Building Your Own Plugin Framework: Part 4

Hybrid C/C++ plugins

January 14, 2008
URL:http://www.drdobbs.com/cpp/building-your-own-plugin-framework-part/205604762

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

Editor's Note: This is the fourth installment of a five-part series by Gigi Sayfan on creating cross-platform plugins in C++. Other installments are: Part 1, Part 2, Part 3, and Part 5.

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 cycel 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 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 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 correponding 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 are 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 appliucation 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)
  {
```

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 maintainance headache. Each wrapper derives from the C++ interface and accepts the correponding 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 objct model, but you don't have too. 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 pluguin 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

```
    turn_->attack(turn_->handle, id);
  }
private:
  C_Turn * turn_;
  ActorInfoIteratorWrapper friends_;
  ActorInfoIteratorWrapper foes_;
};
#endif // OBJECT_MODEL_WRAPPERS_H
```

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 is 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 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 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 inerface
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 headaches to aspiring plugin developers. The **create()** function simply creates a new instance of **T** (the derived class) and initalizes 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 (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 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)
  {
```

Example 6

The implementation of both interface functions is almost identical; **getSelf**(), calls the C++ **IActor** implementation in the derived class via the wonders of polymorphism and employs 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.

```
  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");
  }
}
```

The error handling is another nice feature of **ActorBaseTemplate**. It lets plugin developers 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](https://www.drdobbs.com/).

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 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 chooses to throw a standard **std::runtime_error**, then the error-handling code extracts 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 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_PlatfromServices** 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 shoul not let exceptions propagate across the binary compatibility bounday. 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, concatanate 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 concpeptual foundations are sound though, and it can definitely be extended. Now that I have reduced your expections 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 thast 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 palce 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 his turn to play. The **ITurn** interface lets the actor 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 clinet 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 easilly ruin the user experience for all the legit users.

Implementing the Object Model

The object model implementation is 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 returns the corresponding data members and the **move()** method that updates 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-&tl;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.