"Mircea cel Batran" Naval Academy Scientific Bulletin, Volume XVI – 2013 – Issue 2
Published by "Mircea cel Batran" Naval Academy Press, Constanta, Romania

AN OOP METHOD TO UPDATE THE DIGITAL WATERMARKING APPLICATION DURING RUNNING TIME

Marius ROGOBETE[1]
Ciprian RACUCIU[2]
Cristian APOSTOL[3]
Inf. Marian PARLOAGA[4]
[1] Senior Software Eng., PhD. Std, CAMatWEB Co., Bucharest
[2] Professor Eng. PhD, Titu Maiorescu University, Bucharest
[3] Engineer, PhD, Std. Military Technical Academy, Bucharest
[4] Engineer, PhD, Std. Military Technical Academy, Bucharest

Abstract: *As many security applications should be updated without stop the running processes (e.g. monitoring, sniffers/snort or capture), the presentation shows how an application is updated during link time (adding code to the application), using specific technology. The C++ classes polymorphism is highlighted through the OS/compiler tools in a way that allows a base-class object pointer which is pointed to a inherit class to access the new added methods of new class, using the same common abstract interface known to the program at the compile time. The presented work is focused, first of all, on the practical method that allows updating the code classes of any kind of application without stop it. The real, complex digital watermarking application is then used, instead the simple application example.*

1. INTRODUCTION

The necessity to use C++ applications allow updating new class during run-time, independently of platform, has indentified that specific library could provides these facilities (e.g. plugins).

Using standard C++ programming language in order to compile it with any C++ compiler that offers good template support, it is able to be link over cross-platform, because the minimal platform specific layer that handles operating system specifics (actually Linux and Win32 supported).

2. PROBLEM OVERVIEW

Among the programming languages, C++ offers important and rich features, most important for the case, the functionality within the same time module (meaning all sources and libraries) can be exposed out without difficulties. All these features (overloading, overriding) must to be extended to run-time, therefore with shared libraries (shared dynamic object files – SO on Linux or Dynamic Link Libraries – DLL on windows), being necessary to use the right compiling/linking flags for templates, meaning to pay attention to the specific compiler documentation. However, it quickly becomes rather difficult, since there is an intricate linking process going on between the application and the loaded modules. The C++ name mangling, automatically generated templates and a rich set of compiler/linker options, make dependencies become complicated, when a code base under heavy development is linked with a DLL/SO compiled on a different system [3].

This approach usually assumes that the same compiler is used for the host and the library. To link the binaries, often also the compiler version must be similar. It could not expect to link together a template or class library compiled with G++ with an application compiled with a Microsoft based compiler.

As it's well known, C++ has never had easy to make its internal features available to the outside world in a standard way. The old style only works for global functions and variables:

extern "C" int FunctionToExport(...)

The customization of any application at run-time is enabled when it supports plugins. The host application provides services which the plugin can use, including a way for plugins to register themselves with the host application and a protocol to exchange data with plugins, they do not usually work by themselves. Of course, the host application operates independently of the plugins, making it possible to add and update plugins dynamically without needing to make changes to the host application.

A plugin is a more decoupled run-time library, not dependent on the main application, and preferably the requirements on the plugin should be a bit poorer, (they shouldn't need to know about every header file and *#define* of the applications, and they don't need mutually resolve massive amounts of global symbols). C++ doesn't provide any language or standard library support for this, but some programming language features are good starting points for.

On the Windows platform, there is Component Object Model (COM) binary-interface standard techniques as a way forward. Because it has to be language neutral, it limits (severely) what features of C++ can be exposed and then re-used inside another run-time module (including inheritance relationships). Based on a particular way of sharing the *VTables* for run-time binary objects, it works, but has not evolved in a cross-platform direction [2].

Thus the problem is how to define a middle between the compile time process, when the whole feature set with all header files are shared and the runt-time loading, when only type-less symbols could be looked up in a loaded module.

3. DYNAMIC CLASSES/OBJECTS

Let's consider the class definition [3]:

```
class Test {
public:
   Test( int i ) : m_i(i) { }
   virtual int Add( int i );
   int Sub( int i );
   int Get( ){ return m_i; }
   bool operator < (int i){ return m_i
   static int StaticFunc( );
protected:
   int m_i;
};
```

The key issue in dynamic loading is the link, applies to the members of *Test*:

- The constructor *Test(int i)* is inline so it does not generate any linking

- The virtual function *int Add(int i)* is an entry in the VTable of the class. To the compiler and linker, it is an index in this table.
- The function *Sub(int i)* does generates linking. It refers to a function implemented somewhere else.
- *Get()* is an inline function and does not generate linking.
- *operator* < is also an inline function (no linking).
- The *StaticFunc()* member requires linking. This is true for any static member.
- The member variable *m_i* is a type and offset into the binary object. It does not generate linking.
  Only static members and functions that are both non-virtual and non-inline members generate any linking [3].
  VTables - To remember what virtual functions are:
- Each class (that has one or more virtual functions) has a unique virtual function table (VTable).
- The VTable is per type, not per instance.
- A pointer to the VTable (if the object has one) is always stored first in the object. It is often referred to as the VPTR.
- A virtual member function is at run-time an index into this VTable where the address of the function to use is stored.
- A derived class has a copy of the base class VTable first in its own VTable. The new functions it introduces are stored at the end of that copy.

If an object is instantiated from the DLL/SO that owns the VTable, this is not part of the link process. The VTable itself is a symbol located inside a DLL

C++ Inheritance - Having:

```
class SBaseC {
public:
   virtual const char *GetPath( );
   virtual bool SetPath( const char *path );
};
class MBaseC : public Test, public SBaseC {
public:
   virtual int Add( int i );
   virtual bool SetPath( const char *path );
protected:
   int m_j;
};
```

MBaseC overrides a function from each base class (it has two base classes). The derived classes have:

1. A derived class inherits its default VTable from its first base class that has a VTable (call it main base class).
2. New virtual functions are appended at the end of the default VTable.
3. For other base classes with VTables, a full copy of the object is stored at an offset into the binary object, including a copy of side base class VTable.
4. The VTable of side bases can be modified (functions are overridden) but it cannot be extended.

An important part of the framework is to account offsets between multiple bases generated by different compilers. What is the most important to recognize here is that: *Inheritance (neither single nor multiple) does not generate any linking*.

Common definition - A class can say be reused and implemented by both a host application and a plugin if is defined:

- with one or more (non-template, non-virtual) base classes
- with any number of virtual functions
- with operators that are either virtual or inline
- and any inline function
- with any non-static data members

The code made by one compiler may use such a class compiled from another, It works all the time for function members, also in the cross-compiler case. For data members, it works as long as the compilers on each side have used the same sizes and padding for the data members.

When moving (casting) between base classes, the address offset must be calculated based on offsets generated by the source (plugin) compiler. This is in addition to the old *extern "C" MyFunction(...)* style.

Link at run-time - It is clear that instances of a class fulfilling above specs can be used by both the host application and the plugin, without requiring a linking process. Since a call across the plugin boundary is essentially typeless, we cannot communicate the type directly to the plugin as a C++ type.

Plugin object creation - On the host, the plugin creates an instance of *Test*:

It is used a factory function in the plugin that take the type encoded as a string and an integer:

```
extern "C" void* CreatePluginObject( const char *type_name, int type_id )
   if( !strcmp(type_name,"Test") &&
      type_id == TEST_TYPE_ID )
      return new Test;
   else return NULL;
}
On the host side:
typedef void* (*PluginFactoryFn)(const char *type, int id);
// First locate CreatePluginObject inside a loaded DLL
PluginFactoryFn create_fn = /* Locate it */;
// Now create object
Test *pae = (Test*)create_fn( "Test",
      TEST_TYPE_ID );
```

The framework automates this conversion step, allowing to use the expression:
Test *pe = do_new<Test>;

Plugin object destruction - To keep cross-compiler compatibility, there are some points to consider here:

- It's not sure that the host and the plugin share the same memory allocator (on Windows this is often not the case). Don't use C++ *delete* on plugin objects.
- Virtual destructors are used in different ways by different compilers.

The host must be sure that a plugin object is 'recycled' by the same plugin that created it. Each object that is created has a virtual member function *doDestroy()*:
RTObj *pdo = … /* Create object and use it */;
pdo->doDestroy(); // End of object
The *RTObj* is a base class for the objects created by a plugin and the framework can use any classes. The only run-time linking to do is to lookup these factory functions inside the plugin DLL (and possibly some other init/exit functions).

4. IMPLEMENTATION

The main properties of the RTObj library implementation solution, is described with focus on the plugin/linking problem [2]. The library/plugin cross compiler can be a different one then the main application compiler. All casting between types is allways done based on offsets from the source (library) compiler. The RTObj supports ordinary C++ classes across the plugin boundary. Any class consists of:

- Zero, one or more base classes/interfaces
- Virtual functions (argument overloading not supported)
- Inline functions
- Operators
- Data members (pay attention to member alignment!)

A pretty large subset of the C++ class concepts can be used over the boundary. This is what cannot be used:

- Non-virtual member functions implemented in a separate source file
- Static members (functions, data)

The model of the object from a plugin represents a full C++ object, including the possibility of having multiple nested base classes. At source code level, a tagging scheme is used to decide which bases to expose. The whole (exposed part) of the inheritance tree is communicated to users of the object. The object is usually accessed using a single inheritance interface/class. Using a cast operation (query type) one can move between the different interfaces/sub-objects that are implemented. An object can implement a number of interfaces and/or classes. To query an object for another type, the C++ template is used:
template<class U, class T> U do_cast(T t)
It operates the same way as C++ *dynamic_cast*<> and provides typed safe casts across the plugin boundary. *do_cast* (and related functions) provides similar functionality to *QueryInterface* in COM.
An example:

```
RTI pdi = /* Basic RTI pointer from somewhere */;
RTSharedI pdsi = do_cast<RTSharedI*>(pdi)
```

Types - The library introduces a small interface and class collection, based on *RTI* (a class which knows its own type and can be queried for other types). Both classes based on *RTI* and arbitrary classes with virtual methods may be used across the plugin boundary. When are used classes derived from *RTI*, a separate registration step may be skipped, since a *RTI* object always knows its own type. The provided classes derived from *RTI* also provides a certain way of instantiating and destroying objects (*RTObj*), handling objects with shared ownership (*RTSharedI*), and weak references. When are using arbitrary classes, they must have **at least one** virtual member function. The library provides templates that safely detect if an object has a VTable or not. To use such objects across a plugin boundary, one instance of the type must be registered first.

Identifiers - Types are identified based on the pair of **t**ype string and type identifier (32-bit integer). Plugins role could be to use types from the main application (as long as it has headers for it) and also from other loaded plugins. It can also instantiate plugin objects (from itself, other plugins or the main app). A run-time string class, *RTStr* (in itself a plugin object) is provided, giving plugins a way to deal with Unicode strings. The library relies on the default way of using VTables in C++ together with a binary type description structure. This is a simple binary scheme. The plugin classes could be used from any language that can use these. A C implementation is straight forward (an object would be a structure with the first member being a pointer to an array of functions). Also, a plugin class could be implemented in another language and used from C++. Inline functions cannot be shared with another language (they are really compiled on both host and plugin side).

Requirements - The C++ compiler has to meet the following requirements to be able to compile/link the library:

- To use vtables in the default way (one pointer per function, first function at index 0, new functions are stored in declaration order)
- Support for **extern "C"** style of exposing non-mangled function names
- Support for __**cdecl** function calling convention

When a library is compiled, this information is stored and made available at load time, so an incompatible library can be detected.Because the compilers implement the virtual destructors in different ways, they are not used across plugin boundaries.The calling convention can be configured when the library is compiled, some other convention could be used as long as the main and plugin compiler agree on it.

5. RTOBJ LIBRARY

The RTObj is an cross-platform library that uses the run-time plugin approach described before. The mechanisms used are generic and simple, the library fills many gaps and make it straight-forward to use plugins inside a C++ application [3],[2].

The library provides:

- A small class hierarchy (*VObj, RTI, RTObj, RTSharedI*) establishing some common ground between a host and a plugin. The RTObj library also works with classes that are not rooted in this hierachy.
- A type description facility that allows types to be defined and shared by both host and plugins (*RTObjType*).
- A way to convert C++ types to a plugin library (*doTypeInfo*, *DO_DECL_TYPE_INFO*).
- Instantiating C++ objects from plugins (*do_new<T>*).
- A plugin library loading/unloading mechanism (*RTObjLib*).
- C-level functions to handle objects.
- Other practical C++ classes and templates for objects.

All classes are defined in *RTObj.h*. The library is based on some properties of objects with VTables:

- The VPTR is always stored first in a binary object
- VTables are shared by all instances of a class, but not with instances of any other class (so it provides a type identifier).

In C++ there is not a built-in way to denote these classes. However, we define the class *VObj* to represent an object with a VTable with unknown size and methods. *VObj* in that sense becomes the 'stipulated' root class for all classes that contain one or more virtual functions. To determine if a class is a *VObj* or not, these templates can be used:

```
bool has_vtable = IsVObj<SomeType>::v;
template<class T>
VObj* to_vobj( T* pt );
```

This provides type safety so that we cannot try to convert say a *char** to an interface pointer (the compiler would give an error). The RTI class provides a way to know its type (*doGetType*) and for asking about other types it supports (*doGetObj*). To ask if a *RTI* supports the *RTStr* interface:

```
RTI *pdi = /* Wherever pointer comes from */;
RTStr *pds = pdi->doGetObj("RTStr");
```

This is equivalent to:

```
RTStr *pds = do_cast<RTStr*>(pdi);
```

The *RTI* class has an advantage over *VObj*:

- It knows its own derived type

In contrast, to find the type of a *VObj*, a lookup into a global table, using the VPTR, has to be made (and works only after the types has been registered).

The *RTObj* interface represents an object that can be created an destroyed. It represents an object owned from a single point. Usually the functions *doDestroy* and *doDtor* would be implemented like:

```
void AClass::doDestroy(){ ::delete this; }
void AClass::doDtor(){ this->~AClass(); }
```

The *RTSharedI* interface represents an object with shared ownership. *doAddRef* and *doRelease* increases and decreases the ownership counter:

```
virtual void docall doDestroy( ) {
  if( !m_ref_cnt )
    ::delete this;
  else
    SetError(DOERR_DESTROY_ON_NON_ZERO_REF,
        "RTSharedI - Destroy on non-zero ref");
}
```

6. USING REAL-TIME LIBRARY WITH WATERMARKING APPLICATION

A set of plugin libraries are created. Then, they are used from a watermarking main application [4],[6]. The plugin object will be instantiated, used as C++ classes and queried for supported types.

Create an interface to manage data for watermarking/image apps:

```
#include <string.h>
class RTStr;
/* %%RTOBJ class(RTI)   <---Directive to parser preprocessor */
class WatermarkI : public RTObj {
public:
  /* RTI methods       <---Implement GetType and Destroy - for all RTObj:s */
  virtual RTObjType* docall doGetType( ) const;
  virtual void docall doDestroy( ) { delete this; }

  /* WatermarkI methods    <---Add our new methods */
  virtual const char* docall GetFormat( ) const = 0;
  virtual int docall GetSize() const = 0;

  virtual bool docall SetFormat( const char *name ) = 0;
  virtual bool docall SetSize(int age) = 0;

  /* ---Simple default inline implementation of operator */
  virtual bool docall operator<( const WatermarkI& other ) const {
    return strcmp(GetFormat(),other.GetFormat()) < 0;
  }

  /* ---Non-virtual, inline convenience function*/
  /*Derived cannot override. */
  WatermarkI& operator=( const WatermarkI& other ) {
    SetSize( other.GetSize() );
    SetFormat( other.GeFormat() );
    return *this;
  }
};
```

Using a parser preprocessor on the code file (with option to write generated code directly into the file) is added a section at the beginning of the header file:

```
// %%DYNOBJ section general
// This section is auto-generated and manual changes will
// be lost when regenerated!!
```

```
#ifdef DO_IMPLEMENT_WATERMARKI
#define DO_IMPLEMENTING // If app has not defined it already
#endif
#include "RTObj/RTObj.h"

// These are general definitions & declarations used
// on both the user side [main program]
// and the implementor [library] side.

// --- Integer type ids for defined interfaces/classes ---
#define WATERMARKI_TYPE_ID 0x519C8A00

// --- Forward class declarations ---
class WatermarkI;

// --- For each declared class, doTypeInfo template specializations ---
// This allows creating objects from a C++ types and in run-time casts
DO_DECL_TYPE_INFO(WatermarkI,WATERMARKI_TYPE_ID);

// %%RTOBJ section end
```

Now it's necessary to convert from a C++ *WatermarkI* type to the type strings and type ID:s that are used across the plugin boundary. If we would like to move this section we're free to do that. The next time the preprocessor is run on the same file, it will keep the section where we put it. The code has been inserted at the end of the file

Next we create a source file that implements the interface (*WatermarkImpl1.cpp*).

Finally we create the main application (*main1.cpp*) that uses the plugin:

```
#include <stdio.h>
#include <stdlib.h>

// RTObj support
#include "RTObj/RTObj.h"
#include "RTObj/RTObjLib.h"
#include "RTObj/DoBase.hpp"
// Interfaces we're using
#include "WatermarkI.h"
#include "PixFormatI.h"

int main( ) {
  // Check that things have started OK
  if( !doVerifyInit() ){
    printf("Failed DoVerifyInit\n");
    exit(-1);
  }
```

Now we want to start using the plugin. For this, we use RTObjLib which wraps a cross-platform run-time (DLL/SO) library loader

```
  // Load library
  RTObjLib lpimpl1( "pimpl1", true );
  if( lpimpl1.GetState()!=DOLIB_INIT ){
    printf( "Could not load library (pimpl1): status (%d)\n",
        lpimpl1.GetState() );
    exit( -1 );
  }

  // Create object
  WatermarkI *pp = (WatermarkI*)lpimpl1.Create("WatermarkI",WATERMARKI_TYPE_ID);
  if( pp ) {
    pp->SetFormat( "JPEG" );
    pp->SetSize( 648720 );
    ...
```

We have instantiated the object the 'raw' way here, giving type name and ID to *RTObjLib*. After that, it is just to start using the object as any standard C++ object.

We next query the object for an interface using *do_cast*:

```
  WatermarkI *pi = do_cast<WatermarkI*>(pp);
  if( pi )
    ;// Use the interface

  pp->doDestroy();
  return 0;
}
```

The template do_cast<T> takes care of the details of checking if *WatermarkI* is supported. It transforms the C++ type to type name and ID. Using type information from the plugin, it can walk the class layout and return an adjusted interface pointer. It is important that any address offsets applied inside objects are always based on information from the plugin compiler. When is used an interface pointer returned, it can only assume it is valid for the duration of the current function. It is not needed to release it in any way.

Finally, we delete the object using *RTObj::doDestroy* (which will recycle it in the memory manager of the plugin that created it).

7. CONCLUSIONS

Based on the sample example that has used a watermarking application, the real-time object dynamic library provides powerful support in order to use run-time plugins inside a C++ application.

The plugins can typically be used to dynamically update the specific signature for digital watermarking. The ownership identification for any multimedia is changed on request, without break the emission process at all. Thus the copyright watermarking information is updated but also the algorithm could be modified.

We have used it in a mobile agent image/video application as module of a large 2D/3D network framework.

REFERENCES

[1] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis Mutandis: Safe and predictable dynamic software updating. In *Proc. POPL*, 2005.
[2] G. Hjalmtysson and R. Gray. Dynamic C++ classes, a lightweight mechanism to update code in a running program. In *Proc. USENIX ATC*, 1998.
[3] M. W. Hicks. *Dynamic Software Updating*. PhD thesis, The University of Pennsylvania, August 2001.
[4] M. Rogobete, C. Răcuciu, First and second order image statistics in specific image artifact detection, International Conference on Innovative Technologies, 2012.
[5] T Kobayashi, N Otsu, Image feature extraction using gradient local auto-correlations, Computer Vision–ECCV 2008, 2008 - Springer
[6] M. Rogobete, C. Răcuciu, "Original methodology and algorithm able to identify visible noisy in still images and video stream", MegaByte Journal of Titu Maiorescu Univeristy, no.12/2012, Bucharest, Romania (in progress)