Strategies for Using C++ in Objective-C Projects (and vice versa)

Update (May 2012): while nothing in this article is incorrect, there have been some changes to Objective-C since clang became Apple's primary compiler. This means there is now an easier way to combine C++ and Objective-C than the techniques proposed here, *as long as you're using clang and don't need to maintain GCC compatibility.* I have written about the new technique and the feature enabling it in a new article. For a more extensive explanation and some alternative solutions, all of which still work with GCC, read on.

~~If you're in a hurry and want to get straight to the solution of embedding C++ objects in Objective-C classes without tainting the header files so they can still be included from plain Objective-C, you can skip straight to the conclusion showing the solution to use in ~95% of cases. The rest of the article contains deeper analysis of the issue at hand and alternative approaches to solving it.~~

Why mix Objective-C with C++?

When using Objective-C for whatever reason, typically for iOS or Mac development, I've often encountered situations where I wanted to incorporate C++ in the project in some way. Sometimes the best library for the job happens to be written in C++. Sometimes the solution to a problem is more succinctly implemented using C++. The most obvious use are C++ templates, which can save you from repeating boilerplate code. Maybe less obviously, I find that Objective-C is sometimes *too* object oriented. This is obviously heresy among the "everything is an object" folks, but for non-trivial data structures, I often find classical object orientation unwieldy, and C's structs just a bit too weak. C++'s model is a continuum.

Objective-C also is quite assertive about memory management, which can get in the way, at least in the absence of garbage collection. The STL (and its newer `shared_ptr` extension) often lets you forget about that issue altogether, or to concentrate it in constructors and destructors, rather than littering your code with `retain` and `release`. It is of course a matter of taste and applies differently to different situations; automating memory management tends to be most helpful in code with complex data structures, or code which is heavily algorithmic.

Another good reason for mixing Objective-C with C++ is the opposite situation: the need to use Objective-C libraries, such as those for the Apple platforms, from a C++ project. One common scenario for this is porting a game or engine to those platforms, and most of the following techniques can be applied in those cases too.

Finally, you might want to use C++ for performance reasons. The flexibility of Objective-C messaging adds some overhead compared to most implementations of C++ virtual functions, even with the method caching techniques used in modern runtimes. Objective-C objects don't have an equivalent of C++'s non-virtual functions, which are faster still. This can be relevant for optimising performance hotspots.

The lowest common denominator: C

One option for using the two languages in the same project is to separate them completely. You only allow them to communicate via a pure C interface, thus avoiding mixing the languages altogether. The code using the C++ library goes in a .cpp file, the code calling it is pure Objective-C (.m), the interface is declared in a C header, and the C++ side implements its interface with `extern "C"` functions.

This will work quite well in simple cases, but more likely than not you'll be writing quite a bit of wrapper code. Anyone with experience writing dynamically loadable C++ libraries via a public C interface knows this all too well. [1] Virtually all Objective-C seems to be compiled with GCC or clang nowadays. Both compilers support Objective-C++, usually a better means for mixing the languages.

Objective-C++ and the trouble with header files

At first glance, using the Objective-C++ dialect looks like a straightforward approach. It is the result of mashing C++ and Objective-C together in the same compiler, and robust implementations exist in GCC and now clang. Considering just how different the details of Objective-C and C++ are, the GCC hackers have done a great job of it. But as you start renaming your .m files to .mm to introduce chunks of C++, you quickly realise it's not quite so simple.

Header files and the C preprocessor in general have caused headaches for C, C++ and Objective-C programmers for decades. It gets worse when you try to mix the languages. Say you wanted to use the STL's `map` in an Objective-C class in your project. Apple's Foundation libraries to my knowledge don't contain a sorted, tree-based map; one of our StyleKit Components needs exactly that, for example. So we simply create an instance variable for the map in our class and away we go:

```
#include <map>
@interface MyClass : NSObject {
  @private
  std::map<int, id> lookupTable;
}
// ...
@end
```

However, `std::map<int, id>` [2] only makes sense to a C++-aware compiler, and only after an `#include <map>` [3], so this header now can only be `#import`ed from Objective-C++ files. Any code using this class now needs to be converted to Objective-C++ itself, and importing from other headers leads to a cascade effect that quickly encompasses the whole project.

In some cases, this may be acceptable. However, switching a whole project or large parts of it across just to introduce a library which is used in one location is not only excessive; if you're the only one who knows C++ on a project with multiple Objective-C programmers, you might find this to be an unpopular idea. It might also cause issues in the same way that compiling pure C code with a C++ compiler rarely is completely hassle-free. Moreover, it means that code isn't automatically reusable in other Objective-C projects.

In most cases, using Objective-C++ only where necessary is the way to go, keeping the majority of the code in pure Objective-C or C++, but the best way of doing so was not immediately apparent to

me. I also found remarkably little mention of the issue on the web.

Shooting roughly in the direction of your foot: `void*`

Given these problems, the objective is to remove all trace of C++, mainly member types, from header files. The typical C way of hiding types is to use a pointer to void. This will certainly work here, too.

```
@interface MyClass : NSObject {
  @private
  // is actually a  std::map<int, id>*
  void* lookupTable;
}
// ...
@end
```

In the code using the table, we always have to cast to the correct type using `static_cast<std::map<int, id>*>(lookupTable)` or `((std::map<int, id>*)lookupTable)`, which is annoying at best. If the actual type of the member ends up changing, all the casts must be changed manually - an error prone process. With a growing number of members, keeping track of the correct types becomes infeasible. You really are getting the worst of both worlds from static and dynamic typing. If you use this approach when dealing with objects in a class hierarchy, you're dicing with death outright due to the possibility that an `A*` and a `B*` to the same object don't have identical `void*` representations.[4]

Suffice to say, we can do better.

Conditional compilation

Losing type information sucks, but since we can only use the C++ typed fields from Objective-C++ code, and the pure Objective-C compiler only needs to be aware of their presence for the purposes of memory layout, we can provide 2 different versions of the code. The preprocessor symbol `__cplusplus` is defined in Objective-C++ mode, so how about this:

```
#ifdef __cplusplus
#include <map>
#endif
@interface MyClass : NSObject {
  @private
#ifdef __cplusplus
  std::map<int, id>* lookupTable;
#else
  void* lookupTable;
#endif
}
// ...
@end
```

It's not pretty, but it's *much* easier to work with. The C++ standard probably doesn't guarantee that a class-pointer and a void-pointer have the same memory properties, but Objective-C++ is a non-standard GNU/Apple thing anyway. I've only found pointers to virtual member functions to be a problem when converting to void* in practice, and the compiler will complain loudly if you attempt this. If you're worried, use `static_cast<>` instead of C-style casts.

Still, C happily casts `void*` to other pointer types implicitly, so it may be preferable to replace the C part of the #ifdef with a pointer to an opaque struct with a unique and recogniseable name, e.g. `struct MyPrefix_std_map_int_id`. You can even define a macro which expands to the correct type depending on the compiler's language. You can't auto-mangle templated types in a C macro though, and you'll struggle with multiple levels of namespace nesting.

```
#ifdef __cplusplus
#define OPAQUE_CPP_TYPE(cpptype, ctype) cpptype
#else
#define OPAQUE_CPP_TYPE(cpptype, ctype) struct ctype
#endif

// ...

#ifdef __cplusplus
#include <map>
#endif

@interface MyClass : NSObject {
  @private
  OPAQUE_CPP_TYPE(std::map<int, id>, cpp_std_map_int_id)* lookupTable;
}
// ...
@end
```

You can't avoid conditionally including C++ headers with this method, and you may confuse/upset those who don't know/like C++, and it's all rather ugly. Luckily, there are other options.

Abstract classes, interfaces and protocols

As a C++ programmer, you're probably familiar with pure virtual functions and, as a consequence, abstract classes. Other langages such as Java and C# have an explicit "interface" concept which deliberately leaves out any implementation details. Hiding the implementation is exactly what we're trying to do, so can we use a similar pattern here?

Recent versions of Objective-C do support *protocols*, which are similar to Java/C# interfaces in spirit, if not in syntax. [5] We could specify our class's public methods in a protocol in the header and specify and implement the concrete class conforming to said protocol in private code. This works well for instance methods, but there's clearly no way to directly create class instances via the protocol. You therefore need to delegate the task of allocating and initialising new objects to a factory object or a free C function. Worse, protocols are orthogonal to the class hierarchy, so reference declarations will look different than those for other types:

```
id<MyProtocol> ref;
```

Instead of the expected

```
MyClass* ref;
```

Workable, but probably not ideal.

True abstract classes in Objective-C

So what about abstract classes? There's no direct, idiomatic support for them in the language, but even the very prominent `NSString` is abstract, and you wouldn't know it just by using it. It turns out

that documentation on the subject is scarce. One option is to leave out the method implementations in the abstract base altogether and live with the compiler warning about the incomplete class. At runtime, attempts to call unspecified methods will raise exceptions. More helpfully, but also much more laboriously, you can create dummy implementations which do nothing but raise an exception explaining the situation.

In most languages you need to know the concrete class when creating instances, or delegate this task to a factory. Interestingly, you can alloc and init `NSString` directly and receive subclass instances! As far as I know, the init methods return a different object than the `self` they are given, or they're internally doing funky things with the object's type. Alternatively, `NSString`'s alloc class method could be overridden to call its `NSCFString` counterpart, `NSString` thus acting as a factory for its own subclasses. If you do this yourself, you'll also need to define all the init* methods the concrete class uses on the abstract class, too, or they won't be visible to users of your class.

So far, this is definitely the cleanest solution for the header file and for users of the class, but it's also by far the most laborious, requiring an extra class, dummy/abstract methods and complicated init wrangling.

However, C++ programmers have found an elegant solution to similar problems. Exploding compile times due to seemingly infinite header dependencies are a serious issue in large C++ projects, and hiding class internals from library users is also often desired. As it happens, the solution to these can be applied to the Obective-C/C++ conundrum.

Pimpl

Short for "pointer to implementation" or "private implementation," this idiom is less unpleasant than its name might suggest at first. It's well-documented in the C++ literature. It's also quite simple. In the public header, add a forward declaration of an implementation `struct`, typically using the public class's name suffixed with "Impl" or some such convention. This `struct` will hold all the members we want to hide from the public class's header. Add a pointer to the struct as a class instance variable, and define the struct's members in the .cpp file (or in our case, the .mm), not the header. On construction (here: `-init*`), construct an instance of the struct using the `new` operator and set the instance variable to it, and ensure `delete` is called on destruction (here: in `-dealloc`).

In MyClass.h:

```
// ...
struct MyClassImpl;
@interface MyClass : NSObject {
  @private
  struct MyClassImpl* impl;
}
// public method declarations...
- (id)lookup:(int)num;
// ...
@end
```

In MyClass.mm:

```
#import "MyClass.h"
#include <map>
struct MyClassImpl {
  std::map<int, id> lookupTable;
};
```

```
@implementation MyClass
- (id)init
{
  self = [super init];
  if (self)
  {
    impl = new MyClassImpl;
  }
  return self;
}
- (void)dealloc
{
  delete impl;
}
- (id)lookup:(int)num
{
  std::map<int, id>::const_iterator found =
    impl->lookupTable.find(num);
  if (found == impl->lookupTable.end()) return nil;
  return found->second;
}
// ...
@end
```

This works because forward declarations of structs are valid C code, even if the struct later turns out to have implicit or explicit C++ constructors, or even base classes. The public class's methods access the contents of the struct via the pointer, possibly via member functions of the struct. Construction and destruction of members is handled by the `new` and `delete` operators as long as they are called correctly.

It's more or less up to you whether the functionality lives in member functions of the public class or the implementation class, or both. You might gain some efficiency by avoiding Objective-C messages in some cases, but it can get messy if the C++ methods need to call the Objective-C class's.

It should be noted that instead of single-member implementation structs, the implementation class can actually *derive* from the type of the member instead, thus avoiding the indirection on method calls. For example, in MyClass.h:

```
@interface MyClass : NSObject {
  struct MyClassMap* lookupTable;
}
- (id)lookup:(int)i;
@end
```

and MyClass.mm:

```
#import "MyClass.h"
#include <map>
struct MyClassMap : std::map<int, id> { };
@implementation MyClass
```

```
- (id)init {
  self = [super init];
  if (self)
    lookupTable = new MyClassMap;
  return self;
}
- (id)lookup:(int)i {
  MyClassMap::const_iterator found = lookupTable->find(i);
  return (found == lookupTable->end()) ? nil : found->second;
}
- (void)dealloc {
  delete lookupTable; lookupTable = NULL;
  [super dealloc];
}
@end
```

This becomes less practical with larger numbers of members due to the many new/delete pairs required. Locality of reference may also be worse.

Limitations

In pure C++, you may of course declare the implementation as a `class`, though this clearly won't work in our case where the forward declaration must be valid Objective-C. In the C++ literature, you may also find recommendations to use `shared_ptr<>` or `auto_ptr<>` to handle automatic deletion of the implementation object, or even to use a mix-in (templated base class) to provide the functionality. Neither will work with Objective-C headers. Even in runtimes that support correct construction/destruction of C++ members, the implementation member *must* be a pointer, as the struct type is incomplete in the header and reserving the correct amount of memory in the class will fail.

Because the implementation's definition is private, accessing members from a derived class, i.e. protected members, isn't directly possible. You can however move the definition of the implementation struct to a semi-private header, which only needs to be included by subclasses requiring direct access to it. Those subclasses must therefore be written in Objective-C++ themselves. Extending the implementation by subclassing it will be tricky as the pointer will still have the superclass type; you might prefer to create a second, separate struct instead.

Final thoughts

Nevertheless, I've found the Pimpl idiom to be the best choice for embedding C++ in Objective-C in almost all cases. Even when the struct only contains one member, the lack of casts easily makes up for the indirection. No efficiency is lost as long as the struct member isn't a pointer itself. For the reverse case of embedding Objective-C in a C++ class, use a similar approach. Define the public interface of the C++ class and a forward declaration for the implementation class in the header and place its definition with Objective-C member types in the corresponding .mm file.

A real-world example of embedding C++ can be found in my Objective-C wrapper for the Open-VCDiff decoder.

An Update (May 2012)

Since I first wrote this article in November 2010, Apple has switched to the clang compiler and made some changes to the Objective-C language. One of the new features, the class extension, allows you to declare instance variables outside the header file, opening up new possibilities for language

mixing. I have written a new article explaining the class extension technique - follow the link to read more on the subject.

Acknowledgements

Thanks to Björn Knafla for the valuable suggestions after reading drafts of this article and for the Twitter discussion which ultimately lead to writing this article in the first place.

Thanks also go to Markus Prinz for reading drafts of this article and making helpful suggestions.

References and Footnotes

[1] e.g. Chapter 7, Matthew Wilson, *Imperfect C++*; 2005 Addison-Wesley

[2] Note that in earlier versions of Objective-C++, lookupTable would have had to be a pointer and the map would need to be constructed explicitly with `new`, as the STL map has a non-trivial constructor. Clean C++ member construction and destruction in the Objective-C object lifecycle is a recent addition to Apple's implementation. I haven't tested it on other implementations of the runtime.

[3] You can forward declare `std::map` in theory but lose the ability to use the defaults for the third and fourth template parameters by doing so.

[4] I have so far only encountered this in connection with multiple inheritance. Still, class hierarchies can change, and memory layouts tend to be implementation-defined.

[5] In line with the rest of Objective-C, protocols also "looser" and more dynamic than interfaces in C#/Java, but that doesn't change anything in this case.