposed to be used to design intelligent wrappers. The **TurnWrapper** is a little more conservative and forwards calls to **getSelfInfo()**, **attack()**, and **move()** to its stored **C_Turn** pointer. It takes a different approach with the **getFoes()** and **getFriends()** methods. It saves the friends and foes in **ActorInfoIteratorWrapper** data members that it simply returns from calls to **getFriends()** and **getFoes()**. The **ActorInfoIteratorWrapper** objects implement the **IActorInfoIterator** interface, of course, so they have the proper data type required by the C++ **ITurn** interface.

How bad is the performance hit?

It depends. Remember that you may wrap every C type in your object model, but you don't have to. You may opt instead to use some C objects as is. The real overhead comes in if you pass deep nested data structures as arguments

and you decide to wrap each and every one of them. This is exactly the choice I made in a recent project. I had a complicated data structure that involved several maps that contained vectors of some struct. I wasn't worried about the wrapping overhead because this complex data structure was used for initialization only.

The big issue here is if you want the caller to maintain ownership of the data or if you want to copy it and not worry about the memory management strategies of the caller and if the data is mutable or not (which will preclude storing a snapshot). These are general C++ design concerns and are not specific to the object model wrappers.

ActorBaseTemplate

ActorBaseTemplate is the heart of the hybrid approach. The plugin developer just has to derive from it and implement the C++ **IActor** interface and automatically the plugin will communicate with the plugin manager via the C interface and provide full binary compatibility. The plugin developer should never see the C interface or even be aware of it.

This template provides many services to its sub-classes so let's take it slowly. Example 1 contains the declaration of the template.

```
template <typename T, typename Interface=C_Actor>
class ActorBaseTemplate :
    public C_Actor,
    public IActor
{
    ...
};
```

Example 1

There are two template parameters: **T** and **Interface**. **T** is type of the subclass and when you derive from **ActorBaseTemplate** you must provide the type of the derived class to the base class (template). This is an instance of [CRTP](#), the "Curiously Recurring Template Pattern". **Interface** is the interface that the plugin object will use to communicate with the plugin manager. It can be the C++ **IActor** or the C **C_Actor**. By default it is **C_Actor**. You may wonder why is not always **C_Actor**. After all if the plugin object wishes to communicate with the plugin manager using C++ it can just register itself as a C++ object and directly derive from **IActor**. This is good thinking. The reason **AutoBaseTemplate** supports **IActor** too, is to

let you switch effortlessly from C to C++ interfaces. This useful during debugging when you want to skip the whole C wrapper code and also if you want to deploy in a controlled environment and you don't need the full C compatibility. In this case with a flip of a template parameter and you change the underlying communication channel.

ActorBaseTemplate itself derives from both **C_Actor** and **IActor**. It even provides a trivial implementation of **IActor** in case you want to implement only part of the interface. That saves you from declaring empty methods yourself. The **C_Actor** is the critical interface because this is the interface used to communicate with the plugin manager when **Interface=C_Actor**.

Example 2 is the constructor.

```
ActorBaseTemplate() : invokeService_(NULL)
{
    // Initialize the function pointers of the C_Actor base class
    C_Actor::getInitialInfo = staticGetInitialInfo;
    C_Actor::play = staticPlay;
    C_Actor * handle = this;
    C_Actor::handle = (C_ActorHandle)handle;
}
```

Example 2

It accepts no arguments initializes the **invokeService_** function pointer to NULL and goes on to initialize the members of its **C_Actor** interface to point to static functions and the assigns the **this** pointer to the handle. This is very similar to the C/C++ dual object model and indeed it is a dual object except that the actual C++ implementation that does the real work is in the derived class.

Example 3 is the mandatory **PF_CreateFunc** and **PF_DestroyFunc** that are registered with the plugin manager and are invoked to create and destroy instances.

```
// PF_CreateFunc from plugin.h
static void * create(PF_ObjectParams * params)
{
    T * actor = new T(params);
    // Set the error reporting function pointer
    actor->invokeService_ = params->platformServices->invokeService;
```

```

// return the actor with the correct interface

return static_cast<Interface *>(actor);

}

// PF_DestroyFunc from plugin.h

static apr_int32_t destroy(void * actor)

{

    if (!actor)

        return -1;

    delete ActorBaseTemplate<T, Interface>::getSelf(reinterpret_cast<Interface
*>(actor));

    return 0;

}

```

Example 3

They are named **create()** and **destroy()** but the names are irrelevant because they are registered and invoked as function pointer and not by name. The fact that **ActorBaseTemplate** defines them saves a lot of headache to aspiring plugin developer. The **create()** function simply creates a new instance of **T** (the derived class) and initializes assigns the **invokeService** function pointer to the **invokeService_** data member. The **destroy()** function casts the **void** pointer it gets to the **Interface** template arguments and then use the **getSelf()** method (will be discussed shortly) to get a properly typed pointer to the **T** derived class. It subsequently calls delete to destroy the instance for good. This is really nice. The plugin developer creates a simple C++ class with a standard constructor (that accepts **PF_ObjectParams**, but it can ignore it) and destructor and the **ActorBaseTemplate** does its magic under the covers and make sure that all the weird static functions will be routed properly to derived class.

Example 4 contains the thrice-overloaded **getSelf()** static method.

```

// Helper method to convert the C_Actor * argument

```



```
// in every method to an ActorBaseTemplate<T, Interface> instance pointer
static ActorBaseTemplate<T, Interface> * getSelf(C_Actor * actor)
{
    return static_cast<ActorBaseTemplate<T, Interface> *>(actor);
}
static ActorBaseTemplate<T, Interface> * getSelf(IActor * actor)
{
    return static_cast<ActorBaseTemplate<T, Interface> *>(actor);
}
static ActorBaseTemplate<T, Interface> * getSelf(C_ActorHandle handle)
{
    return static_cast<ActorBaseTemplate<T, Interface> *>((C_Actor *)handle);
}
}
```

Example 4

There are three overloads for **IActor**, **C_Actor**, and **C_ActorHandle**. The **getSelf()** method just performs a **static_cast** to get from the interface to full dual object as you have seen before. In the case of the handle it just performs a C cast to make it a **C_Actor**. As you saw in the constructor and later again the **ActorBaseTemplate** often gets an Interface or handle when it really needs itself to keep going.

Example 5 contains the static **reportError** method.

```
// Helper method to report errors from a static function
static void reportError(C_ActorHandle handle,
                      const apr_byte_t * filename,
                      apr_uint32_t line,
                      const apr_byte_t * message)
{
    ActorBaseTemplate<T, Interface> * self = ActorBaseTemplate<T,
Interface>::getSelf(handle);
    ReportErrorParams rep;
    rep.filename = filename;
    rep.line = line;
    rep.message = message;
    self->invokeService_((const apr_byte_t *) "reportError", &rep);
}
}
```

Example 5

This is a pure convenience function that forwards the call to the **invokeService** function pointer. It saves the caller from packing its

arguments into the **ReportErrorParams** defined by the application's services.h header and from invoking the service with the "reportError" string. These error reporting conventions are defined by the application service layer and are immaterial to the plugin developer who just wants to churn out plugin objects as fast and easy as possible.

Example 6 contains the implementation of the **C_Actor** interface.

```
// C_Actor functions
static void staticGetInitialInfo(C_ActorHandle handle, C_ActorInfo * info)
{
    ActorBaseTemplate<T, Interface> * self = ActorBaseTemplate<T,
Interface>::getSelf(handle);
    try
    {
        self->getInitialInfo(info);
    }
    catch (const StreamingException & e)
    {
        ActorBaseTemplate<T, Interface>::reportError(handle, (const apr_byte_t
*)e.filename_.c_str(), e.line_, (const apr_byte_t *)e.what());
    }
    catch (const std::runtime_error & e)
    {
        ActorBaseTemplate<T, Interface>::reportError(handle, (const apr_byte_t
*)__FILE__, __LINE__, (const apr_byte_t *)e.what());
    }
    catch (...)
    {
        ActorBaseTemplate<T, Interface>::reportError(handle, (const apr_byte_t
*)__FILE__, __LINE__, (const apr_byte_t *)"ActorBaseTemplate<T,
Interface>::staticGetInitialInfo() failed");
    }
}

static void staticPlay(C_ActorHandle handle, C_Turn * turn)
{
    try
    {
        TurnWrapper tw(turn);
        getSelf((C_Actor *)handle)->play(&tw);
    }
    catch (const StreamingException & e)
    {

```

```

        ActorBaseTemplate<T, Interface>::reportError(handle, (const apr_byte_t
*)e.filename_.c_str(), e.line_, (const apr_byte_t *)e.what());
    }
    catch (const std::runtime_error & e)
    {
        ActorBaseTemplate<T, Interface>::reportError(handle, (const apr_byte_t
*)__FILE__, __LINE__, (const apr_byte_t *)e.what());
    }
    catch (...)
    {
        ActorBaseTemplate<T, Interface>::reportError(handle, (const apr_byte_t
*)__FILE__, __LINE__, (const apr_byte_t *)"ActorBaseTemplate<T,
Interface>::staticPlay() failed");
    }
}

```

Example 6

The implementation of both interface functions is almost identical: **getSelf()**, call the C++ **IActor** implementation in the derived class via the wonders of polymorphism and employ robust error handling. Before I discuss the error handling, pay attention to **staticPlay()** function. It accepts a **C_Turn** interface, wraps it in a **TurnWrapper** and then passes it to the **IActor::play()** method where it will arrive as a C++ **ITurn**. This is what the wrappers are for.

The error handling is another nice feature of **ActorBaseTemplate**. It allows plugin developers again to forget that they are writing a plugin object that must adhere to strict rules (such as not throwing exceptions across the binary compatibility boundary) and just throw exceptions on error. Every call to the derived class (except for the constructor and destructor) is wrapped in these try-except clauses. There is here a chain of exception handler from the most informative to the least informative. The plugin developer may elect to throw the **StreamingException** class defined by plugin framework. This is a nice standalone exception class that contains the location (filename and line number) of thrown exception in addition to an error message. If you want to learn more about **StreamingException**, see [Practical C++ Error Handling in Hybrid Environments](#).

Listing Two contains a few convenient macros for checking and asserting that throw **StreamingException** on failure.

```

#ifndef PF_BASE
#define PF_BASE

```

```

#include "StreamingException.h"
#define THROW throw StreamingException(__FILE__, __LINE__) \
#define CHECK(condition) if (!(condition)) \
    THROW << "CHECK FAILED: '" << #condition << "'"
#ifdef _DEBUG
    #define ASSERT(condition) if (!(condition)) \
        THROW <<"ASSERT FAILED: '" << #condition << "'"
#else
    #define ASSERT(condition) {}
#endif // DEBUG
//-----
namespace base
{
    std::string getErrorMessage();
}
#endif // BASE_H

```

Listing Two

This is very nice for debugging purposes because you the end result is all this information will propagate to the application **invokeService()** implementation via the **reportError()** method. If the plugin developer chose to throw a standard **std::runtime_error** then the error-handling code will extract the error message from the **what()** method, but no meaningful filename and line number will be provided. The **__FILE__** and **__LINE__** macros will report the file and line number of the error handling code in **ActorBaseTemplate** and not the actual location of the error. Finally, the fallback is catching any exception with the elipsis except handler. Here, there isn't even an error message to extract and a generic message that at least records the name of the failed function is provided.

The bottom line is that **ActorBaseTemplate** frees the plugin developer from all the vagaries of implementing a plugin object and allows the developer to concentrate on implementing the object interface in standard C++ (**IActor** in this case) without getting tangled up with strange requirements like defining special static methods for creation and destruction, reporting error through funny function pointers or dealing with any shred of C.

PluginHelper

The **PluginHelper** is yet another helper class that takes the drudge out of writing the plugin glue code. Listing Three is the code.

```

#ifndef PF_PLUGIN_HELPER_H
#define PF_PLUGIN_HELPER_H
#include "plugin.h"
#include "base.h"
class PluginHelper
{
    struct RegisterParams : public PF_RegisterParams
    {
        RegisterParams(PF_PluginAPI_Version v,
                       PF_CreateFunc cf,
                       PF_DestroyFunc df,
                       PF_ProgrammingLanguage pl)
        {
            version=v;
            createFunc=cf;
            destroyFunc=df;
            programmingLanguage=pl;
        }
    };
public:
    PluginHelper(const PF_PlatformServices * params) :
        params_(params),
        result_(exitPlugin)
    {
    }
    PF_ExitFunc getResult()
    {
        return result_;
    }
    template <typename T>
    void registerObject(const apr_byte_t * objectType,
                       PF_ProgrammingLanguage pl=PF_ProgrammingLanguage_C,
                       PF_PluginAPI_Version v = {1, 0})
    {
        RegisterParams rp(v, T::create, T::destroy, pl);
        apr_int32_t rc = params_>registerObject(objectType, &rp);
        if (rc < 0)
        {
            result_ = NULL;
            THROW << "Registration of object type "
                << objectType << "failed. "
                << "Error code=" << rc;
        }
    }

```

```

    }
private:

    static apr_int32_t exitPlugin()
    {
        return 0;
    }
private:
    const PF_PlatformServices * params_;
    PF_ExitFunc result_;
};
#endif // PF_PLUGIN_HELPER_H

```

Listing Three

It is designed to work with plugin object classes that implement the **PF_CreateFunc** and **PF_DestroyFunc** mandatory functions as static methods. That's it. No other requirements. As it happens **ActorBaseTemplate** satisfies this requirement so plugin object classes that derive from **ActorBaseTemplate** are automatically compatible with **PluginHelper**. The **PluginHelper** is designed to be used inside the mandatory **PF_initPlugin()** entry point. You will see it in action in the next article when I cover writing plugins. For now, I'll just go over the services **PluginHelper** makes available to the plugin developer. The job of the entry point function is to register all the plugin object types supported by the plugin and if successful return a function pointer to a **PF_ExitFunc** exit function with a particular signature. If something goes wrong it should return NULL.

The **PluginHelper** constructor accepts a pointer to the **PF_PlatformServices** struct that contains the host system plugin API version and **invokeService** and **registerObject** function pointers and stores them. It also stores in its **result** member the **exitPlugin** function pointer that will be returned if the plugin initialization is successful.

PluginHelper provides the templated **registerObject** method that does most of the work. The **T** template parameter is the object type that you want to register. It should have a **create()** and **destroy()** static methods that conform to **PF_CreateFunc** and **PF_DestroyFunc**. It accepts an object type string and optional programming language (defaults to **PF_ProgrammingLanguage_C**). This method performs a version check to make sure the plugin version is compatible with the host system. If everything is fine it prepares a **RegisterObjectParams** struct and calls the **registerObject()** function and check the result. If the version check or the invocation of the **registerObject** function pointer fail it will report the

error (this is done by the `CHECK` macro if the condition is false), set the `result_` to `NULL` and swallow the exception thrown by `CHECK`. The reason it doesn't let the exception propagate is because `PF_initPlugin` (where `PluginHelper` is supposed to be used) is a C function that should not let exceptions propagate across the binary compatibility boundary. Catching all exceptions in `registerObject` saves the plugin developer the trouble doing it (or worse, forgetting to do it). This is a fine example of the convenience of using the `THROW`, `CHECK`, and `ASSERT` macros. The error message is constructed easily using the streaming operator. No need to allocate buffers, concatenate strings or use `printf`. The resulting `reportError` call will contain the exact location of the error (`__FILE__`, `__LINE__`) without having to explicitly specify it.

Typically, a plugin will register more than one object type. If any object type fails to register the `result_` will be `NULL`. It may be okay for some object types to fail registration. For example, you may register multiple versions of the same object type and one of the versions is not supported anymore by the host system. In this case only this object type will fail to register. The plugin developer may check the value of `result_` after each call to `PluginHelper::registerObject()` and decide if it's fatal or not. If it's a benign failure it may eventually return `PluginHelper::ExitPlugin` after all.

The default behavior is that every failure is fatal and the plugin developer should just return `PluginHelper::getResult()` that will return the value of `result_`, which will be `PluginHelper::ExitPlugin` (if all registrations succeeded) or `NULL` (if any registration failed).

The RPG Game

I love RPG games (Role Playing Games), and being a programmer, I have always wanted to write my own. However, the problem with serious game development is that it takes more than just programming to produce a good game. I worked for Sony Playstation for a while, but I worked on multimedia related projects and not on games. So, I benched my aspirations for a spectacular 100 man-years, 10 bazillion dollars RPG. I did a couple of small shoot-em up and board games and focused on writing looooong articles in various developer journals.

I picked a really stripped down RPG game as the vehicle to showcase the plugin framework. It's not going to amount to much. It is more of a game demo because the main program controls the hero and not the user. The conceptual foundations are sound though and it can definitely be extended. Now, that I have reduced your expectations to zero we can move on.

Concept

The concept of the game is very basic. There is a heroic hero, who is as much brave as he is fearless. This hero has been teleported by a mysterious force to a battle arena over populated with various monsters. The hero must fight and defeat all the monsters to win.

The hero and all the monsters function as actors. Actors are entities that have some attributes such as location in the battlefield, health, and speed. When the health of an actor gets down to 0 (or below) it dies.

The game takes place on a 2-D grid (the battlefield). It is a turn-based game. In each turn the actors get to play. When an actor plays it can move or attack (if it's next to another monster). Each actor has a list of friends and foes. This enables the concepts of parties, clans and tribes. In this game the hero has no friends and all the monsters are his foes.

Designing the Interfaces

The interfaces should support the conceptual framework of course. Actors are represented by the **ActorInfo** struct that contains all their stats. Actors should implement the **IActor** interface that allows the **BattleManager** to get their initial stats and to instruct them to play. The **ITurn** interface is what an actor gets when it's their turn to play. The **ITurn** interface allows the actor to get its own information (if it doesn't store it), to move around and to attack. The idea is that the **BattleManager** is in charge of the data and the actors receive their information and operate in a managed environment. When an actor moves, the **BattleManager** should enforce moving based on its movement points, make sure it doesn't go out of bounds etc. The **BattleManager** can also ignore illegal operations (according to its policies) by actors like attacking multiple times or attacking friends. That's all there is to it. The actors relate to each other through opaque ids. These ids are refreshed every turn because actors might die and new ones may appear. Since, it's just a sample game I didn't actually implement too much policy enforcement. In online games (especially MMORPG), where the user interacts with the server using a client over a network protocol it is very important to validate any action of the client to prevent cheating, fraud and griefing. Some of these games have virtual and/or real economies and people try all the time. These can easily ruin the user experience for all the legit users.

Implementing the Object Model

The object model implementation is pretty straightforward once you get past the the dual C/C++ thing. The actual implementation resides in the C++ methods. The **ActorInfo** is just a struct with data. The **ActorInfoIterator** is just a container of **ActorInfo** objects. Let's examine the **Turn** object. It is a somewhat important object because it is a turn-based game. A fresh **Turn** object is created for each actor when it is the actor's turn to play. The **Turn** object is passed to the **IActor::play()** method of each actor. A **Turn** object has its actor information (in case the actor doesn't store it) and it has two lists of foes and friends. It provides three accessor methods **getSelfInfo()**, **getFriends()**, and **getFoes()** and two action methods: **attack()** and **move()**.

Example 7 contains the code for the accessor methods that simply return the corresponding data members and the **move()** method that update the location of the current actor.

```
ActorInfo * Turn::getSelfInfo()
{
    return self;
}
IActorInfoIterator * Turn::getFriends()
{
    return &friends;
}
IActorInfoIterator * Turn::getFoes()
{
    return &foes;
}
void Turn::move(apr_uint32_t x, apr_uint32_t y)
{
    self->location_x += x;
    self->location_y += y;
}
```

Example 7

I don't validate anything. The actor may move way outside of the arena or move more than its movement points permit. That wouldn't fly in a real game.

Example 8 contains the **attack()** code along with its helper function **doSingleFightSequence()**.

```

static void doSingleFightSequence(ActorInfo & attacker, ActorInfo &
defender)
{
    // Check if attacker hits or misses
    bool hit = (::rand() % attacker.attack - ::rand() % defender.defense) > 0;
    if (!hit) // miss
    {
        std::cout << attacker.name <<" misses " << defender.name <<std::endl;
        return;
    }
    // Deal damage
    apr_uint32_t damage = 1 + ::rand() % attacker.damage;
    defender.health -= std::min(defender.health, damage);
    std::cout << attacker.name << "(" <<attacker.health << ") hits "
        << defender.name << "(" <<defender.health <<"), damage: " << damage
<< std::endl;
}
void Turn::attack(apr_uint32_t id)
{
    ActorInfo * foe = NULL;
    foes.reset();
    while ((foe = foes.next()))
        if (foe->id == id)
            break;
// Attack only foes
    if (!foe)
        return;

    std::cout << self->name << "(" << self->health << ") attacks "
        << foe->name << "(" << foe->health << ")" << std::endl;
    while (true)
    {
        // first attacker attacks
        doSingleFightSequence(*self, *foe);
        if (foe->health == 0)
        {
            std::cout << self->name << " defeated " << foe->name << std::endl;
            return;
        }
        // then foe retaliates
        doSingleFightSequence(*foe, *self);
        if (self->health == 0)
        {

```

```
        std::cout << self->name << " was defeated by " << foe->name
<<std::endl;
        return;
    }
}
```

Example 8

The attack logic is simple. When an actor attacks another actor (identified by id), the attacked actor is located in the foes list. If it's not a foe the attack ends. The actor (via the **doSingleFightSequence()** function) hits the foe and the amount of inflicted damage is reduced from the foe's health. If the foe is still alive, it retaliates and hits the attacker and so on and so forth until one fighter dies.

That's all for today. In the next (and last) article in the series I'll cover the **BattleManager** and the game's main loop. I'll explore in-depth writing plugins for the RPG game and walk you through the directory structure of various libraries and projects that the plugin framework and the sample game are comprised of. Finally, I'll compare the plugin framework I describe here to [NuPIC's plugin framework](#). NuPIC stands for Numenta's Platform for Intelligent Computing. I developed most of the concepts and ideas I present here while creating NuPIC's plugin infrastructure.

Part 5

This is the final article in a series about developing cross-platform plugins in C++. In previous articles - [Part 1](#), [Part 2](#), [Part 3](#), and [Part 4](#) - I examined the difficulties of working with C++ plugins in portable way.

In this installment, I cover the missing pieces of the sample game introduced in Part 4 and give a demonstration. I also take a quick tour of the [source code](#) that accompanies this series, tell a few good stories (it's about time), and finally compare the plugin framework to the [NuPIC](#) (Numenta's Platform for Intelligent Computing) plugin framework, which is its conceptual ancestor. But first, let's take a look at some monster plugins that will be loaded into the game.

Monster Plugins

I created four different plugins to demonstrate the scope and diversity of the plugin framework.

- A pure C++ plugin
- A pure C plugin
- A hybrid plugin deployed as dynamic/shared libraries
- One static C++ plugin that should be linked directly to the executable.

All these plugins register their monsters with the PluginManager as actors. In addition, the game itself implements the **Hero** as an object that implements the **IActor** interface and large parts of the code that work at the **IActor** interface don't (and can't distinguish) between the game-supplied **Hero** and any monster. The game could also provide some built-in monsters.

Dynamic C++ Plugins

The dynamic C++ plugin registers the **KillerBunny** and **StationarySatan** monsters. Listing One is the KillerBunny.h header where the **KillerBunny** class is defined. **KillerBunny** is derived directly from the C++ **IActor** interface (which makes it a pure C++ plugin). It implements the **create()** and **destroy()** static functions to support creation and destruction via the PluginManager. The **StationarySatan** and any other pure C++ plugin object should look exactly the same (except for private members if any).

```
#ifndef KILLER_BUNNY_H
#define KILLER_BUNNY_H
#include <object_model/object_model.h>
struct PF_ObjectParams;
class KillerBunny : public IActor
{
public:
    // static plugin interface
    static void * create(PF_ObjectParams *);
    static apr_int32_t destroy(void *);
    ~KillerBunny();
};
```

```

// IActor methods
virtual void getInitialInfo(ActorInfo * info);
virtual void play(ITurn * turnInfo);
private:
    KillerBunny();
};
#endif

```

Listing One

Example 1 contains the implementation of **create()** and **destroy()**. They are almost trivial. The **create()** function simply instantiates a new **KillerBunny** object and returns it (as opaque **void** pointer). The **destroy()** function accepts a **void** pointer, which is actually a pointer to an instance created earlier using the **create()** function. It casts the **void** pointer to a **KillerBunny** pointer and deletes it. The important part here is that these functions let the **PluginManager** create **KillerBunny** objects without “knowing” anything about the **KillerBunny** class. The returned instance is usable via the **IActor** interface later (even though it is returned as a **void** pointer).

```

void * KillerBunny::create(PF_ObjectParams *)
{
    return new KillerBunny();
}
apr_int32_t KillerBunny::destroy(void * p)
{
    if (!p)
        return -1;
    delete (KillerBunny *)p;
    return 0;
}

```

Example 1

Example 2 contains the implementation of the **IActor** interface methods. These methods are trivial too. The **getInitialInfo()** method simply populates the **ActorInfo** struct with some data. The **play()** method is where a real **KillerBunny** will do the actual work, run around, avoid or attack enemies, and generally justify its name. Here, it just gets the list of friends from the **ITurn** interface to verify it works. This is

just laziness on my part, and in fact all the monsters don't do anything. The **Hero** is the only one actually fighting. The monsters do defend themselves when attacked and even retaliate.

```
void KillerBunny::getInitialInfo(ActorInfo * info)
{
    ::strcpy((char *)info->name, "KillerBunny");
    info->attack = 10;
    info->damage = 3;
    info->defense = 8;
    info->health = 20;
    info->movement = 2;

    // Irrelevant. Will be assigned by system later
    info->id = 0;
    info->location_x = 0;
    info->location_y = 0;
}
void KillerBunny::play(ITurn * turnInfo)
{
    IActorInfoIterator * friends = turnInfo->getFriends();
}
```

Example 2

The main point here is that writing pure C++ plugin objects is pretty easy. Other than the boilerplate **create()** and **destroy()** static methods, you just implement a standard C++ class. No arcane incantations are required.

Listing Two contains the plugin initialization code. It's not too bad, but it's boring and error prone: Define an exit function, in the **PF_initPlugin** function, define a **PF_RegisterParams** struct, populate it, and register every plugin object. Make sure you return NULL if initialization failed. Less than exhilarating. That's all it takes to write a pure C++ monster plugin (with two monsters).

```
#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;

    // Register KillerBunny
    rp.createFunc = KillerBunny::create;
    rp.destroyFunc = KillerBunny::destroy;
    res = params->registerObject((const apr_byte_t
*) "KillerBunny", &rp);
    if (res < 0)
        return NULL;

    // Register 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

Static C++ Plugins

Listing Three is the initialization code of the static plugin. It is also a C++ plugin, but it is initialized differently. All dynamic plugins (both C and C++) must implement the well-known entry point function **PF_initPlugin**. This is what the PluginManager is looking to initialize. Static plugins, on the other hand, are statically linked to the application. That means that if two plugins implement the same function name, a name clash

occurs and the application fails to link. So static plugins must have a unique initialization function and the main application must know about it to initialize it at the beginning of the program. Once the initialization function has been called, static plugins are indistinguishable from any other plugin. The last line defines a Plugin Register instance. This non-portable technique works on non-Windows platforms and allows static plugins to register themselves. This is nice and saves the application code from tight coupling with static plugins (other than linking). It makes it easy to add new static plugins just by changing the build system and without recompiling existing code. Static plugin objects look just dynamic plugin objects and implement the **IActor** interface. Example 3 contains the **play()** method of the **FidgetyPhantom** class that the static plugin registers with the PluginManager. The **FidgetyPhantom** is actually doing something in its **play()** method. It is a good example of a plugin object using objects created and managed by the application via the object model interfaces. **FidgetyPhantom** gets the first foe (usually the **Hero**), moves towards him, and attacks, if possible. It uses the **findClosest()** utility function to find the closest point (restricted by its movement points) to its foe and moves towards this point. If it reaches its enemy, it attacks.

```
void FidgetyPhantom::play( ITurn * turnInfo)
{
    // Get self
    const ActorInfo * self = turnInfo->getSelfInfo();

    // Get first foe
    IActorInfoIterator * foes = turnInfo->getFoes();
    ActorInfo * foe = foes->next();

    // Move towards and attack the first foe (usually the
    hero)
    Position p1(self->location_x, self->location_y);
    Position p2(foe->location_x, foe->location_y);

    Position closest = findClosest(p1, p2, self->movement);
```



```

turnInfo->move(closest.first, closest.second);
if (closest == p2)
    turnInfo->attack(foe->id);
}

```

Example 3

Dynamic C Plugins

C plugins register objects that implement the **C_Actor** interface. The plugin itself - and even the **C_Actor** implementation - may be C++ classes (using static methods). In this case, I implemented everything in C, just to ensure the system supports pure C plugins (there were a few compilation gotchas before it worked). The C plugin registers a single monster called **MellowMonster**. The header file of this quaint monster is presented in Example 4. This is a C object, so there is no class definition only the global functions **MellowMonster_create()** and **MellowMonster_destroy()**, which correspond to **PF_CreateFunc** and **PF_DestroyFunc**. The names are qualified with the **monster** type because, in general, a single plugin may register monster types with different pairs of **create()/destroy()** functions and in C we can't hide them in a namespace or as static methods of a class.

```

#ifndef MELLOW_MONSTER_H
#define MELLOW_MONSTER_H

#include <plugin_framework/plugin.h>

// static plugin interface
void * MellowMonster_create(PF_ObjectParams *);
apr_int32_t MellowMonster_destroy(void *);

#endif

```

Example 4

Example 5 presents the actual monster. It's just a **struct** that contains a **C_Actor** member and optionally more monster-specific data. Not much of a monster so far.

```
typedef struct MellowMonster_
{
    C_Actor actor;

    /* additional monster-specific data */
    apr_uint32_t dummy;
} MellowMonster;
```

Example 5

Example 6 is the implementation of the **C_Actor** interface and consists of two static functions (not visible outside of this compilation unit)

- **MellowMonster_getInitialInfo()** and **MellowMonster_play()** - that correspond to the **IActor** methods. The big difference is that the C++ methods get the object instance as the implicit **'this'** pointer. In C, you must pass a **C_ActorHandle** explicitly (well, not you, but the PluginManager) and the C functions laboriously cast the handle to a **MellowMonster** pointer. When the **C_Turn** object is used in the **play()** function, you must pass it its own handle too.

```
void MellowMonster_getInitialInfo(C_ActorHandle handle,
C_ActorInfo * info)
{
    MellowMonster * mm = (MellowMonster *)handle;
    strcpy((char *)info->name, "MellowMonster");
    info->attack = 10;
    info->damage = 3;
    info->defense = 8;
    info->health = 20;
    info->movement = 2;

    /* Irrelevant. Will be assigned by system later */
    info->id = 0;
    info->location_x = 0;
    info->location_y = 0;
}

void MellowMonster_play(C_ActorHandle handle, C_Turn *
turn)
{
    MellowMonster * mm = (MellowMonster *)handle;
```

```

C_ActorInfoIterator * friends =
turn->getFriends(turn->handle);
}

```

Example 6

Example 7 contains

the **MellowMonster_create()** and **MellowMonster_destroy()** functions and ties up loose ends.

The **MellowMonster_create()** function allocates a **MellowMonster** struct (using **malloc**, of course), assigns the pointer to the handle member of the actor field (without checking if the memory allocation failed, boo :-), and goes on to assign the **MellowMonster_getInitialInfo()** and **MellowMonster_play()** functions to proper function pointers. Finally it returns the **MellowMonster** pointer as an opaque **void** pointer. It is important that the **C_Actor** interface be the first member of the **MellowMonster** struct, because the PluginManager (via the adapter) casts the returned **void** pointer to a **C_Actor** pointer and treats it as such from then on.

The **MellowMonster_destroy()** frees the memory. If there is any need for destructor-like cleanup, it can do it too.

Let's check the initialization code of the C plugin in Listing Four. It looks just like the C++ plugin. This isn't surprising because it is a C function that needs to prepare a C struct and call yet another C function. The only real difference is that the registered programming language for **MellowMonster** is **PF_ProgrammingLanguage_C**. That tells the PluginManager that it's dealing with a C object and it should adapt it.

```

#ifdef WIN32
#include "stdafx.h"
#endif

#include "c_plugin.h"
#include "c_plugin.h"
#include "plugin_framework/plugin.h"
#include "MellowMonster.h"

PLUGIN_API apr_int32_t ExitFunc()
{

```

```

    return 0;
}

PLUGIN_API PF_ExitFunc PF_initPlugin(const
PF_PlatformServices * params)
{
    int res = 0;

    PF_RegisterParams rp;
    rp.version.major = 1;
    rp.version.minor = 0;

    // Register MellowMonster
    rp.createFunc = MellowMonster_create;
    rp.destroyFunc = MellowMonster_destroy;
    rp.programmingLanguage = PF_ProgrammingLanguage_C;

    res = params->registerObject((const apr_byte_t
*) "MellowMonster", &rp);
    if (res < 0)
        return NULL;

    return ExitFunc;
}

```

Listing Four

As you can see, working at the C API level is much thornier. You need pass explicit handles around, cast a lot, pay attention to function names, and hook up free functions to your monster **C_Actor** interfaces. It's not fun, but it's survivable if you must work with C.

Hybrid C/C++ Plugins

So C is cumbersome and you would like to use the C++ API, but you must provide total binary compatibility. What a conundrum. Fortunately, the Hybrid C/C++ plugin is just the ticket. It provides C compatibility with a C++ API via **ActorBaseTemplate**. It also cuts down significantly on the plugin initialization code by using **PluginHelper**. The plugin registers two monsters: **GnarlyGolem** and **PsychicPiranea**. Both

derive from **ActorBaseTemplate** and both implement the C++ **IActor** interface, but **GnarlyGolem** is actually talking in C to the PluginManager while **PsychicPiranea** is talking C++.

Example 8 contains the definitions of both classes. You can see how compact and clean they look. There is no need for the **create()** or **destroy()** static methods anymore (**ActorBaseTemplate** takes care of it). The only difference between the two is that **PsychicPiranea** specifies the **IActor** as the second template parameter for **ActorBaseTemplate**. This is the infamous **Interface** parameter, which is **C_Actor** by default.

```
class GnarlyGolem : public ActorBaseTemplate<GnarlyGolem>
{
public:
    GnarlyGolem(PF_ObjectParams *);
    // IActor methods
    virtual void getInitialInfo(ActorInfo * info);
    virtual void play(ITurn * turnInfo);
};

class PsychicPiranea : public
ActorBaseTemplate<PsychicPiranea, IActor>
{
public:
    PsychicPiranea(PF_ObjectParams *);
    // IActor methods
    virtual void getInitialInfo(ActorInfo * info);
    virtual void play(ITurn * turnInfo);
};
```

Example 8

PsychicPiranea could have been derived from **IActor** directly, just like **KillerBunny** and **StationarySatan** in the C++ plugin. The reason it derives from **ActorBaseTemplate** is threefold:

- It saves you from write **create()/destroy()** static methods
- It lets you switch quickly between C and C++ if you deploy the same plugins in different situations
- So I can demonstrate this cool capability

This is really cool because between the automatic adaptation of C objects by the PluginManager and

the nice C++ wrapper that **ActorBaseTemplate** provides, application developers and plugin developers can be blissfully ignorant of the C that flows between them. The only developers who should be concerned with the C/C++ dualism are the object model developers. If your system is a serious platform, then the object model will congeal sooner or later. Then everyone can forget about C and just extend the application that's built on top of the object model and write lots of plugins - all in C++.

The implementation of **GnarlyGolem** and **PsychicPiranea** is of typical C++ plugin implementations. **GnarlyGolem** is C under the covers, but it doesn't care.

Listing Five is the initialization code of the hybrid plugin. There are more **#include** statements than code. (I kid you not. Count them. This is as close to a domain-specific language as it gets - without macros, anyway.) You define a **PluginHelper** object and call **registerObject()** for each object. No need for annoying structs with lots of function pointers, no need for error checking. Pure simplicity. In the end, return the result.

```
#include "wrapper_plugin.h"
#include "plugin_framework/plugin.h"
#include "plugin_framework/PluginHelper.h"
#include "GnarlyGolem.h"
#include "PsychicPiranea.h"

extern "C" PLUGIN_API
PF_ExitFunc PF_initPlugin(const PF_PlatformServices *
params)
{
    PluginHelper p(params);
    p.registerObject<GnarlyGolem>((const apr_byte_t
*) "GnarlyGolem");
    p.registerObject<PsychicPiranea>((const apr_byte_t
*) "PsychicPiranea", PF_ProgrammingLanguage_CPP);

    return p.getResult();
}
```

Listing Five

There is one nit to pick. When registering the `PsychicPiranea`, you need to specify `PF_ProgrammingLanguage_CPP` (the default is `PF_ProgrammingLanguage_C`). The programming language can be gleaned automatically from the `PsychicPiranea` class itself because it passed the programming language as the `Interface` parameter to the `ActorBaseTemplate`. However, this requires some template meta-programming tricks (type detection) and I'm running out of scope quickly. Here is a quick link if you're curious: www.ddj.com/cpp/184402050.

Let's Play!

It's play time. I'll take you now to a quick tour of the game itself. You will see how the `PluginManager` is initialized, how the monsters are created and how battles are conducted. Let's start with `main()`.

Listing Six contains the `main()` function. You can skip all the `#include` statements and concentrate on the `DummyInvokeService()` function. This function serves as the `invokeService` in the `PF_PlatformServices` struct that the plugins receive. It doesn't do anything in this case, but in real applications it has a major role in providing system services to plugins.

```
#ifdef WIN32
#include "stdafx.h"
#endif

#include "plugin_framework/PluginManager.h"
#include "plugin_framework/Path.h"
#include "BattleManager.h"
#include "static_plugin/static_plugin.h"
#include <string>
#include <iostream>

using std::cout;
using std::endl;

apr_int32_t DummyInvokeService(const apr_byte_t *
serviceName, void * serviceParams)
```

```

{
    return 0;
}

#ifdef WIN32
int _tmain(int argc, _TCHAR* argv[])
#else
int main (int argc, char * argv[])
#endif
{
    cout << "Welcome to the great game!" << endl;
    if (argc != 2)
    {
        cout << "Usage: great_game <plugins dir>" << endl;
        return -1;
    }
    // Initialization
    ::apr_initialize();
    PluginManager & pm = PluginManager::getInstance();
    pm.getPlatformServices().invokeService =
    DummyInvokeService;
    pm.loadAll(Path::makeAbsolute(argv[1]));

    PluginManager::initializePlugin(StaticPlugin_InitPlugin);

    // Activate the battle manager
    BattleManager::getInstance().go();

    ::apr_terminate();

    return 0;
}

```

Listing Five

The **main()** function is defined to conform to both Windows and UNIX systems. The game is very portable. I tested it on Windows XP SP2, Vista, Mac OS X 10.4 (Tiger), Mac OS X 10.5 (Leopard), and Kubuntu 7.10 (Gutsy Gibbon). These are the most common modern OS out there. It will probably work as-is or with tweaks to the build procedure on a range of other OSs.

Inside **main()** there is a check that the user passed the plugin directory as a command-line argument. The APR library is initialized and the

PluginManager makes its entrance. It is a singleton and is destructed when the application terminates. The next step is to assign **DummyInvokeService** to the platform services struct. Once **invokeService** is ready, the plugins can be initialized. First, all of the dynamic plugins are loaded from the directory and passed as `**argv[1]**`, then the static plugin is initialized explicitly. This is unpleasant, but I couldn't find a portable solution that works on Windows. Once all the plugins are initialized, BattleManager takes command. Finally, the APR library is cleaned up.

Pretty straightforward: Check command-line arguments, initialize global resources, load plugins, transfer control to the application logic, and clean up global resources.

BattleManager is the brain of the game. Example 9 contains the entire **go()** method. It starts by extracting all the registered monster types from the PluginManager.

```
void BattleManager::go()
{
    // Get all monster types
    PluginManager & pm = PluginManager::getInstance();
    const PluginManager::RegistrationMap & rm =
pm.getRegistrationMap();

    for (PluginManager::RegistrationMap::const_iterator i =
rm.begin(); i != rm.end(); ++i)
    {
        monsterTypes_.push_back(i->first);
    }

    // Dump all the monsters
    for (MonsterTypeVec::iterator i = monsterTypes_.begin();
i != monsterTypes_.end(); ++i)
    {
        std::string m = *i;
        std::cout << m.c_str() << std::endl;
    }

    // Add the Hero to its faction (later allies may join)
    ActorInfo ai, heroInfo;
    hero_.getInitialInfo(&heroInfo);
}
```

```

    // Don't keep the hero's IActor *, because she is
    treated differently
    actors_.insert(std::make_pair((IActor *)0, heroInfo));
    heroFaction_.push_back(&actors_[0]);

    // Instantiate some monsters randomly
    for (apr_int32_t i = 0; i < MONSTER_COUNT; ++i)
    {
        IActor * monster = createRandomMonster(rm);
        monster->getInitialInfo(&ai);
        ai.id = i+1; // Hero is id 0
        actors_.insert(std::make_pair(monster, ai));

        enemyFaction_.push_back(&actors_[monster]);
    }

    while (!gameOver_)
    {
        playTurn();
    }

    heroInfo = actors_[0];
    if (heroInfo.health > 0)
        std::cout << "Hero is victorious!!!" << std::endl;
    else
        std::cout << "Hero is dead :-( " << std::endl;
}

```

Example 9

This is a dynamic step. BattleManager doesn't have a clue what monsters are there - and doesn't care. It doesn't even know about the **FidgetyPhantom** from the static plugin. It dumps all the monster types to the console for good measure (and for me to make sure all the monsters were registered properly). Then it puts the **Hero** (which is known and gets special treatment) in the actors list. BattleManager needs to know about **Hero** because the fate of **Hero** is linked with the fate of the entire game and the all important "game over" condition. Then the monsters are created randomly using the **createRandomMonster()** function. Finally, we get to the main loop: "while the game is not over play a turn". When the game is over, it's time to display a dazzling text message to the console,

which states if the hero won or died. As you can see, my art budget for this game was pure fiction. Example 10 contains

the **createRandomMonster()** method. It selects a random monster by index, based on the total number of registered monster types, then creates it by calling the **ActorFactory::createActor()** method, passing it the monster type.

```
IActor * BattleManager::createRandomMonster(const
PluginManager::RegistrationMap & rm)
{
    // Select monster type
    apr_size_t index = ::rand() % monsterTypes_.size();
    const std::string & key = monsterTypes_[index];
    const PF_RegisterParams & rp = rm.find(key)->second;
    // Create it
    IActor * monster = ActorFactory::createActor(key);

    return monster;
}
```

Example 10

The **ActorFactory** is the application-specific object adapter that derives from the generic **ObjectAdapter** provided by the plugin framework. From the point of view of the **BattleManager**, all created monsters are just objects that implement the **IActor** interface. The fact that some of them are adapted C objects or that some hail from remote plugins is immaterial. This is a turn-based game, so what happens when a turn is played in the main loop? Example 11 contains the **playTurn()** method, which provides the answer.

```
void BattleManager::playTurn()
{
    // Iterate over all actors (start with Hero)
    //For each actor prepare turn info (friends and foes)
    Turn t;

    ActorInfo & ai = actors_[(IActor *)0];
    t.self = &ai;
    std::copy(heroFaction_.begin(), heroFaction_.end(),
std::back_inserter(t.friends.vec));
}
```

```

    std::copy(enemyFaction_.begin(), enemyFaction_.end(),
std::back_inserter(t.foes.vec));
    hero_.play(&t);

    ActorInfo * p = NULL;
    ActorMap::iterator it;
    for (it = actors_.begin(); it != actors_.end(); ++it)
    {
        if (!it->first || isDead(&it->second))
            continue;

        t.self = &(it->second);
        std::copy(heroFaction_.begin(), heroFaction_.end(),
std::back_inserter(t.foes.vec));
        std::copy(enemyFaction_.begin(), enemyFaction_.end(),
std::back_inserter(t.friends.vec));

        it->first->play(&t);
    }

    // Clean up dead enemies
    Faction::iterator last =
std::remove_if(enemyFaction_.begin(),
enemyFaction_.end(), isDead);
    while (last != enemyFaction_.end())
    {
        enemyFaction_.erase(last++);
    }

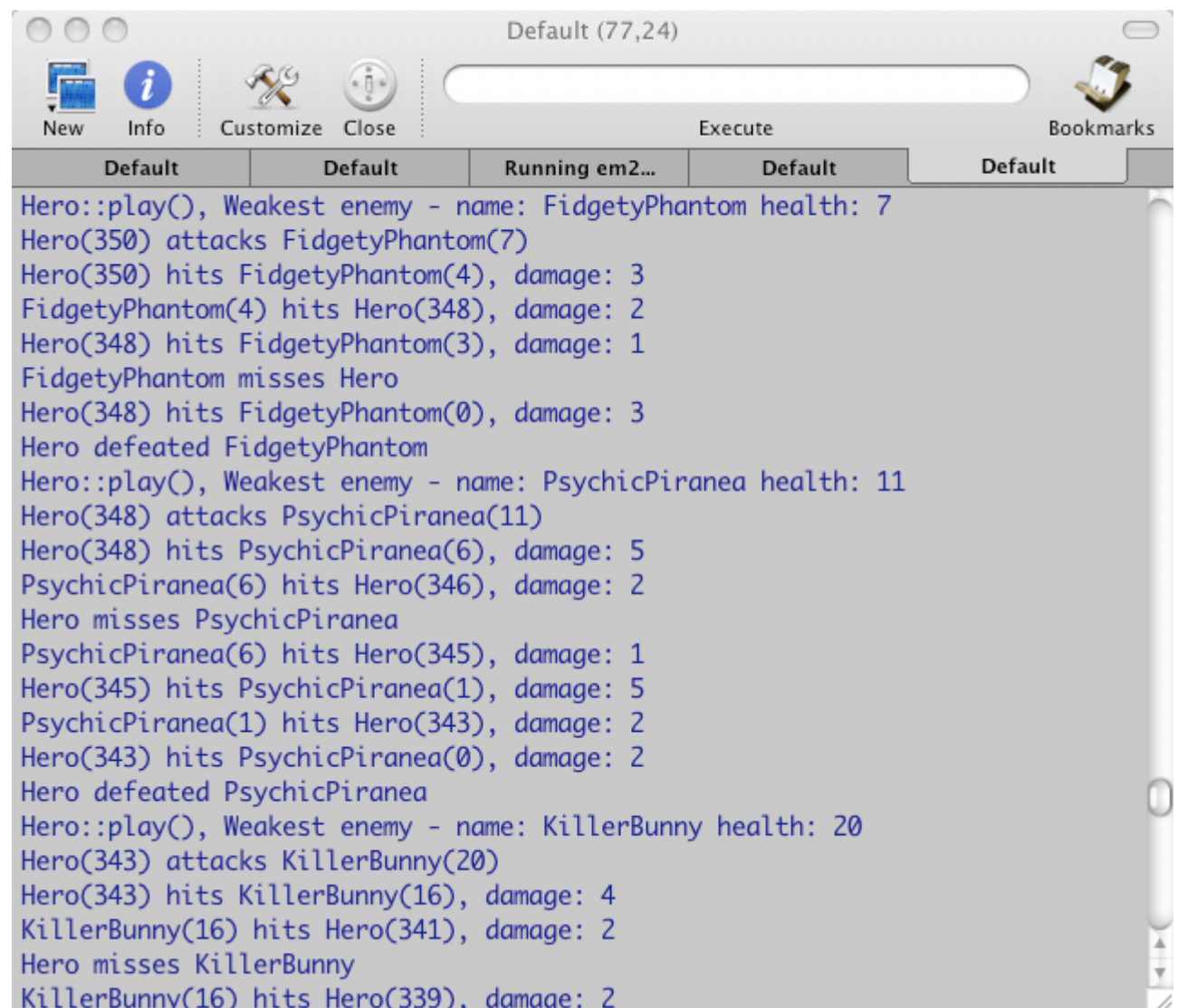
    // Check if game is over (hero dead or all enemies are
dead)
    if (isDead(&actors_[(IActor *)0]) ||
enemyFaction_.empty())
    {
        gameOver_ = true;
        return;
    }
}

```

Example 11

The BattleManager starts by creating a **Turn** object on the stack. Each [live] actor gets his **Turn** object with proper information and acts on it. The **Hero** goes first. The BattleManager invokes its **play()** method, passing the turn object.

The **Hero** moves about and attacks. The BattleManager is aware of all the action that transpires because the data structures that are manipulated are the **ActorInfo** structs the BattleManager manages. Once the hero is done, each of the other actors gets its chance to do some mayhem. After all the actors do their worst, it's time to remove the bodies from the battle field. This is done with the help of the standard **std::remove_if** algorithm and the **isDead()** predicate that checks if the actor has zero health points. Before the turn ends, the BattleManager checks if the **Hero** or all the monsters are dead. The game goes on until one of these conditions is fulfilled. Figure 1 shows the game in progress.



The screenshot shows a software interface with a menu bar (New, Info, Customize, Close, Execute, Bookmarks) and a toolbar. Below the toolbar is a tabbed interface with tabs labeled 'Default', 'Default', 'Running em2...', 'Default', and 'Default'. The main area displays a log of game events in blue text:

```
Hero::play(), Weakest enemy - name: FidgetyPhantom health: 7
Hero(350) attacks FidgetyPhantom(7)
Hero(350) hits FidgetyPhantom(4), damage: 3
FidgetyPhantom(4) hits Hero(348), damage: 2
Hero(348) hits FidgetyPhantom(3), damage: 1
FidgetyPhantom misses Hero
Hero(348) hits FidgetyPhantom(0), damage: 3
Hero defeated FidgetyPhantom
Hero::play(), Weakest enemy - name: PsychicPiranea health: 11
Hero(348) attacks PsychicPiranea(11)
Hero(348) hits PsychicPiranea(6), damage: 5
PsychicPiranea(6) hits Hero(346), damage: 2
Hero misses PsychicPiranea
PsychicPiranea(6) hits Hero(345), damage: 1
Hero(345) hits PsychicPiranea(1), damage: 5
PsychicPiranea(1) hits Hero(343), damage: 2
Hero(343) hits PsychicPiranea(0), damage: 2
Hero defeated PsychicPiranea
Hero::play(), Weakest enemy - name: KillerBunny health: 20
Hero(343) attacks KillerBunny(20)
Hero(343) hits KillerBunny(16), damage: 4
KillerBunny(16) hits Hero(341), damage: 2
Hero misses KillerBunny
KillerBunny(16) hits Hero(339), damage: 2
```

That's it. Go ahead and give it a try, but don't go addict on me :-)

Source Code Walkthrough

Okay, there is a lot of source code. This subject is complicated and I didn't want to give just a couple of snippets of sample code and half-baked libraries. I put a lot of effort in organizing the code in a reusable form. I divided the code to multiple libraries and directories according to their function and their dependencies on other libraries. There is one core library, two game-specific libraries, the game itself (the executable), and four different plugins (three dynamic libraries and one static library). In addition, there are two third-party libraries - APR and boost. Finally there are the build system(s) for the various platforms.

Directory Structure

There are three top-level directories: `include`, `lib`, and `projects`. `include` and `lib` contain the external libraries and `projects` include the code I wrote. Inside `include` and `lib` are subdirectories for `darwin86`, `linux32`, and `win32`, that contain the platform-specific header files (in `include`) and static libraries. Actually, the only real library is APR because I use only header files from Boost that don't require building and they are identical on all platforms. So Boost resides directly under `include` and not in platform specific subdirectory.

Source Code

All the source code is in subdirectories under "projects":

- **plugin_framework**. This is the core library that contains the portability layer (`DynamicLibrary`, `Directory` and `Path`), the plugin API definitions (`plugin.h`) and all the support and helper objects. You will need this library for your plugin-based systems. You may wish to put the OS portability classes in a separate library or

replace them with your OS abstraction layer. I put them in the same library to make it self contained (see caveat below in the external libraries section).

- **utils**. This is a silly little library that contains two functions: **calcDistance** and **findClosest**. The reason it is a library is because it is used by the **Hero** which is part of the main executable and by some of the plugin objects, so it needs to be linked to multiple binaries. You can safely ignore it.
- **object_model**. This is where all the game-specific dual C/C++ objects reside, along with **ActorFactory**, the object model wrappers and the **ActorBaseTemplate** classes used by the hybrid plugins. If you want to use this plugin framework for your own projects, then this library gives you a blueprint of all the stuff you need to implement. It is important to note that you can pick and choose. If you don't care about C compatibility because you build the main system and all the plugins, then you can just get rid of all the C stuff. If on the other hand you care about C but you don't want to provide nice hybrid C/C++ API to plugin developers, you can just ignore the **ActorBaseTemplate** and the object model wrappers. Read the article again (especially Parts 3 and 4) with the code nearby and you will know what to do.

The plugin libraries are straightforward. `c_plugin` is the C plugin (**MellowMonster**), `cpp_plugin` is the direct C++ plugin (**KillerBunny** and **StationarySatan**), `static_plugin` is the statically linked plugin (**FidgetyPhantom**) and `wrapper_plugin` is the hybrid C/C++ plugin (**GnarlyGolem** and **PsychicPiranea**). In most situations, **wrapper_plugin** is the right choice. It is easy to write plugins if you invest in writing the base template and object model wrappers. The plugin related overhead is practically zero.

- **great_game**. This is the game itself. It contains the **Hero** and **BattleManager** and of course the **main()** function. It gives you a good feel for how to bootstrap the whole process and what is involved in integrating the **PluginManager** with an application.

This article series covers the code well and explains the reasons for design decisions and trade offs. I tried to maintain a clean and consistent style. The `plugin_framework` and the `object_model` libraries, which are the heart of the system, are pretty close to industrial-strength level (maybe with a little more error handling and documentation). The rest of the code demonstrates the entire system in action. The only part that I didn't demonstrate convincingly is the **`invokeService()`** call. All the mechanisms are in place though. I invoke my author privileges and assign it as the dread exercise to the reader to implement a couple of services in the **`DummyInvokeService()`** function in `great_game.cpp`

External Libraries

The game and the framework use two external libraries: APR and boost. The Apache Portable Runtime (APR) is a C library. I use its portable basic types (all the `apr_xxx_t` thingies) and some (very little) of its OS abstraction capabilities. I implemented most of the portable OS layer (`DynamicLibrary`, `Directory`, and `Path`) directly as alternative implementations on top native Windows and POSIX APIs. Boost is used mostly for its **`shared_ptr`** and **`scoped_ptr`** templates. This is very small ROI (return on investment) for burdening oneself with external dependencies, especially in the context of a cross-platform portable system. This is true. I considered removing all the dependencies and implement what little I needed from APR directly, have my own portable basic types header, and grab a smart pointer template instead of dragging Boost. I decided to leave them in to show that this is indeed a serious system that can cater to different needs and integrate with external libraries. I paid a price for it as you will hear soon enough.

Building the Sample Game and Its Plugins

I put a lot of effort into this part because going cross-platform can increase the complexity of a system by an order of magnitude. If you don't use proper tools and processes you can spend a lot of time chasing platform-specific issues.

- **Windows.** I provide a Visual Studio 2005 solution file in the root directory. This is preferred development environment for 99.99% of Windows developers. Visual Studio Express is free for all, so there is no cost. Some people think that cross-platform development means a single build system. I'm not one of them. I had my share of bad experiences with cygwin and MinGW. It's nice when it works and it's terrible when it doesn't. It is definitely a chore to keep multiple build systems in sync. Maybe in a future article I'll explore this issue further.
- **Mac OS X.** On the Mac (and Kubuntu) I used NetBeans 6.0. I can't tell you how impressed I am. It is an amazing C++ development environment. It comes this close ("putting two fingers very close to each other") to Visual Studio. It totally blows Eclipse CDT (C/C++ development tools). The only annoying part (that Visual Studio gets right) is that when you set project properties like preprocessor symbols, compiler flags, etc., you must do it separately for each project even if you have 20 or 200 projects that need the same settings. The other part where NetBeans is a little weak is the debugger. It uses GDB, which is not a stellar C++ debugger (did you ever managed to probe into an `std::map` in gdb?). The best part about NetBeans 6.0 (and earlier versions too) is that it uses Makefiles as its underlying build system. That means that you can deploy your source code to anyone (as I'm doing here) and they'll be able to do: `./configure; make` and run your application. I still encourage you to get NetBeans 6.0 if you don't develop exclusively on Windows.

- **Linux (Kubuntu 7.10).** Punch line: Eventually, I got it working. The story starts with me assuming I can just take my nice NetBeans project from the Mac and it works with a minor tweak or two. Think again. Kubuntu 7.10 comes with NetBeans 5.5.1 installed. I was happy and just tried to browse to my project. Nope. It turns out that only the Java environment is installed. Okay, I tried to update, but the C++ pack wasn't available. Okay, moving on. I tried to download the sources from NetBeans.org and build it, but I was informed that JDK1.6 update 3 is required. I tried to apt-get it and Kubuntu asks me for the installation CD. Now, I'm running in a VM that I installed from an .iso image. I deleted the 4+ GBytes .iso image because I didn't think I'll need it (why can't I update from the web?). I didn't want to download 4+ gigs, so I gave up on NetBeans and I opted to use KDevelop. It is the native KDE development environment, C++ is its strong suite and I knew that it uses automake/autoconf as its underlying build system. I had about 10 projects, but I wanted to check it out and see how easy it is to crank top-notch automake-based solution with multiple sub-projects. Well, I failed. I don't remember all the details, but I failed. I remember that it took a few minutes to locate the Automake manager (a vertical tab on the right-side of the IDE, when all the important tabs are on the left, you can move it to left when you find it). I decided to fall back to command line. It is Linux after all. People can get by without a flashy GUI IDE. I took my NetBeans-generated Makefile-based projects and started configuring and building. I had to "sed s/GNU-MacOSX/GNU-Linux-x86/g -i" a couple of times and a few more serious tweaks like replacing the linker flags (it turns out on the Mac its totally different) and I got pretty far. All the projects compiled and all the plugins linked successfully. Only, the great_game itself wouldn't link due to APR failures (couldn't find pthreads mostly). It didn't make sense pthreads

is installed of course. I tried to apt-get just to see if there are some dev libraries missing. Nada. My pre-built APR library (actually two libraries) was built on a different linux distro. I download the APR source and tried to build from source. It built just fine, but I still got the same link errors.

I got really upset and grudgingly bit the bullet. I re-downloaded the Kubuntu-7.10 huge .iso image and mounted it. Things went downhill from here. I installed the JDK 1.6 update 3 through apt-get, and got and installed NetBeans 6.0 C++ pack from NetBeans.org. It created an icon on the desktop and I launched NetBeans. It loaded the Mac project files just fine and I had to make a couple of minor modifications and one more serious one. It turns out I had to add pthreads explicitly to the project, where on the Mac it just worked. So much for a unified build system. Finally all was well, after a few mistrials (I built the plugins into the wrong directory), I got everything working.

The Mac and Linux builds share the same Makefile in each project directory. This shared Makefile includes other files from a sub-directory called nbproject. Some of these files are different for the Mac and Linux. I wanted to ship build systems for both so I created two subdirectories under each project called nbproject.mac and nbproject.linux . This means that if you try to open these projects on the Mac or Linux with NetBeans, you will fail. You need to rename the appropriate sub-directory (based on your platform) to nbproject. This is annoying when you have eight different projects, so I created a little shell script called build.sh in the root directory. You should run it with either 'mac' or 'linux' and it copies the nbproject.mac or nbproject.linux to nbproject in each project and even runs make. You need to do it just once and then you will be able to open the projects in NetBeans.

Building External Libraries

The good news are that you don't have to. I provide pre-built static libraries for APR for all the platforms and the subset of Boost I use is just header files so no need to build it.

There is a little story here too. Cross-platform development is a difficult undertaking, so companies usually start with one platform and when they grow up and mature and the codebase stabilizes, they port to other platforms. Numenta (the company I work for) had a different idea. They decided to go cross-platform from the get-go, but only do Mac and Linux. This is probably the only company EVER to do that. When we launched our research platform, the most common complaint was the absence of a Windows version. I just completed the second phase of the NuPIC plugin framework. This is the ancestor of the plugin framework and all the idioms I present here. I quickly volunteered to take on the Windows port. I was the only developer with significant Windows expertise (all the other guys are UNIX gurus), but mostly I wanted to be able to debug in Visual Studio and forget about gdb. Our codebase was pretty much high-level standard C++. There were quiet a few direct system calls and many UNIX path issues, but nothing too terrible. All was going well, until I got to Boost. At this time we used a few Boost libraries that require building: `boost::file_system`, `boost::regex` and `boost::serialization`. I labored for days, scouring the web and trying strange combination of symbols, but I couldn't build these libraries under VC++ 2005. Eventually I had to write an OS abstraction layer on top of Win32 and POSIX APIs to replace these Boost libraries. The Directory and Path classes of the plugin framework are some of the fruits of this effort.

The NuPIC Plugin Framework

Well, before I let you go home I want to share the history of the plugin framework. It grew out of the NuPIC plugin framework. NuPIC is a platform for intelligent computing and its got a pretty interesting and very clean architecture. I don't mind praising it because it was designed before I came on board, so it's not auto-complimentation (don't confuse with auto-completion).

NuPIC is a platform for distributed computing conceived to run massive HTM (Hierarchical Temporal Memory) networks. The term "node" is usually reserved to a physical machine or a host in cluster nomenclature. In NuPIC the term "node" refers to a node in the HTM network. These nodes can be sensors, compute nodes or effectors. From a software engineering point of view, there are many node processor processes running on many cores (same or different machines), each one of them running multiple HTM nodes, there is a supervisor process that orchestrates all the action and there are tools that communicate with the supervisor via a sophisticated API. The goal is usually to train a HTM network by feeding it some data and then perform inference by showing it novel data and see how similar it is to something the network learned. The runtime engine is the supervisor and all the node processor. The application or tools communicate with the supervisor to load networks files, run computations, and control the live networks at runtime by sending commands and setting parameters. Where do plugins fit into the picture? Actually everywhere. Every node in the HTM network is implemented as a plugin. When a network file is loaded into the runtime engine, it is parsed and the specific nodes are instantiated based on their type just like the monsters in the game. Figure 2 is a high-level illustration of NuPIC's architecture.

NuPIC Software Architecture

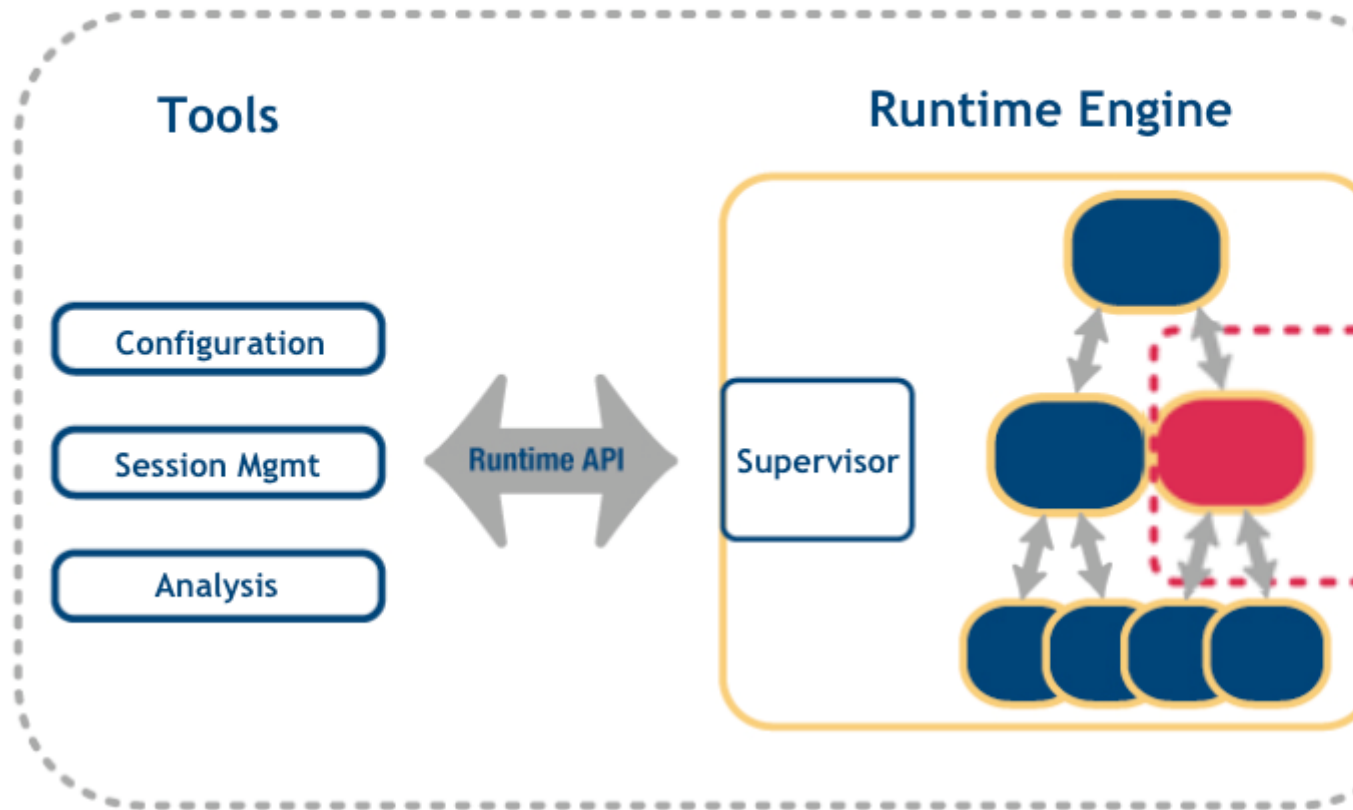


Figure 2: NuPIC architecture

I think that looking at another system with a radically different interface helps clarify the boundary between generic plugin framework concerns and the application and its object model.

Listing Seven contains the definition of the **INode** interface. It is the moral equivalent of the **IActor** interface in the game. It is how the runtime engine knows its nodes are same as the game know its actors.

```
struct INode
{
    virtual void init(INodeInfo & nodeInfo) = 0;
    virtual void compute() = 0;
    virtual void execute(IReadBufferIterator & args,
        IWriteBuffer & out) = 0;
    virtual void getParameter(const Byte * name, IWriteBuffer
        & value) = 0;
    virtual void setParameter(const Byte * name, IReadBuffer
        & value) = 0;
    virtual void saveState(IWriteBuffer & state) = 0;
}
```

```
};
```

Listing Seven

The **compute()** method is the main one. It is called repeatedly and it is pretty much like the **IActor::play()** method. The **init()** method initializes the node and allows persistent nodes with the corresponding **saveState()** method. The runtime engine can tell the node to save its state. Later it can recreate the same node (e.g., on a different host) based on the state.

The **get/setParameter()** and **execute()** methods are something new. They allow plugin nodes to arbitrarily extend their interface. Any node can have any number of parameters and execute commands. This effectively sidesteps the limitations of a well-defined interface. The runtime engine can't use these node-specific parameters and commands because it works at the **INode** interface, but users and applications construct networks that contain specific node types and can interact with them at runtime by getting/setting specific parameters and sending commands. There is nothing new here. The data types used for the generic methods are **IReadBuffer** and **IWriteBuffer**, which are glorified opaque buffers. They have some formatting capabilities similar to IO streams via overloaded **read()** and **write()** methods, but it is up to the user to know what is the format of each parameter, what arguments each command expects and what are the valid values to pass. This is very fragile and akin to having all your functions accept a void pointer opaque buffer that the caller must pack and the function must unpack or parse. A partial solution to this problem is the node spec. The node spec is a struct that defines for each parameter its name, type, element count (1 is a scalar, >1 is an array, 0 is variable size), description, constraints, default value, and access (create, get, set, and combinations). This is a lot of metadata and lets the runtime engine and generic tools interact with nodes at a very high level. For example, tools can automatically display the description of every parameter as a tooltip and validation can be performed based on numerical

ranges (0.5 ... 1.2), regular expressions (ab.*) or enumerations (red, green, blue). The nice thing about it is that all this metadata is not available only on the runtime engine, but also on the client side (tools). The tools libraries load the plugins too via the PluginManager and they have all this information at their disposal. So, if a user misspells the name of a command the tools can let her know right there and display a list of valid commands. If she then tries to set a parameter that has only a "get" access it will get descriptive error message that includes the access right for this parameter.

Wrap Up

I'm done. Thank you for making it all the way. I'm sure it wasn't easy. I hope that you will use this framework and the patterns and idioms it supports in your own applications and systems.