CASE
STUDY

WRAPPER C/C++
DEVELOPING A C WRAPPER API FOR OO C++ CODE

2016.11/0.1

WRAPPER C/C++

DEVELOPING A C WRAPPER API FOR OO C++ CODE,

Develop a set of C APIs that will wrap around our existing C++ APIs to access our core logic (written in object-oriented C++). This will essentially be a glue API that allows our C++ logic to be usable by other languages. We based this development on a specific design pattern called hourglass pattern.

All we ended up doing was:

- Every object is passed about in C an opaque handle.
- Constructors and destructors are wrapped in pure functions
- Member functions are pure functions.
- Other built-ins are mapped to C equivalents where possible.

FIRST APPROACH

So a class like this (C++ header)

```
class MyClass
{
 public:
        explicit MyClass( std::string & s );
        ~MyClass();
        int doSomething( int j );
}
```

Would map to a C interface like this (C header):

```
struct HMyClass; // An opaque type that we'll use as a handle
typedef struct HMyClass HMyClass;

HMyClass * myStruct_create( const char * s );
void myStruct_destroy( HMyClass * v );
int myStruct_doSomething( HMyClass * v, int i );
```

The implementation of the interface would look like this (C++ source)

```
#include "MyClass.h"

extern "C" {
   HMyClass * myStruct_create( const char * s ) {
            return reinterpret_cast<HMyClass*>( new MyClass( s ) );
   }

   void myStruct_destroy( HMyClass * v ) {
            delete reinterpret_cast<HMyClass*>(v);
   }

    int myStruct_doSomething( HMyClass * v, int i ) {
            return reinterpret_cast<HMyClass*>(v)->doSomething(i);
    }
}
```

```
* reinterpret_cast only guarantees that if you cast a pointer to a different type, and then reinterpret_cast it back to the original type, you get the original value…

* static_casting a pointer to and from void* preserves the address
```

Exceptions are not part of the C ABI so you cannot let Exceptions ever be thrown past the C++ code. So your header is going to look like this:

```
#ifdef __cplusplus
extern "C"
{
#endif
   void * myStruct_create( const char * s );
   void myStruct_destroy( void * v );
   int myStruct_doSomething( void * v, int i );
#ifdef __cplusplus
}
#endif
```

And your wrapper's .cpp file will look like this:

```
void * myStruct_create( const char * s ) {
   MyStruct * ms = NULL;
   try { /* The constructor for std::string may throw */
     ms = new MyStruct(s);
   } catch (...) {}
     return static_cast<void*>( ms );
}
```

```
void myStruct_destroy( void * v ) {
   MyStruct * ms = static_cast<MyStruct*>(v);
   delete ms;
}



int myStruct_doSomething( void * v, int i ) {
   MyStruct * ms = static_cast<MyStruct*>(v);
   int ret_value = -1; /* Assuming that a negative value means error */
   try {
      ret_value = ms->doSomething(i);
   } catch (...) {}
   return ret_value;
}
```

Even better: If you know that all you need as a **single instance** of MyStruct, don't take the risk of dealing with void pointers being passed to your API. Do something like this instead:

```
static MyStruct * _ms = NULL;

int myStruct_create( const char * s ) {
   int ret_value = -1; /* error */
   try { /* The constructor for std::string may throw */
      _ms = new MyStruct(s);
      ret_value = 0; /* success */
   } catch (...) {}
   return ret_value;
}

void myStruct_destroy() {
   if (_ms != NULL) {
      delete _ms;
   }
}

int myStruct_doSomething( int i ) {
   int ret_value = -1; /* Assuming that a negative value means error */
   if (_ms != NULL) {
      try {
         ret_value = _ms->doSomething(i);
      } catch (...) {}
   }
   return ret_value;
}
```

OTHER POINTS

- Namespace should be replaced by a prefix for each function.
- To be continued…