Mixing Objective-C, C++ and Objective-C++: an Updated Summary

Originally published on 25th May 2012, updated on 15th July 2012.

Quite some time ago, I ran into the situation of including a C++ library in an Objective-C project. I failed to find any sensible documentation on the subject, so I came up with a solution myself and eventually wrote it up in an article. That article went on to become something of a sleeper hit (by my modest standards anyway) and is to this day one of the highest-ranked results for Objective-C++ and related keywords on Google.

Since then, Apple has switched to the LLVM-based clang as the primary compiler for Mac and iOS development. One of the effects of this has been an accelerated pace of changes to the Objective-C language, compared to the rather more glacial rate of change under the GCC regime. One particular change has caused my old article to no longer be up-to-date. This, along with the steady stream of clarification questions I receive about it, has prompted me to write this new article.

Recap of the problem

To save you going through the old article, here's the issue: let's say you have some existing C++ code, a library perhaps, and you want to use it in an Objective-C application. Typically, your C++ code will define some class you'd like to use. You *could* switch your whole project to Objective-C++ by renaming all the .m files to .mm, and freely mix C++ and Objective-C. That's certainly an option, but the two worlds are quite different, so such "deep" mixing can become awkward.

So usually you'll want to wrap the C++ types and functions with Objective-C equivalents that you can use in the rest of your project. Let's say you have a C++ class called `CppObject`, defined in CppObject.h:

```
#include <string>
class CppObject
{
public:
  void ExampleMethod(const std::string& str);
  // constructor, destructor, other members, etc.
};
```

You can have C++-typed members in an Objective-C class, so the typical first attempt is to do this with your wrapper class, `ObjcObject` - in ObjcObject.h:

```
#import <Foundation/Foundation.h>
#import "CppObject.h"

@interface ObjcObject : NSObject {
  CppObject wrapped;
}
- (void)exampleMethodWithString:(NSString*)str;
```

```
// other wrapped methods and properties
@end
```

And then implementing the methods in ObjcObject.mm. Many are then surprised to get preprocessor and compile errors in ObjcObject.h and CppObject.h when they `#import "ObjcObject.h"` from a pure Objective-C (.m) file directly or indirectly via another header (.h) file. The thing to bear in mind is that the preprocessor basically just does text substitution, so `#include` and `#import` directives are essentially equivalent to recursively copy-and-pasting the contents of the file in question into the location of the directive. So in this example, if you `#import "ObjcObject.h"` you're essentially inserting the following code:

```
// [lots and lots of Objective-C code from Foundation/Foundation.h]
// [fail to include <string>] as that header is not in the include path outside of C++ mode
class CppObject
{
public:
  void ExampleMethod(const std::string& str);
  // constructor, destructor, other members, etc.
};

@interface ObjcObject : NSObject {
  CppObject wrapped;
}
- (void)exampleMethodWithString:(NSString*)str;
// other wrapped methods and properties
@end
```

The compiler will get enormously confused by `class CppObject` and the block following it, as that's simply not valid Objective-C syntax. The error will typically be something like

```
Unknown type name 'class'; did you mean 'Class'?
```

as there is no `class` keyword in Objective-C. So to be compatible with Objective-C, our Objective-C++ class's header file must contain only Objective-C code, absolutely no C++ - this mainly affects types in particular (like the CppObject class type here).

Keeping your headers clean

In the old article, I talked through a few solutions to this, so I won't reiterate them here. The nicest one at the time was the PIMPL idiom. It continues to work well today, and is still the best way for the opposite problem of wrapping Objective-C with C++ (more on that later on). However, with clang, there is a new way to keep C++ out of your Objective-C headers: ivars in class extensions.

Class extensions (not to be confused with *categories*) have existed in Objective-C for a while: they let you declare additional parts of the class's interface outside the public header before the `@implementation` block. As such, the only sensible place to put them is just above said block, e.g. ObjcObject.mm:

```
#import "ObjcObject.h"
@interface ObjcObject () // note the empty parentheses
- (void)methodWeDontWantInTheHeaderFile;
@end
@implementation ObjcObject
// etc.
```

This much already worked with GCC, but with clang, you can also add an ivar block to it. This means we can declare any instance variables with C++ types in the extension, or at the start of the `@implementation` block. In our case, we can reduce the ObjcObject.h file to this:

```
#import <Foundation/Foundation.h>

@interface ObjcObject : NSObject
```

```
- (void)exampleMethodWithString:(NSString*)str;
// other wrapped methods and properties
@end
```

The missing parts all move to the class extension in the implementation file (ObjcObject.mm):

```
#import "ObjcObject.h"
#import "CppObject.h"
@interface ObjcObject () {
  CppObject wrapped;
}
@end

@implementation ObjcObject
- (void)exampleMethodWithString:(NSString*)str
{
  // NOTE: if str is nil this will produce an empty C++ string
  // instead of dereferencing the NULL pointer from UTF8String.
  std::string cpp_str([str UTF8String], [str lengthOfBytesUsingEncoding:NSUTF8StringEncoding]);
  wrapped.ExampleMethod(cpp_str);
}
```

Alternatively, if we don't need the interface extension to declare any extra properties and methods, the ivar block can also live at the start of the `@implementation`:

```
#import "ObjcObject.h"
#import "CppObject.h"

@implementation ObjcObject {
  CppObject wrapped;
}

- (void)exampleMethodWithString:(NSString*)str
{
  // NOTE: if str is nil this will produce an empty C++ string
  // instead of dereferencing the NULL pointer from UTF8String.
  std::string cpp_str([str UTF8String], [str lengthOfBytesUsingEncoding:NSUTF8StringEncoding]);
  wrapped.ExampleMethod(cpp_str);
}
```

Either way, we now `#import "ObjcObject.h"` to our heart's content and use `ObjcObject` like any other Objective-C class. The CppObject instance for the `wrapped` ivar will be constructed using the *default constructor* when you *alloc* (**not** init) an ObjcObject, the destructor will be called on *dealloc*. This often isn't what you want, particularly if there isn't a (public) default constructor at all, in which case the code will fail to compile.

Managing the wrapped C++ object's lifecycle

The solution is to manually trigger construction via the `new` keyword, e.g.

```
@interface ObjcObject () {
  CppObject* wrapped; // Pointer! Will be initialised to NULL by alloc.
}
@end
@implementation ObjcObject
- (id)initWithSize:(int)size
{
  self = [super init];
  if (self)
  {
```

```
    wrapped = new CppObject(size);
    if (!wrapped) self = nil;
  }
  return self;
}
//...
```

If using C++ exceptions, you may want to wrap the construction in a `try {...} catch {...}` block and handle any construction errors. With explicit construction, we also need to explicitly destroy the wrapped object:

```
- (void)dealloc
{
  delete wrapped;
  [super dealloc]; // omit if using ARC
}
```

Note that the extra level of indirection involves an extra memory allocation. Objective-C heavily allocates and frees memory all over the place, so this one extra allocation shouldn't be a big deal. If it is, you can use placement `new` instead, and reserve memory within the Objective-C object via an extra `char wrapped_mem[sizeof(CppObject)];` ivar, creating the instance using `wrapped = new(wrapped_mem) CppObject();` and destroying it via an explicit destructor call: `if (wrapped) wrapped->~CppObject();`. As with any use of placement new, though, you'd better have a good reason for it. Placement `new` returns a pointer to the constructed object. I would personally keep that (typed) pointer in an ivar just as with regular `new`. The address will normally coincide with the start of the char array, so you could get away with casting that instead.

Wrapping up

Now you'll probably want to wrap a bunch of member functions with Objective-C methods, and public fields with properties whose getters and setters forward to the C++ object. Make sure that your wrapper methods only return and take parameters with C or Objective-C types. You may need to do some conversions or wrap some more C++ types. Don't forget Objective-C's special nil semantics don't exist in C++: NULL pointers must not be dereferenced.

The reverse: using Objective-C classes from C++ code

I've had some email regarding the opposite: calling into Objective-C from C++. Again the problem lies with header files. You don't want to pollute the C++ header with Objective-C types, or it can't be `#include`d from pure C++. Let's say we want to wrap the Objective-C class `ABCWidget`, declared in ABCWidget.h:

```
#import <Foundation/Foundation.h>
@interface ABCWidget
- (void)init;
- (void)reticulate;
// etc.
@end
```

Once again, this kind of class definition will work in Objective-C++, but this time not in pure C++:

```
#import "ABCWidget.h"
namespace abc
{
  class Widget
  {
    ABCWidget* wrapped;
  public:
    Widget();
    ~Widget();
    void Reticulate();
  };
}
```

A pure C++ compiler will trip over the code in Foundation.h and eventually the `@interface` block for `ABCWidget`.

Some things never change: PIMPL

There's no such thing as a class extension in C++, so that trick won't work. PIMPL, on the other hand, works just fine and is actually quite commonly used in plain C++ anyway. In our case, we reduce the C++ class to its bare minimum:

```
namespace abc
{
  struct WidgetImpl;
  class Widget
  {
    WidgetImpl* impl;
  public:
    Widget();
    ~Widget();
    void Reticulate();
  };
}
```

And then, in Widget.mm:

```
#include "Widget.hpp"
#import "ABCWidget.h"
namespace abc
{
  struct WidgetImpl
  {
    ABCWidget* wrapped;
  };
  Widget::Widget() :
    impl(new WidgetImpl)
  {
    impl->wrapped = [[ABCWidget alloc] init];
  }
  Widget::~Widget()
  {
    if (impl)
      [impl->wrapped release];
    delete impl;
  }
  void Widget::Reticulate()
  {
    [impl->wrapped reticulate];
  }
}
```

This is mostly self-explanatory; the reason it works is that a forward declaration of a struct or class suffices for declaring variables or members as *pointers* to such struct or class objects. We only dereference the `impl` pointer inside Widget.mm after we fully define the `WidgetImpl` struct type.

Notice that I `release` the wrapped object in the destructor. Even if you use ARC in your project, I recommend you disable it for C++-heavy Objective-C++ files like this one. You *can* make your C++ code behave itself even with ARC, but it'll often be more work than just putting in the `release` and `retain` calls. You can disable ARC for individual files in XCode under the 'Build Phases' tab in the build target's properties. Fold out the 'Compile Sources' section and add `-fno-objc-arc` to the compiler flags for the file(s) in question.

A shortcut for wrapping Objective-C objects in C++

You may have noticed that the PIMPL solution uses *two* levels of indirection. If the wrapper is as thin as the one in this example, that's probably overkill. Although Objective-C types can generally not be used in plain C++, there are a few types that are actually defined in C. The `id` type is one of them, and it's declared in the `<objc/objc-runtime.h>` header. You lose what little type safety Objective-C gives you, but it does mean you can place your object pointer directly into the C++ class definition:

```
#include <objc/objc-runtime.h>
namespace abc
{
  class Widget
  {
    id /* ABCWidget* */ wrapped;
  public:
    Widget();
    ~Widget();
    void Reticulate();
  };
}
```

Sending messages to id isn't really advisable, as you lose a lot of the compiler's checking mechanism, particularly in the presence of ambiguities between differently-typed methods with the same selector (name) in different classes. So:

```
#include "Widget.hpp"
#import "ABCWidget.h"
namespace abc
{
  Widget::Widget() :
    wrapped([[ABCWidget alloc] init])
  {
  }
  Widget::~Widget()
  {
    [(ABCWidget*)impl release];
  }
  void Widget::Reticulate()
  {
    [(ABCWidget*)impl reticulate];
  }
}
```

Casting like this all the time is tedious and can easily hide bugs in your code, so let's try to do a better job in the header:

```
#ifdef __OBJC__
@class ABCWidget;
#else
typedef struct objc_object ABCWidget;
#endif

namespace abc
{
  class Widget
  {
    ABCWidget* wrapped;
  public:
    Widget();
    ~Widget();
```

```
    void Reticulate();
  };
}
```

So, if this header is `#import`ed in a .mm file, the compiler is fully aware of the specific class type. If `#include`d in pure C++ mode, `ABCWidget*` is identical to the `id` type: `id` is defined as `typedef struct objc_object* id;`. The `#ifdef` block can of course be further tidied up into a reusable macro:

```
#ifdef __OBJC__
#define OBJC_CLASS(name) @class name
#else
#define OBJC_CLASS(name) typedef struct objc_object name
#endif
```

We can now forward-declare Objective-C classes in headers usable by all 4 languages:

```
OBJC_CLASS(ABCWidget);
```

Acknowledgements

Many thanks to Christopher Atlan, Uli Kusterer and Jedd Haberstro for their suggestions and corrections after reading drafts of this article.

Thanks to Rick Mann for making a suggestion that prompted me to come up with the final version for wrapping Objective-C classes with C++.