Doing OO in C

by Stephan Beal
Last Updated YYYYMMDD: 20101208

Abstract:

This document describes different approaches to object-oriented programming in C and develops a generic approach which has, in the author's experience, proven to be very useful, more type-safe than conventional methods, and strictly in compliance with the C language rules.

Maintainer: stephan@wanderinghorse.net (http://wanderinghorse.net/home/stephan/)

Table of Contents

1 Preliminaries

1.1 Disclaimers

The obligatory disclaimers:

- This document assumes that one has a solid working knowledge of C, and may also assume some C++ knowledge (but nothing too archaic). It is targeted as seasoned C programmers.
- My own knowledge is of course fallible, the implication being that this document may contain one or more outright lies. That is not the intention, of course, but it is possible.

1.2 License

> *"You cannot guaranty freedom of speech and enforce copyright law."*
>
> *Ian Clarke*
>
> *"This [document] is encrypted with ROT26 encoding. Decoding it is in violation of the Digital Millennium Copyright Act."*
>
> *Anonymous Software Developer*

This document, and all associated code, is released into the Public Domain. The author (Stephan Beal) explicitly disclaims all copyrights.

1.3 Credits

> *"It's a thankless job, but I've got a lot of Karma to burn off."*
>
> *Anonymous Software Developer*

The old saying goes, "everything I need to know, I learned in kindergarten," but that's not quite true in this case. The following people and sources have been very helpful in improving my command of the C language:

- My mother and father, Bonnie Pickartz and Joe Hudgins, without whom this paper would not exist.
- Simone Yvonne Müller, who literally saved my life and without whom i would not have been able to write any more software (or documentation, for that matter).
- W. Richard Stevens and Stephen A. Rago, authors of *Advanced Programming in the Unix Environment*, which is *the* book any C hacker (Unix or not) needs to have a copy of.
- Scott Meyers, author of several *excellent* C++ books, continually always influences my programming work.
- D. Richard Hipp, father of sqlite and Fossil, is responsible for my renewed interest in C after having given it up in 1995 in favour of higher-level languages. It turns out I can do almost as much with C as i can with C++ (but still requiring 5x as much code most of the time).
- My first and oldest programming mentor, Peter "SharpDog" Angerani, provided helpful feedback on early drafts of this paper.
- Long-time friend and former colleague Amir "Ashran" Mohammadkhani also provided helpful feedback on the first drafts.
- Herb Sutter's article *What's in a Class?* redefined how i think about (and use) object-oriented programming.
- Pete Harlow inadvertently provided the template for document (along with many others).
- Wikipedia contains a number of excellent articles with topics of interest to programmers, e.g. how hashtables work (or how hashing in general works) and the ins and outs of bitmasking.

1.4 Why?

Why write this document? Because...

i like to write technical documentation.

This particular topic has interested me since I started work on libwhefs (http://code.google.com/p/whefs/), an "embedded filesystem" library for C. i wanted the library to be able to use arbitrary back-end storage, so i need an interface which allowed me to do that. Out of that work (libwhio: http://fossil.wanderinghorse.net/repos/whio/), an OO model emerged which i have since come to use quite heavily, and i thought i'd share how it's done.

It is my hope that a random novice C programmer, out there in the real world, will be able to glean something of use from it.

2 Introduction

Most people, in my experience, do not tend to think of C as an *Object Oriented* (OO) language. Indeed, it is not object *oriented*, but it can indeed be used to program using OO idioms. This document is about how (in my opinion) to make safe and effective use of OO structures in C. We will cover some traditional approaches, point out some flaws in them, and proceed to improve upon them. This is not the end-all, be-all of OO idioms for C. It is, however, a model i have come to know, like, and use on a regular basis.

C is, in my opinion, only two features short of having a really workable object engine: constructors and destructors. We can get around a lack of constructors fairly easily, but a lack of destructors means we have to be very careful when it comes to error handling in conjunction with object cleanup.

Obviously, other features would also make OO programming in C simpler and safer, but in my humble opinion, it's really only the lack of destructors which i sorely miss.

Speaking of cryptic terms like "OO", before we continue let's make sure we're all using the same vocabulary...

2.1 What is an Interface?

An interface is a specification which describes how one should interact with something. A specification, in this sense, is simply "the documentation."[1] The documentation ideally describes what inputs are legal, their ranges, and what outputs to expect for the given inputs. It should also describe any special cases, corner-cases, and deficiencies or side-effects which might affect the client.

Even when not programming, we continually use interfaces every day without calling them interfaces. When we go into a building or get into a car, the door is (normally) the interface we use for doing so. When we use a toilet, the interface dictates that for proper use we must first lift the lid. And if we violate the interface, either by not opening the lid or by violating capacity restraints, then we make a big mess (the latter possibly resulting in a buffer overflow and a grumpy end-user).

Because documentation cannot cover every possible error case, we often see text like, "if the inputs are not valid, behaviour is undefined." This document often uses the term "undefined behaviour" to describe such a situation, and "undefined" means just that – the interface does not define what happens in that case. Most such references really mean something like "you are likely to cause a segmentation fault or memory corruption." In *all cases* we want to avoid instigating undefined behaviour.

2.2 What is a Client?

We keep referring to "the client." Who is he? The word has many meanings, but in the context of this document we use it to refer to whoever is using the code in question. That is, in effect, any user of a given piece of software is a *client* of that software. For our purposes, the word *client* only refers to *programmers* using our APIs in their software, and not to end users (who are blissfully unaware of what's going on in the software they are using).

2.3 What is a Class?

A class is, abstractly speaking, a data type. Most people do not use the term *class* for the most basic of data types, e.g. numbers and strings, but even such types atomic types fall into concrete classes (or categories).

For purposes of this document a class is a C structure. When generically referring to a class, the reference also implies any functions which are designed to be used with that class[2]. In higher-level languages, e.g. C++ or Java, classes normally have an explicit set of "member functions" which have access to private data within a class (private meaning that client code cannot access it). C does support members which are function pointers, but not true member functions. C also does not directly support the notion of public vs. private data, but idioms exist which allow us to implement private data to the extent that only clients who import the source code for the class can get at it.

For those of you C++ programmers: i once read about a guy who knew a guy who's step-sister's half-brother's uncle once reported seeing the following C++ code:

```
#define private public
#define protected public
#include <SomeThirdPartyHeader.hpp>
```

1 While most programmers despise documentation (as is implied by their lack of API docs), i'm a big fan of detailed technical documentation and tend to document individual functions and classes in pedantic detail.

2 This "extended definition" of a class is prompted by Herb Sutter's *What's in a Class?* article.

2.4 What is an Object?

An object is, in the abstract, any data. Some documentation refers to individual integers as objects, whereas higher-level documentation might refer to a very complex data structure as simply "an object." All such usages are correct.

Throughout this document we normally use the term object to refer to an instance of a class (see above). An instance is a portion of memory dedicated for use by/with one specific object.

2.5 What is a Subclass?

Subclass normally means a class which *inherits* behaviours from a different (a.k.a. *parent*) class. Our definition, for purposes of this document, is slightly more abstract. This document stresses the separation of interface and implementation. Classes play the role of defining the interface, and individual objects are responsible for adhering to their class' interface. For our purposes, a subclass is essentially the same as a class: a specification of an interface, possibly including one or more parent interfaces. We also sometimes use the term to refer to a specific concrete implementation of an abstract interface.

3 Brief Overview of Common OO Models in C

Here we provide a brief overview of OO models used in conventional C code. This is not a complete list – there are certainly many options which are not mentioned[3].

3.1 There Can Be Only One

Many, if not most, C classes are not designed to be subclassed. They are monolithic, often opaque, in structure, and you already know them well. Examples include the C-standard `FILE` handle type or even simple file descriptors (which are opaque handles representing a larger underlying object). The fact that `FILE` handles can sometimes be used in non-file contexts (e.g. `popen(3)`, defined by POSIX-1.2001), implies that it can be customized somewhat for specific device implementations, but such customizations are not part of the public `FILE` interface (as far as i'm aware, anyway).

3.2 Same Signature, Different Type

We sometimes see, particularly in older or lower-level code, a subclassing idiom in which the core API defines a structure and subclasses must have the same structure but *be* different types. Subclasses may "extend" the parent class by adding all of the same members and optionally adding new members *after* those members specified by the base class.

For example, consider this base class:

```
struct BaseClass {
   int field1;
   double field2:
   char * string;
   unsigned int stringLength;
   int (*func1)( int );
   double (*func2)( double, int );
};
```

(We won't document this interface, since the details of the interface are moot. Here we are interested only in the structure.)

A function which uses such a structure might look like:

```
int do_something( struct BaseClass * obj, … args … );
```

To create a concrete implementation of such an interface (i.e. a class), implementors are required to create a structure with the same initial members, and to add their own to the end. For example:

```
struct MyClass {
```

3 Tell me about them and i'll be happy to add them here.

```
    int field1;
    double field2;
    … continue as above, then add custom fields …
    long customField1;
    long customField2;
    …
};
```

On the surface, that seems only slightly insane (and highly unmaintainable) but not technically problematic.

Now comes the scary part: we have to pass an instance of our object to a function expecting a different signature:

```
struct MyClass my = …;
int x = do_something( (struct BaseClass*)&my, … );
```

Yes, there are actually C APIs which do this, but the only ones i have seen have been around since before ANSI standardized C. I prefer to believe that no self-respecting coder would actually do that nowadays.

Do not use this model in your own code. It is not only difficult to extend, but has undefined results. C does not require that the pointer address will be the same for both types when a cast is done this way. It will in fact work on most, if not all, modern C compilers, but relying on this means relying on undefined behaviour.

Given its unsightliness, we won't say any more about this approach.

3.3 The Conventional Approach

A more conventional, probably the most common, approach works like the first approach except that each instance uses the exact same base type and populates its members in accordance with the interface documentation. For example, to implement a specific class of the interface shown in the first example we would do something like this:

```
struct BaseClass my;
my.field1 = 7;
my.func1 = some_function;
…
```

There is nothing fundamentally wrong with this approach. It conforms to the C specification and is straightforward to implement. It can, however, be tedious to extend.

4 My Preferred Approach (it has no name)

The approach to subclassing described throughout the rest of this document is discussed in much more detail than other approaches because:

- This is the approach i use on a daily basis.
- i have found it to be flexible and relatively type-safe.
- Its unified object finalization API helps me keep me keep my libraries leak-free, and in turn helps clients keep their applications leak-free. Since i am my own biggest client, i especially appreciate this feature.

It is essentially the same as the more conventional approach described above, but it adds one more layer of indirection for reasons explained in detail below.

4.1 Basic class layout

Before describing this approach, let's introduce a handful of structures and data types which we will refer to as we develop the model:

```
// First, the core interface:
struct MyInterface;

// Interface-specific functions:
typedef int (*MyInterface_func1_f)( struct MyInterface *, int );
typedef double (*MyInterface_func2_f)( struct MyInterface *,
```

```
                                        double );

// Cleanup/finalization functions:
typedef int (*MyInterface_cleanup_f)( struct MyInterface * );
typedef int (*MyInterface_finalize_f)( struct MyInterface * );


// Holder class for all public member functions:
typedef struct MyAPI {
      MyInterface_func1_f func1;
      MyInterface_func2_f func2;
      MyInterface_cleanup_f cleanup;
      MyInterface_finalize_f finalize;
} MyAPI;

// Secondly, storage for implementation-specific details:
typedef struct pimpl {
      void * data;
      void const * typeID;
};

// Lastly, our interface class. This class is the one clients
// will see in the signatures of functions in the public API.
typedef struct MyInterface {
      const MyAPI * api; // conceptually similar to a "vtbl" pointer
      pimpl impl;
};
```

(For those who do not know this: "pimpl" is the conventional abbreviation for "private implementation" data.)

With that in place, we have everything we need in order to start creating various concrete implementations (i.e. (sub)classes) of `MyInterface` which conform to the `MyAPI` interface[4].

Compared to the conventional approach of having the member function pointers in the main class itself, the immediate implications of the overall model shown above include:

1. We have a slightly more verbose calling syntax (e.g. `obj->api->func1(obj,3)` instead of `obj->func1(obj,3)`) and one additional pointer dereference per call. But..
2. We save memory on *every instance* of this class. All concrete instances of a class share (by definition) the same set of implementation functions, and therefore we just point to the same (static/shared/const) object. For an API with N member function pointers, the conventional approach requires `(((N * sizeof(functionPointer)) - sizeof(MyAPI*))`[5].more bytes of memory *per instance*.
3. It might not be obvious, but this separation implies that we *must* document the interface (apparently contrary to conventions), and do so generically, rather than in terms of a single implementation. One point of this separation is to allow (indeed, encourage) multiple concrete implementations of a given interface (i.e. creating classes and subclasses). Doing so requires that the implementations have a common reference which explains how they should behave, and normally that role is played by the API documentation. If an interface is not documented, it cannot be effectively implemented by anyone other than the interface's creator. And six months after writing it, even the original author may no longer remember what the interface is supposed to do.

We discuss the functions named `cleanup()` and `finalize()` in detail in sections 4.2 and 4.7. For now, just be aware that `cleanup()` is responsible for freeing internally-used memory and `finalize()` is used for freeing the object itself. We will see later why this distinction is significant.

4.2 Semantics of the Clean-up API

Our interface includes two functions for cleaning up. Why two? Because that's the only way the interface can generically clean up objects independently of their allocation source. That is, without having to know whether the object is stack-allocated, `malloc()`-allocated,or custom-allocated. We'll try to convince you...

The semantics of these functions are:

4 We still don't know what this interface is actually for, but that's not important for this discussion.
5 Remember that C does not guaranty that `sizeof(void*)` and `sizeof(functionPointer)` have the same value.

`cleanup(X)` must free any resources associated with the object passed to it, and do any implementation-specific clean-up, but must not free the `X` the object.

`finalize(X)` must call `X->api-cleanup(X)` and then deallocate `X` using the method complementary to how `X` was allocated.

This generic approach allows the objects in question to be de/allocated using various methods without affecting most of the API. A type's factory function(s) and its `finalize()` implementation must use complementary de/allocation methods, but other code is not generally concerned by the differences in objects' origins. Sometimes it is useful if `cleanup()` also knows about this distinction, as we will see in section 4.7.

4.3 Empty Initializers

Before we proceed to demonstrate how this model is implemented, let's get some work out of the way now which will later save us hassle and grief: provide initializers for all of our basic types. The following "bootstrap" code may initially seem tedious, but has long-term benefits which have proven themselves to me time and time again:

```
// Empty-initialized MyAPI object:
#define MyAPI_empty_m = { \
      NULL /* func1() */, \
      NULL /* func2() */, \
      NULL /* cleanup() */, \
      NULL /* finalize() */ \
}

// Empty-initialized MyAPI object:
extern const MyAPI MyAPI_empty;

/* Note to the curious:
   The "_m" suffix convention denotes "macro", to distinguish
   from the like-named _objects_ named without the "_m" suffix.
*/
// Empty-initialized pimpl object:
#define pimpl_empty_m = { \
      NULL /* data */, \
      NULL /* typeID */ \
}
extern const pimpl pimpl_empty;

// Empty-initialized MyInterface object:
#define MyInterface_empty_m = { \
      NULL /* api */, \
      pimpl_empty_m /* impl */ \
}
extern const MyInterface MyInterface_empty;
```

We use such initialization objects in the following contexts:

1. When using newly-allocated objects, copying the "empty" objects provides a safe, intuitive way to ensure that all members of the object are initialized to a known state. (Again, possibly with some members set to non-0 values.)
2. The macros are used to uniformly initialize inlined members of *other* classes, potentially with non-0 default values.
3. We can provide multiple variants of the "empty" objects which have different default values. We internally use this when setting up concrete implementations (demonstrated later).

The various `extern const` objects serve the same purpose as the macros, but are (in my opinion) cleaner if only because they have full debugging symbol information. However, they cannot be used to inline-initialize members of other classes because C does not allow objects to be used that way. For example:

```
struct {
    MyAPI api;
} Foo[2] = {
```

```
      MyAPI_empty_m, // legal
      MyAPI_empty    // not legal
};
```

The reason for this, my old mentor Pete reminds us, revolves around the question of whether to directly *use* the const data directly or to *copy* it. That's not answerable for the general case.

4.4 Setting up the Implementation

Before we can create objects, we need to get the lower-level parts out of the way. That includes declaring the functions of the API and setting up a shared `MyAPI` instance, along with some internal types and instances we'll be using.

Given our interface and initialization macros, described in the previous sections, the setup looks like this...

First, let's get our empty-initialize objects out of the way:

```
const MyAPI MyAPI_empty = MyAPI_empty_m;
const pimpl pimpl_empty = pimpl_empty_m;
const MyInterface MyInterface_empty = MyInterface_empty_m;
```

Now we need to declare the implementations for all of the functions of from `MyAPI` which we need to implement for our subclass[6]:

```
// Implements MyInterface_func1()
static int MyImpl_func1( MyInterface *, int );

// Implements MyInterface_func2()
static double MyImpl_func2( MyInterface *, double );

// Implements MyInterface_cleanup()
static int MyImpl_cleanup( MyInterface * );

// Implements MyInterface_finalize()
static int MyImpl_finalize( MyInterface * );
```

We will demonstrate the last two functions in section 4.7, because their general approach is largely independent of the underlying type, but the other implementations necessarily depend on the `MyAPI` interface being documented, which it is not.

After those declarations are in place (they need not be static, by the way), we need to set up our implementation-specific types:

```
// A shared MyAPI instance for MyImpl objects:
static const MyAPI MyImpl_api = {
      MyImpl_func1,
      MyImpl_func2,
      MyImpl_cleanup,
      MyImpl_finalize
};

// pimpl instance for MyImpl objects:
#define MyImpl_pimpl_empty_m { \
      NULL /* pimpl::data */, \
      (void const *)&MyImpl_api /* pimpl::typeID */ \
}
static const pimpl MyImpl_pimpl_empty = MyImpl_pimpl_empty_m;

// Prototype object for our concrete MyInterface implementation:
#define MyInterface_MyImpl_empty_m { \
      &MyImpl_api /* MyInterface::api */, \
      MyImpl_pimpl_empty_m /* MyInterface::impl */ \
```

6 In some cases we can share implementations across several classes. If a given function is fully independent of the implementation details, and can be implemented solely in terms of the class' public interface, then it should be made a free functions instead of a member. Doing so saves one pointer per class and spares implementors of concrete classes the work of having to initialize that extra member.

```
}

// Implementation-specific internal/private data:
typedef struct MyImpl {
      int x;
      double y;
      … any impl-dependent data we need …
      MyInterface iface; // not required, but see notes below
} MyImpl;

static const MyImpl MyImpl_empty = { 0 /*x*/, 0.0/*y*/,
      … other args … ,
      MyInterface_MyImpl_empty_m /* iface */
};
```

(We finally get a glimpse of the `typeID` field. We will see more of it later on.)

A note about the `iface` member of the `MyImpl` class: having this member allows us to avoid allocating *both* a `MyInterface` *and* a `MyImpl` object in our factory functions. Instead we allocate only a `MyImpl` object, and return its `iface` member to the caller of the factory function. It adds a bit of complexity to the configuration but saves us one call to `malloc()` and the corresponding call to `free()`. It also incidentally gives us a way to implement `MyImpl_finalize()` to behave safely if passed a pointer which was allocated by the client (e.g. stack-allocated). With that hack in place (we'll see it in sections 4.6.1 and 4.7), the distinction between `cleanup()` and `finalize()` basically goes away, and the client can always use `finalize()`. That requires, however, "the hack", and it may not be appropriate for all classes.

Now, finally, with all of that in place we can continue on to the topic of creating concrete instances of the interface class...

4.5 `pimpl::typeID` and `MYIMPL_DECL()`

The `pimpl` interface shown above contains a field called `typeID`. So far we haven't see it much, but we will soon, so we need to understand what it is for. It is used to mark objects so that we can confirm their type later on. With it, we can ensure that our `MyImpl` functions (which take `MyInterface` pointers) are passed `MyImpl` objects instead of other `MyInterface` pointers. We do this "tagging" when we allocate the object, and implementation functions can check the tag using private functions or macros like the one shown below.

In later examples we will use the following macros[7] to simplify our checking of the `typeID` field:

```
// Evalutes to true if IfaceP is-a MyImpl implementation.
// IfaceP must be-a (MyInterface*).
#define ISA_MYIMPL(IfaceP) ((IfaceP) && \
      ((IfaceP)->impl.typeID == (void const *)&MyImpl_api) )

// Declares local (MyImpl * my) and casts it from
// IfaceP->impl.data. If IfaceP is-not-a MyImpl then
// it calls return(ReturnVal). For void returns, leave ReturnVal
// empty (some compilers cannot explicitly return void).
#define MYIMPL_DECL(IfaceP,ReturnVal) \
      MyImpl *my = ISA_MYIMPL(IfaceP) \
          ? (MyImpl*)(IfaceP)->impl.data \
          : NULL; \
          if( ! my ) return ReturnVal
```

We will use the `MYIMPL_DECL()` throughout our `MyImpl` code, as we will need this capability in any functions which are part of the `MyAPI` interface.

Rather than have a dedicated `typeID` field, we could instead compare (`m->api == &MyImpl_api`) and that would serve the same end (confirming that the incoming object is of the proper concrete class). That would allow us to get rid of the `typeID` field altogether. We don't do that in this demonstration because i actually have some client code which breaks that approach[8]. The `typeID` field is reserved for this purpose,

7 These can typically be defined in the private implementation, and need not bleed into the public API.
8 A special-case subclass swaps the `api` member pointer with its own implementation in order to replace two functions. In that case, all other `api` member functions are retained and work as expected, but the pointer value of the instance's `api` member is then different, meaning that checking its address would cause the other functions to unduly fail.

so we'll use it.

The `typeID` value can be *anything*. In practice it points to one of the various internal private objects. Occasionally it makes sense to put the `typeID` value for a given concrete implementation in the public API, as this allows clients to write functions which use the generic interface but require that the interface pointer passed to them be of a specific concrete type (or types). Alternately, some algorithms might work generically but provide optimizations specific to certain concrete implementations.

4.6 Factories (a.k.a. Constructors)

Construction of new objects is necessarily type-specific, and must take into consideration at least the following points:

1. Allocate the instance in the factory function, or require the client to allocate it and pass it in?
2. What arguments are needed for initialization?
3. How must the client *destruct* the instance in order to ensure that all resources are recovered.

The answer to point 2 is necessarily class-specific. We might need multiple factories for different arguments, or perhaps even a variadic factory.

Point 1 is especially interesting, and we will demonstrate that an API can easily support both approaches while also providing a uniform clean-up API (which is the topic of point #3).

Initialization of new objects at least partially depends on whether the object was allocated on the stack or on the heap (i.e. dynamically). Initialization routines (or factory functions) often take one or more of the following forms:

```
// Return new object as result of initialization:
MyInterface * MyImpl_init( … );

// Alternative form of previous example, assign result to *target:
int MyImpl_init( MyInterface ** target, … );

// Take an existing object and initialize it:
int MyImpl_init( MyInterface * obj, … );
```

(Note that we are assuming the conventional result type of `int`, with 0==success, but the return type is not important for purposes of this model. Use whatever return type you prefer.)

A less common, but more flexible approach, looks much like the second example from above, but has different semantics for the first argument:

```
// Accept a pointer to NULL or a pointer to an existing object:
int MyImpl_init( MyInterface ** my, … );
```

It can be used like:

```
MyInterface * my = NULL; // DO NOT use a default-initialized ptr!
int rc = MyImpl_init( &my, … );
```

or:

```
MyInterface M = MyInterface_empty;
MyInterface * my = &M;
int rc = MyImpl_init( &my, … );
```

In the first case, the function allocates a new object and assigns `*my` to it, returning ownership of the object to the caller. In the second case the function assumes that the caller allocated the object (but doesn't care how), and proceeds to use it for the initialization. The exact semantics are described, and the initialization process is demonstrated, in section 4.6.1.

There is a particularly vexing question in all of this, though: if initialization fails, how can the routine generically ensure that the object is properly cleaned up before it returns?

The answers lie in two of the functions we defined in the `MyAPI` interface: `cleanup()` and . `finalize()`. Which does what, and the specifics of each, are defined in section 4.7, but we're not quite yet finished with discussing constructor/factor functions.

4.6.1 Implementing a Constructor/Factory

Here we demonstrate how the above-mentioned approach to construction can be implemented. In the abstract, the constructor/factory function signature looks like this:

```
int MyImpl_factory( MyInterface ** obj, … type-specific args … );
```

The primary advantage of returning an integer (or some specific error type), and passing back the "return object" via an output parameter, is that we can return richer error codes to the caller. If we return the object as a pointer then we must return `NULL` on error but cannot tell the caller *why* it is `NULL` (File not found? Out of memory? An argument is out of range? Printer jammed?).

There is, however, a more subtle advantage: we have more choice about how the object is allocated. Here's how...

The semantics of the first parameter are generic enough to concretely define here, regardless of the first argument's type:

> The argument may not be `NULL`. If it *points* to `NULL` (i.e. `(NULL==*obj)`) then this function allocates a new object, otherwise it assumes the caller allocated the object (possibly on the stack), that it is empty-initialized, and proceeds to use it.

But that's not all. We must also define how it behaves if the initialization fails. What if, after initializing several sub-parts of the object, it fails? It must clean up the private parts, free the object *only* if it allocated it, and return the error code to the caller.

The specifics of the cleanup process are described in section 4.7, but overall the on-error rules are as generic as the on-success rules, regardless of the type of the first argument:

> If initialization fails then the object is cleaned up as follows: if the caller passed a pointer to `NULL` (meaning this routine allocated the object), then `newObj->api->finalize(newObj)` is called and the `obj` *parameter* is not modified. If the caller passed in his own object (which could have an arbitrary memory source), this routine calls `(*obj)->api->cleanup(*obj)` and the caller must free its memory using whatever means compliment its allocation (e.g. do nothing for stack-allocated objects and call `free()` for `malloc()`'ed memory).

The factory's on-success results can be at least partially generically described, but some details may be specific to the class:

> Ownership of `*obj` is given to the caller, who must eventually free it using one of these two methods:
>
> 1) Iff[9] the object was allocated by this function, call `obj->api->finalize(obj)`.
>
> 2) Iff the object was allocated by the client, call `obj->api->cleanup(obj)` to free the internal resources and then free `obj`'s memory using the method which compliments its allocation method.

How's that for service: i've just written half of the API docs for you :).

One side-effect of the genericness of this approach is that it can be used in conjunction with custom allocators, for those [of us] who want to avoid `malloc()` in certain cases.

The implementation of such a constructor/factory might look like the following, pretty much regardless of the concrete implementation type:

```
int my_factory( MyInterface ** tgt, … whatever … )
{
      if( ! tgt ) { return InvalidArgError[10]; }
      int rc = 0;[11]
      const bool ownsObj = *tgt ? false : true;
      MyImpl * impl = (MyImpl*) malloc(sizeof(MyImpl));
      if( ! impl ) return AllocationError;
      *impl = MyImpl_empty /* sets up all API members and typeID */;
      MyInterface * iface = ownsObj
                  ? &impl->iface
                  : *tgt;
```

9 No, that's not a typo: "Iff" is geek-speek for "if and *only* if"

10 Assume that `InvalidArgError` and other named error codes are defined via an enum.

11 Note that we're simplifying here by using C99 syntax. Feel free to restructure it for C89 compatibility if needed. There is nothing in this model which inherently requires C99-only constructs.

```
      iface->impl.data = impl;

      … set up the impl object … set rc to non-0 on error …

      if( 0 == rc )
      { /* success... */
          // Pass ownership to caller:
          if( ownsObject ) *tgt = iface;
          // else *tgt already is iface
      }
      else
      { /* failure... */
          // Clean iface + all resources:
          if( ownsObject ) iface->api->finalize(iface);
          // Clean resources, but caller must free iface:
          else iface->api->cleanup(iface);
      }
      return rc;
}
```

The above code can be used as-is for any types using that clean-up interface, rather than having to perform custom cleanup in every factory function.

Note that if the client passed in his own object, we clean it up to avoid leaving it in an undefined state (since initialization failed, the state is presumably not usable), but we do not free `*tgt` itself. This feature allows the client to allocate `*tgt` using whatever means he likes. The caveat is that we cannot legally call `finalize()` on an object which is allocated using an arbitrary mechanism[12], and the creator of the object must be sure to use `cleanup()`, instead of `finalize()`, and then deallocate it appropriately. i say "caveat", but Indeed, the distinction between stack and non-stack allocation *is* the reason the cleanup interface requires two functions instead of one.

In fact, the question is not actually "did the memory come from the stack or the heap?" The question is actually (from the destructor's point of view) "was the object i have been asked to deallocate actually allocated using the allocation method i expect/require [from, e.g., the factory functions]?" It is possible, sometimes useful, to have class-specific custom allocators. As long as the factory functions and the destructors (see the next section) cooperate, they only need to be able to answer the question "did this allocation come from my API?" If the answer is no, then the object is, from the perspective of the destructor, "stack allocated" (i.e, "clean it up, but do not deallocate it because it belongs to someone else").

4.7 Destructors

One of the beautiful things about C++ is destructors. They provide us with a common interface for performing object clean-up once the object is no longer needed. Objects get cleaned up in a uniform manner, the only difference being whether the object was stack- or dynamically-allocated. The former requires no programmer effort (the object simply goes out of scope) and the latter requires an explicit call to destroy the object and release its memory.

In C we do not have destructors but we must nonetheless ensure that our objects leak no resources. C programmers have been doing it since forever, and that's nothing new.

If an object is created on the stack, it must not be explicitly freed, but we may still need to free up any dynamically-allocated resources associated with the object. If an object is created dynamically we must first free its associated data and then free the object in a manner complementary to its allocation.

It turns out that we can unify the cleanup and destruction steps into a simple, generic interface which, in my experience, can be used as-is for near-arbitrary client types. (The code below is essentially identical to code in many of my own C classes.)

The overall destruction API is described in section 4.2. What we need to do now is create custom implementations of those interfaces which know how to clean up `MyImpl` objects:

```
// Implements MyInterface_cleanup_f()
int MyImpl_cleanup( MyInterface * self )
{
```

12 Actuallly... `clean()`/`finalize()` *can* be (and have been) implemented, with only a slight amount of inconvenience, such that `finalize()` will behave the same whether its factory function or the client allocated the interface object.

```
      MY_DECL(self,ArgumentError); // See section 4.5

      … free any resources in 'my', but not (yet) 'my' itself …

      if( self != &my->iface )
      { /* this means the client allocated self.
           Clean up our parts here. */
          self->impl = my->iface.impl = pimpl_empty;
          free(my);
      }
      /* Else self lives inside m, and we must instead
         be finalize()'d. If the client follows the interface,
         he called finalize() instead of cleanup().
      */
      return 0;
}

/** Implements MyInterface_finalize_f()
  It is safe to call this particular implementation if
  the client allocated his own MyInterface object
  for use with the factory routine(s). In that case this
  will behave like MyImpl_cleanup() but will not destroy
  the self object. We presume the caller knows that he must
  deallocate it (since he custom-allocated it).
*/
int MyImpl_finalize( MyInterface * self )
{
      MY_DECL(self,-1); // See section 4.5
      MyInterface const * old = &my->iface;
      // The finalize() interface requires that we cleanup():
      self->api->cleanup(self);
      if( old == self )
      { /* means we allocated self in our factory. */
          free(my); // remember: self is my->iface!
      }
      else
      { /* if the client is following the interface, he
           called cleanup() instead of finalize(), and this
         case is handled there. */
      }
      return 0;
}
```

4.8 Using custom de/allocators

It is sometimes useful to use custom memory allocators. To do so, we must simply replace the calls to `malloc()` and `free()` in the constructor and destructor. There are a number of things one can do with these, e.g. allocating objects from a client-specified memory area, or use a pager to pre-allocate and recycle objects, but such topics are out of scope here.

4.9 Adding our Implementations

Finally, at long last, we can get around to adding our implementation functions. In fact, we're not going to do that here because we've already implemented the two `MyInterface` functions which have any documentation. We will, however, show the generic pattern for using this approach:

```
// Would implement MyInterface_func1(), if we knew what that
// interface was supposed to do.
int MyImpl_func1( MyInterface * self, int x )
{
      MY_DECL(self,0); // See section 4.5
```

```
      … do anything useful here …

      return my->x + x;
}
```

A small, possibly not obvious, trick is needed for our `MY_DECL()` macro in conjunction with functions returning void:

```
void MyImpl_func42( MyInterface * self, va_list vargs )
{
      MY_DECL(self,);
      // or: MY_DECL(self,(void)0)
      // MY_DECL(self,void) is not legal in all compilers.
      …
}
```

Calling `MY_DECL(self,void)` would work on some compilers, but some do not allow an explicit return of `void`.

4.10 Use the Implementation in Client Code

The moment it all boils down to is when we can finally use it all in client code:

```
MyInterface * m = NULL;
int rc = MyImpl_factory( &m, … );
if( 0 != rc ) { ... error ... }
else {
      … use m (if we only knew what it was supposed to do!) …
      // When we're done:
      m->api->finalize(m);
}
```

Alternately:

```
MyInterface M = MyInterface_empty;
MyInterface * m = &M;
int rc = MyImpl_factory( &m, … );
if( 0 != rc ) { … error … }
else {
      … use m … if we only knew what it was supposed to do! …
      // When we're done:
      m->api->cleanup(m); // NOT finalize()!!!
}
```

Note that in this second case, if the factory fails then we *must not* call `cleanup()` on the object because it does not have the proper `MyAPI` functions installed in it. The *factory* is responsible for cleaning up the object if initialization/setup fails.

In the `finalize()` implementation we provided, which includes "the `finalize()` hack", we could safely call `finalize()` in either of the above cases and the proper behaviours would be invoked. However, since that implementation is not necessarily suitable for all `MyInterface` types, we follow the generic interface here rather than relying on `MyImpl`-specific behaviours. That allows us to swap out the factory-function call with a different factory and still get proper behaviour vis-a-vis cleanup (assuming the new implementation complies with the interface).

4.11 Real-world Use

The following is code which comes from a real-world API which uses the object model described in this documentation. It is an i/o device interface which supports random access to any back-end for which we can write an implementation. For example, we have implementations which can use `FILE` handles, file descriptors, or memory as storage. Usage is the same for all compliant devices, and they can (in most cases) be freely swapped out. The only difference is how they are constructed (different devices need different arguments for construction).

Here are some example:

```
// File-based devices:
whio_dev * f = whio_dev_for_filename( "my.file", "r+" );
whio_dev * myStdout = whio_dev_for_FILE( stdout, false );

// Dynamically-growing in-memory device:
whio_dev * m = whio_dev_for_membuf( 1024 * 5, 1.5 );

// User-defined memory space:
static unsigned char buf[1024*10};
whio_dev * rw = whio_dev_for_memmap_rw( buf, sizeof(buf) );
whio_dev * ro = whio_dev_for_memmap_ro( buf, sizeof(buf) );

// "Subdevices" use a chunk from a parent device:
whio_dev * sub = whio_dev_subdev_create( rw, 100, 200 );
```

The point is that the factory functions are all different, but performing i/o is the same for all types:

```
f->api->seek( f, 5, SEEK_SET );
rw->api->write( rw, "Hi, world!", 10 );
ro->api->write( ro, "Will fail: this is a read-only device.", 38 );
// We can of course also use device-independent algorithms:
whio_dev_writef( myStdout, "Hi, %s!\n", "world" );
whio_dev_writeat( sub, 20, "Hi, world!", 10 );
```

And, almost as importantly, destruction of the objects is unified:

```
f->api->finalize( f );
sub->api->finalize( sub );
rw->api->finalize( rw );
whio_dev_finalize( ro ); // alternate syntax
...
```

As one can see, the unified cleanup API simplifies destruction of the devices a great deal over having to know about implementation-specific cleanup routines.

5 In Summary

("In summation" always sounds like one word to me.)

The OO model described in this document has seen heavy use in my C code since sometime in late 2008, and i've come to use it whenever my C data types specifically need a hard split in the interface vs. the implementations. A good example of it is libwhio (see link in section 1.4), where we have no less than *six* different interface-conformant i/o device implementations (one of which lives another tree which is itself a client of libwhio).

Other than coming to trust and rely on this model, there's not much more for me to say which has not already been written elsewhere.

So... the demonstrations above should be enough to give one an idea of how to implement this model, but don't take my word for it. Try it out in your own code and see if you like it.