TECH Kam Low

Building a Simple C++ Cross-platform Plugin System

This article talks about some key factors to consider when designing a plugin system in C++, and also provides some practical examples of how we've addressed these issues in our own code. There's a lot to cover here, such as binary compatibility, strict API versioning, and interprocess memory management. Sounds like fun right? Well, if you get it wrong then it sure as hell won't be, sometime in the near future you'll most likely have a suicide inducing customer support nightmare on your hands, but if you get it right then it's really not so bad.

If you're looking for the code, we've released it as an open source LibSourcey module called Pluga on Github.

ABI and Binary Compatibility

The first thing to consider when designing your Plugin API is ABI compatibility. Unlike dynamically scripted languages, C++ demands that any shared libraries loaded by the runtime are binary compatible, otherwise all hell breaks loose. Essentially, this means that both the application and plugins must be compiled using the exactly the same development environment.

If you can, try and stick to this one simple rule: only pass POD (plain old data) data types across process boundaries. By sticking to POD types the binaries will have no interdependent shared libraries, and you can avoid binary compatibility issues altogether.

The tradeoff is that standard libraries differ from compiler to compiler, platform to platform, even version to version, so they should always be deemed to be binary incompatible. This means that unless you want to force clients to use exactly the same OS, compiler and third party dependencies as you when buildng plugins for your application, then you'll need to avoid using STL containers or other complex types in your Plugin API. "But this is C++!?", you cry in anguish, and unless you're rockin' a mullet with a stubbie cooler then who could blame you? Unfortunately that's just the way it is, so in this case all we can do is bite our collective upper lip and move on.

There are alternatives, such as embedding the standard libraries in your project using STLport or similar to ensure consistency between platforms, but why bother? It seems like overkill, wouldn't it be easier to just to pass a `void*` or a `char*` buffer and encode/decode it as required across the process boundary?

The method we've been using recently is actually quite simple. The plugin implements a `onCommand` method which accepts arbitrary commands from the application. The advantage of using this type of interface is that you're able to implement almost any kind of functionality without having to add new methods and break the API each time you roll out a new feature. Obviously it doesn't have it be this simple, but you get the idea!

Command nodes (see code below) are namespaced using a REST style interface like so `resource:action`, and the data buffer contains either a JSON encoded message which for representing and converting to an STL container such a `std::vector` or `std::map` to pass to the internal API, or it may just be a raw data buffer that can be used directly.

Take, for example, the following code:

```
bool onCommand(const char* node, const char* data, unsigned int size)
{
    try {
        // Handle a JSON encoded options hash
        if (strcmp(node, "options:set") == 0) {
            json::Value root;
            json::Reader reader;
            if (!reader.parse(data, size, root))
                throw std::runtime_error("Invalid JSON format: " +
reader.getFormatedErrorMessages());

            // Do something with JSON data here...
        }

        // Handle raw file data
        else if (strcmp(node, "file:write") == 0) {
            std::string path("test.bin");
            std::ofstream ofs(path, std::ios::out|std::ios::binary);
            if (!ofs.is_open())
                throw std::runtime_error("Cannot write to output file: "
+ path);
            ofs.write(data, size);
            ofs.close();
        }

        // Handle unknown commands
        else throw std::runtime_error("Unknown command");
    }
    catch (std::exception& exc) {
        // Catch exceptions here and return false.
        // You could set a lastError string here which is exposed to
        // the application that returns the error message as a char*.
        // See the full example for details.
        std::cerr << "Command error: " << exc.what() << std::endl;
        return false;
    }
    return true;
}
```

Interprocess Memory Handling

One other simple rule that will save you no small amount of frustration later on down the track is: any memory allocated by a process should be deallocated by the same process.

Let's say the application asks the plugin to allocate and return a `char*` buffer, and then proceeds to delete the buffer when it's done with it. Honk! Big no no, you're just asking for a crash.

This is good:

```
bool askPluginForSomeSugar(Plugin* plugin)
{
    // allocate buffer of some sort
    char* data = plugin->gimmeSomeSugarBaby();

    // do something cool with data

    // hand the pointer back to the plugin to be deallocated
    plugin->putSugarBackInTheBowl(data);
}
```

This is bad:

```
bool askPluginForSomeSugar(Plugin* plugin)
{
    // allocate buffer of some sort
    char* data = plugin->gimmeSomeSugarBaby();

    // do something cool with data

    // don't manage memory data allocated by the other process!
    delete[] data;
}
```

Plugin System API

The Pluga plugin system API consists of a single header file that defines a set of macros which export a `PluginDetails` structure. The `PluginDetails` structure exposes basic plugin information, a compile time API version, and a static initialiser function to the main application on runtime. By having an intermediary `PluginDetails` structure that's loaded on runtime before the plugin is instantiated, we can do things like sanity check the API version, and print information about the plugin.

Note that the system API also forward declares the `IPlugin` type, which must be defined externally in your own code. See the Plugin API for more information about that.

```
//
// LibSourcey
// Copyright (C) 2005, Sourcey <http://sourcey.com>
//
// LibSourcey is free software; you can redistribute it and/or
// modify it under the terms of the GNU Lesser General Public
// License as published by the Free Software Foundation; either
// version 2.1 of the License, or (at your option) any later version.
//
// LibSourcey is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program. If not, see <http://www.gnu.org/licenses/>.
//

#ifndef SCY_Pluga_H
#define SCY_Pluga_H


#include "scy/base.h"


namespace scy {
namespace pluga {


// Forward declare the plugin class which must be defined externally.
class IPlugin;


// Define the API version.
// This value is incremented whenever there are ABI breaking changes.
#define SCY_PLUGIN_API_VERSION 1


#ifdef WIN32
# define SCY_PLUGIN_EXPORT __declspec(dllexport)
#else
# define SCY_PLUGIN_EXPORT // empty
#endif


// Define a type for the static function pointer.
SCY_EXTERN typedef IPlugin* (*GetPluginFunc)();


// Plugin details structure that's exposed to the application.
```

```
struct PluginDetails {
    int apiVersion;
    const char* fileName;
    const char* className;
    const char* pluginName;
    const char* pluginVersion;
    GetPluginFunc initializeFunc;
};

#define SCY_STANDARD_PLUGIN_STUFF \
    SCY_PLUGIN_API_VERSION,       \
    __FILE__

#define SCY_PLUGIN(classType, pluginName, pluginVersion)     \
  extern "C" {                                               \
      SCY_PLUGIN_EXPORT scy::pluga::IPlugin* getPlugin()     \
      {                                                      \
          static classType singleton;                        \
          return &singleton;                                 \
      }                                                      \
      SCY_PLUGIN_EXPORT scy::pluga::PluginDetails exports =  \
      {                                                      \
          SCY_STANDARD_PLUGIN_STUFF,                         \
          #classType,                                        \
          pluginName,                                        \
          pluginVersion,                                     \
          getPlugin,                                         \
      };                                                     \
  }


} } // namespace scy::pluga



#endif // SCY_Pluga_H
```

Plugin API

The plugin API defines the `IPlugin` class that's forward declared in the Plugin System API header. The `IPlugin` class is the interface that the application uses to interact with the plugin, and as such it's also the virtual base class that's extended from when implementing plugins.

Below is a bare-bones example that only implements a single `onCommand` method:

```
// testpluginapi.h


#ifndef SCY_TestPluginAPI_H
#define SCY_TestPluginAPI_H



#include "scy/pluga/pluga.h"



namespace scy {
namespace pluga {



class IPlugin
    // Virtual plugin interface.
{
public:
    IPlugin() {};
    virtual ~IPlugin() {};

    virtual bool onCommand(const char* node, const char* data, unsigned
int size) = 0;
        // Handles a command from the application.
};



} } // namespace scy::pluga



#endif
```

Implementing Plugins

Plugin implementations extend from the Plugin API interface to implement plugin functionality.

```
// testplugin.h


#ifndef SCY_TestPlugin_H
#define SCY_TestPlugin_H



#include "testpluginapi.h"



class TestPlugin: public scy::pluga::IPlugin
    // Test plugin implementation.
{
public:
    TestPlugin();
    virtual ~TestPlugin();


    virtual bool onCommand(const char* node, const char* data, unsigned
int size);
        // Handles a command from the application.
};



#endif
```

```
// testplugin.cpp


#include "testplugin.h"

#include <iostream>



SCY_PLUGIN(TestPlugin, "Test Plugin", "0.1.1")



TestPlugin::TestPlugin()

{

    std::cout << "TestPlugin: Create" << std::endl;

}



TestPlugin::~TestPlugin()

{

    std::cout << "TestPlugin: Destroy" << std::endl;

}



bool TestPlugin::onCommand(const char* node, const char* data, unsigned

int size)

{

    std::cout << "TestPlugin: Command: " << node << ": " << data <<

std::endl;

    // Process commands as required

    return true;

}
```

A Simple Application

The following is a totally minimal example application that shows how to use the LibSourcey `SharedLibrary` class to load the plugin shared library, instantiate the `IPlugin`, call it's methods, and destroy it.

```
#include "scy/pluga/pluga.h"
#include "scy/sharedlibrary.h"
#include "testpluginapi.h"
#include <iostream>
#include <assert.h>


using namespace scy;


int main(int argc, char** argv)
{
    // Set the plugin shared library location
    std::string path(SCY_INSTALL_PREFIX);
    path += "/bin/testplugin/testplugin";
#if WIN32
# ifdef _DEBUG
    path += "d.dll";
# else
    path += ".dll";
# endif
#else
    path += ".so";
#endif

    try {
        // Load the shared library
        std::cout << "Loading: " << path << std::endl;
        SharedLibrary lib;
        lib.open(path);

        // Get plugin descriptor and exports
        pluga::PluginDetails* info;
        lib.sym("exports", reinterpret_cast<void**>(&info));
        std::cout << "Plugin Info: "
            << "\n\tAPI Version: " << info->apiVersion
            << "\n\tFile Name: " << info->fileName
            << "\n\tClass Name: " << info->className
            << "\n\tPlugin Name: " << info->pluginName
            << "\n\tPlugin Version: " << info->pluginVersion
            << std::endl;

        // API Version checking
        if (info->apiVersion != SCY_PLUGIN_API_VERSION)
            throw std::runtime_error(
                util::format("Plugin ABI version mismatch. Expected %s,
```

```
got %s.",
                    SCY_PLUGIN_API_VERSION, info->apiVersion));

        // Instantiate the plugin
        auto plugin = reinterpret_cast<pluga::IPlugin*>(info-
>initializeFunc());

        // Call plugin methods
        assert(plugin->onCommand("some:command", "random:data", 11));

        // Close the plugin and free memory
        std::cout << "Closing" << std::endl;
        lib.close();
    }
    catch (std::exception& exc) {
        std::cerr << "Error: " << exc.what() << std::endl;
        assert(0);
    }

    return 0;
}
```

Installing Pluga

For more detailed instructions and full working example see the Pluga insallation guide. All that's requires is to build build LibSourcey with the Pluga module enabled.

And there you have it, a super simple C++ plugin system that you can use in your own projects. Enjoy!