Context Singletons - context-specific shared objects in C++

stephan beal <stephan@s11n.net>

14th June 2004

Abstract

Documentation CVS Revision: $Revision: 1.1 $

This document develops an approach to what i call "context singletons." They are not Singletons in the true sense, but are Singletons within a given Context. Perhaps "shared objects" would be a more appropriate name, but that is too ambiguous to suit me :/.

i document this mainly because i have never seen it used before (outside of my code, obviously), yet i often find it quite useful myself, and i believe that other coders may find some utility in it.

A note to readers of the HTML version of this document: it may contain character translation errors introduced by the LyX –> HTML conversion!

Contents

1 Introduction

It is often the case that programmers want a shared object which they can easily access from "all over the source tree", and which provides a set of shared resources for a project. Classic examples of this include Configuration

objects, Classloaders and Resource Pools. More often that not, these classes are designed to be what are commonly known as Singletons - it is impossible (and normally undesireable) to create more than one instance of these objects. Indeed, that suits the above requirements very well. In my experience, however, this approach is rather limiting: the class can be used for no other purpose except as a Singleton. To be fair, sometimes this is the only proper approach for a given object. Sometimes, however, it is not - it is feasible that class MySingleton might actually be useful in a non-singleton context, but the practice of designing it *to be a singleton* keeps it from being used in any way except *as a singleton.*

This paper discusses another approach to Singletons - those which are designed to be "just another object", but which may be used in a Singleton-like manner. i call them, somewhat inaccurately, Context Singletons - objects which are shared within a given context. Contexts are defined, probably quite unintuitively, as data *types.* To be clear, they are not true Singletons, yet they can be conveniently thought of as Context-specific Singletons.

1.1 References

This paper often makes references to:

- Andrei Alexandrescu's *Modern C++ Design*: ***the*** C++ book to read if you think C++ is going out of style. Read it five or six times, even. Some of the lower-level phoenixing details covered there are not repeated here.
- Scott Meyers' C++ series: *Effective STL*, *Effective C++*, and *More Effective C++*: essential reading for practical-minded programmers. (Scott, here's you're biggest fan, right here!)
- My own `class_loader`: a template-based classloader framework from which this paper evolved: `http://s11n.net/class_loader/`

1.2 Credits

Many thanks to:

- Peter "*What's Happ'nin!?!?!?*" Angerani, for proof-reading this paper.
- And to Ashran, for at least reading it ;).

2 Example case: application configuration data

Let's take the classic example of an application's Configuration data (we'll call it Config, for short). Normally this type of object acts as an interface for getting/setting arbitrary key/value pairs from/to a configuration file (be it a system-wide registry or `~/.myconfig`). This pattern is often implemented by writing the Config object as a singleton, providing an instance of it via a static method, or by making all of it's method's static. Thus the client of a framework might use the Config object like so:

```
Size windowsize = myframework::Config::get_size( "default_app_size" );
```

or:

```
Size windowsize = myframework::Config::instance().get_size( "default_app_size" );
```

Those are both reasonable approaches, and work well for many cases. However, there are inherent problems with these approaches, though they may not be immediately obvious and may never show up in the useful lifetime of a given class: Config is now hard-coded with the following limitations:

- Only one of them may exist at a time. (Indeed, this is the whole point of a Singleton.)
- The source code for Config cannot be used in two projects that might eventually link against each other some day, if those projects each expect to have their own Config object.

This second point probably seems a bit far-fetched, so i will elaborate on it to try to show that it is indeed a realistic case. To do so i will present such a case which i have personally experienced[1], followed by a proposal for how to painlessly solve it for cases where using a *true* Singleton is not strictly necessary.

[1] Admittedly embelished, for clarity.

2.1 The inherent non-reusability of classic Singletons

In many cases an object would be very useful in several contexts, but is designed to work only as a Singleton, and therefore cannot be re-used. Additionally, it is sometimes not practical for two source trees which share the same Singleton source code to be linked together, as the source trees may be using the Singleton for different purposes.

As an example, i have been working on a home-grown shell-style-application framework for a couple of years (called eshell). One of the things the framework provides is a Config object, so each application linked to it automatically gets persistent config settings. The Config is, naturally, used like a Singleton.

At one point i wanted to merge eshell into a larger project i've been working on for some years, QUB (`http://qub.sourceforge.net`), to expand the features of some of QUB command-line client apps. The catch is that QUB uses, not surprisingly, some of the code initially created for eshell. In the case of the Config object, this poses a problem: QUB and eshell both use Config for different purposes, and therefor i do not want them sharing the same Config object. For example, i want to completely avoid the possibility that both sides try to use the same config variables, because they might use them for completely different purposes, and are thus not *semantically* compatible.

Several solutions/workarounds come to mind:

1. Write another Config object.
2. Put a copy of Config into a different namespace, making it a *completely different type*.
3. Make the Config object a *Context Singleton*.

We will immediately discard option 1, primarily because i'm lazy and see no need to do the extra work.

Option 2 has the disadvantage of extra code maintenance (even if that can all be handled by tools like `sed`).

Options 2 and 3 are remarkably similar, in the sense that Context Singletons rely on different Context *types* to distinguish between them. We can, however, accomplish the different-types approach without any namespace (or similar) manipulation. This sounds like the most re-usable and interesting approach, so let's now investigate option 3...

2.2 Replacing Singletons with Context Singletons

The first thing we need to do to accomplish our goal is to make the Config object a non-singleton. That is, we must make sure it behaves properly, using local config data, for each instance of the Config object. Some cases, of course, require *true* Singleton behaviour, and they are not covered here.

In the above case this was a no-brainer because the Config object was not a true singleton to begin with: it was a normal object with a static `instance()` method which provided access to a shared instance of Config. In many cases objects are written to *only* work as Singletons, and those classes will need more tweaking before applying this model to them.

Let's briefly look at a straightforward approach to implementing `Config::instance()`, the method responsible for providing an application access to the shared Config object:

```
Config & instance() {

    static Config meyers;
    return meyers;

}
```

Let's ignore, for the time being, the inherent post-`main()` fragility of this code - that's another topic altogether, to be discussed later.

Now, going back to the original goal of providing two Config objects via Singleton-like access methods, we'll jump right into the Context solution proposed above. As you will see, the answer is so straightforward that the source code can explain it better than any textual description. We simply add one line to the above code:

```
template <typename Context>
Config & instance() {
```

```
    static Config meyers;
    return meyers;
}
```

In case it's not immediately obvious, let's go over the implications of this:

By making `instance()` a method template, we now have not *one* `instance()` method, but (as Alexandrescu puts it) *a family of methods*, each with their own copy of the static object, `meyers`. That is, the following two calls will return *completely different* Config instances:

```
Config & c1 = Config::instance<Foo>();
Config & c2 = Config::instance<Bar>();
```

Why do we use contexts of `Foo` and `Bar`? No reason in particular - the important thing is that we have a Context type, and any `instance()` calls using the same Context types will return the same Config objects. It is significant to note that the Context type is never used to create an object: it is used *only* for the purposes of converting `instance()` from a single method into a family of methods. Thus, using arbitrarily large types as Contexts causes no inherent performance problem.

2.3 Sharing Contexts

As Context types are not "really" used in the above code, *any* class may be used as a Context class. Sometimes it is useful to create explicit context classes, and these can be implemented in the same way as the STL's iterator marker types:

```
struct eshell_sharing_context {};
struct qub_sharing_context {};
```

Keep in mind that `typedefs` do not create new types, so they are not useful for this purpose.

In cases where a whole class is made Context-sensitive, as opposed to one method of the class, `typedefs` are useful for easing access to a given instance. Consider this modification to the Config class:

```
template <typename Context> class Config { ...  }
```

We could then potentially simplify usage of the `instance()` method with `typedefs`:

```
typedef Config<eshell_sharing_context> ShellConfig;
typedef Config<qub_sharing_context> QubConfig;
...
ShellConfig & sc = ShellConfig::instance();
QubConfig & qc = QubConfig::instance();
```

There is at least one immediately-obvious down-side to designing *classes* to be Context-specific: different parameterized types lead to *inherently unrelated* Config types, and that will complicate things if you try to use both of them in similar code-contexts. For example, some otherwise-non-template functions may need to be converted to template functions to accept type `Config<T>`, as opposed to type `Config`. Long-term, such an approach unduly limits many classes, but in section 4 we will design a class for which this approach is ideal.

2.4 Context-specific data sources for Config

One feature of Config which, for simplicity, is now shown in the above code is the bits which loads the config file the first time `instance()` is called. The goal of this whole exercise is to allow eshell and QUB to use different config files, so it would obviously not be appropriate to load the same config file in `instance()`. There are multiple ways to solve that, a couple of which are briefly described here:

1. Use application initialization code to check if the Config object is empty. If it is, populate it from your chosen source.

2. Alow `instance()` to take an optional parameter, a string specifying an input file name. If called with this name it will load the file, otherwise it will return what Config object it has already created. In practice, you call `instance( "myfile" )` once during `main()` to populate the object.

3. Something entirely different.

The solution eshell takes #2, but it cannot be said that there is One Right Way to handle it - it's largely a matter of personal style and existing project conventions.

In this particular case, Config saves itself to it's original input file at destruction time, but that type of feature depends entirely on your objects. If Config did not do that, clients would be forced to do something like `Config::instance().save()` at shutdown. That approach is not useful in the case that `exit()` may be called from outside `main()`, as it requires that clients remember to save the config object (which they rightfully shouldn't have to do). This feature is significant, in terms of shared-object behaviour, because...

Features such as save-at-destruction actually reveal an inherent problem with Singletons: C++ does not (cannot) guaranty the destruction order of statically-created objects post-`main()`, just as it does not (cannot) guaranty their creation order pre-`main()`. Thus, if Config tries to save at destruction and it relies on a resource which has already been destroyed by the runtime environment, Config may get a dead reference when it uses that resource, causing *Undefined Behaviour*. Likewise, if a client object tries to use Config post-`main()`, it may step on a destroyed object. This problem is discussed in section 4.

2.4.1 Initializing Config pre-`main()`

Let's take a quick look at option #2 from the section above, and investigate the implications. As mentioned above, the client would normally be expected to pass the config file name one time during `main()` to initialize the Config object. Here's an approach which does not require a library's client to do anything: in the library code we can initialize the Config object pre-`main()` by doing this:

```
int eshell_config_bogo =

    (Config::instance<eshell_sharing_context>( "$HOME/.eshell.config" ),0);
```

That code goes into the global scope, and gets initialized when static globals are initialized - pre-`main()`. The syntax may look a bit odd: it's saying, assign zero to `eshell_config_bogo`, and while you're at it call `instance()`. This is simply a kludge to allow us to easily run arbitrary code pre-`main()`[2].

Likewise, QUB does something similar:

```
int qub_config_bogo =

    (Config::instance<qub_sharing_context>( "$HOME/.qub.config" ),0);
```

Post-`main()` destruction of static objects then triggers the saving of the config objects, to make application settings persistent. Such post-`main()` usage of objects has it's inherent dangers, the most glaring of which (undefined static object destruction order) is covered in section 4.

There are better (i.e., more generic) ways to populate our Config object, and we cover some of them sections 4.4 and 4.5.

3 Summary of Context Singletons and potential improvements

The above Config example is just one place of many where Context Singletons come in handy. i've found them useful for providing, for example, shared `std::maps`, for use as entity translation maps in (e.g.) simple XML parsers. Additionally, it is now trivial to provide Contexts with their own configuration space, where they cannot possibly hose any other data or be confused by other data. We could even go so far as to provide per-class Config objects. Config is definately *not* the only place Context Singletons are useful, by any means.

Some improvements which can (and should) be applied to Context Singletons are touched on below.

[2] Coincidentally, this kludge provides the whole basis for `class_loader`'s class registration technique.

3.1 Phoenix Singletons

Alexandrescu teaches us about Phoenix Singletons: Singletons which are "reborn" if they are accessed after their post-`main()` destruction by the underlying runtime environment.

Without this feature an application will crash post-`main()` if a shared object is referenced after it is destroyed.

This problem is discussed at length in *Modern C++ Design*, and a solid solution is provided in that book. Additionally, in section 4 we discuss the development of a `phoenix` class which provides phoenixing of arbitrary Context Singletons. Note that the two approaches are fundamentally different: *MC++D* covers true Singletons, whereas `phoenix` provides Context-specific shared objects.

3.2 Populating Context Singletons

During the development of this paper the topic of (re-)populating a Context Singleton came up. In common literature the topic of populating Singletons is not discussed, probably because such behaviour is inherently client/project/object-dependent. The next sections develop an approach which generically adds this feature to Context Singletons, allowing arbitrary initialization code to be assigned to each of them.

4 `phoenix`: a Context Singleton provider

The above sections define the concept of Context Singletons and explain enough for you to go write and use your own. This chapter explains an experimental class, `phoenix`, which provides Context Singletons with phoenixing behaviour (as defined in *MC++D*). Full disclosure: `phoenix` is based partly on *MC++D*'s code. In particular, i've stolen Andrei's trick of matching `atexit()`, manual dtor invokation and the `placement new operator` . That said, this implementation is quite different from that presented in *MC++D*.

`phoenix` is a rather simplistic implementation, but "appears" to cover most bases well enough for my purposes. In contrast to Alexandrescu's approach, `phoenix` does not work with true Singleton types. Contrariwise, `phoenix<T>` *subclasses type* `T`, adding phoenixing capabilities to it while hiding the `phoenix` parts of the data type from users of `instance()`. Any calls made via `phoenix<T>` will use the same instance of `phoenix`, and thus `phoenix` itself provides Monostate semantics even if the shared object type does not.

`phoenix` was developed for `class_loader`, part of the s11n project (`http://s11n.net`). Naturally, this paper evolved as a side-effect of having written `phoenix`. In turn, some features were later added to `phoenix` after they were first contemplated here.

4.1 The public interface (tiny)

```
template <

    typename BaseType,
    typename ContextType = BaseType[3]
    >

struct phoenix :  public BaseType {

    typedef ContextType context_type;
    typedef BaseType base_type;
    static base_type & instance();

};
```

Note that the `typedefs` are not strictly necessary, but are provided to aid any as-yet-unwritten type-determination code.

The latest code accepts an optional 3rd template parameter, described in section 4.5.

[3]This default value leaves significant room for client-side error in the form of mis-shared objects, so an explicit ContextType is recommended.

4.2 Implications of the interface

To continue the Config example, `phoenix<Config>::instance()` will *always* return a valid instance of Config. If the original object is destroyed, as happens at some undefined point post-`main()`, `phoenix` will create a new object *in the same memory space as the old one.* As explained in *MC++D*, C++ guarantees that the memory assigned to static objects will be available for the life of the application, so it is available for us to re-use via the *placement new operator* (covered in *MC++D* and *Effective C++*).

To avoid potential resource leaks caused by re-construction, `phoenix` must manually destruct any object it phoenixes. This support relies on sane `atexit()` behaviour in the underlying C libraries. As *MC++D* explains, such sanity is not always available! That said, the phoenixing behaviours cannot be guaranteed to behave properly on systems where `atexit()` does not "sanely" queue `atexit()` handlers which themselves are registered as a result of `atexit()`'s destruction process. Thus on systems with a "broken" `atexit()` the `phoenix` is theoretically neither more nor less safe than not using a `phoenix` at all.

4.3 An implementation

A first-go implementation is short enough that we can paste it in right here. The more interesting bits are highlighted in blue:

```
template < typename BaseType, typename ContextType = BaseType[4] >
class phoenix :  public BaseType {
public:

    typedef ContextType context_type;
    typedef BaseType base_type;
    static base_type & instance() {

        static this_type meyers;
        if( this_type::m_destroyed ) {
            new( &meyers ) this_type; // re-use meyers' address
            atexit( this_type::do_atexit );
        }
        return meyers;

    }

private:

    typedef phoenix<base_type,context_type> this_type;
    static bool m_destroyed;
    phoenix() { m_destroyed = false; }
    ~phoenix() { m_destroyed = true; }
    static void do_atexit() {

        if( m_destroyed ) return; // ensures that instance()call, below, is safe.
        static_cast<this_type &>(instance()).~phoenix();
        // ^^^ will eventually trigger BaseType's dtor

    }

};
template <typename T, typename C> bool phoenix<T,C>::m_destroyed = false;
```

A more complete implementation, including documentation, debugging output, and the object population approach covered in section 4.5, is available from:

`http://s11n.net/libs/`

[4] Again: this default value leaves significant room for client-side error in the form of mis-shared objects, so an explicit ContextType is recommended.

4.3.1 Sample invokation

```
struct my_sharing_context {};
typedef phoenix<Config,my_sharing_context> SharedConfig;
...
Config & c = SharedConfig::instance();
```

4.3.2 Open questions

Some questions come to mind about this implementation which i cannot immediately answer:

- What happens when `BaseType` has a non-virtual destructor? We never use `delete()`, relying on static destruction instead, so i *think* this is safe. i have successfully used this class with, e.g., `std::map<>` as `BaseType`, but maybe i just have a remarkably accomodating C++ platform.
- Is it possible that `m_destroyed` gets destroyed at a point such that the above usages of it becomes invalid? If so, is there a way to phoenix that `bool`, so to speak?

4.4 Repopulating a phoenixed object

Our first version of `phoenix` is missing a key feature: a way of populating the shared object. Obviously, such population is necessarily client-specific and depends completely on `phoenix`'s `BaseType` parameter. Here we'll cover one potential approach to how client-side code might take advantage of `phoenix` while also being able to control population of the object if indeed it gets phoenixed.

Consider our now-familiar Config class. As a container class, it has a method `empty()` which tells us if it's empty. In our library we assume that a Config object with no filename is not yet initialized. We can take advantage of that property and define a free function which we expect clients to use to get their Config object:

```
Config & config() {
    Config & c = phoenix<Config>::instance();
    if( c.filename().empty() ) { ...  assign filename and read file ...  }
    return c;
}
```

Again, it depends entirely on your object, personal style and existing project conventions. As the Master Painter, Bob Ross, undoubtedly would have said, "it's your code, so do it however you like."

Note that `instance()` always returns the same address, even for re-created objects, so checking for "rebirth" using a pointer comparison will not work.

In the case of eshell, Config saves itself during it's dtor, so a phoenixed Config object would re-load the state of the config as it was when the original static object was destroyed. So, practically speaking, it is as if the Config was never destroyed. What more could we want from a phoenix? There is, of course, the overhead of having to read the config file again, but this overhead seems trival in this case, considering the undefined results of the alternative. It is highly unlikely that an object would be phoenixed more than once, so a performance hit not likely to be an issue for most practical purposes. (Additionally, as the potential performance hit happens post-`main()`, users may never notice. ;)

4.5 Repopulating a phoenixed object, attempt #2

The first attempt at a solution to object-repopulation is effective, but leaves much to be desired in terms of modularity and... well, *style points*.

Another, more flexible, approach to re-population is to modify `phoenix` to accept an *initialization functor* as a template parameter, an instance of which may be used to initialize the shared object upon the first call to `instance()` and for any later re-creations. In the Config case we could, for example, supply a `ConfigInitializer` functor which knows what config file we want to use. For example:

```
struct ConfigInitializer {
```

```
        void operator()( Config & c ) {
            // populate c from $HOME/.myconfig
        }
    };
```

Sample usage:

```
typedef phoenix<Config,SomeContext,ConfigInitializer> MyConfig;
Config & c = MyConfig::instance();
```

`instance()` will trigger the `ConfigInitializer` during the initial call and for any call which triggers a rebirth of the shared object. You *gotta* love templates.

The complete `phoenix` source code includes this support and provides a default no-op functor which is suitable for most cases, so clients only need to provide the first template argument (the shared object type).

4.5.1 Sample use-case

As a test bed for the above code i've used my pet template-heavy library, `class_loader`. i replaced the statically-initialized global classloader path, which used to be a non-phoenixed Context Singleton, with a `phoenix`-style initialization functor like the one described above. While it requires about the same amount of code as the older approach, it provides the benefit of not setting up the shared Path object until the `phoenix`'s `instance()` is actually called, which could theoretically be never. While benefits of lazy instantiation are trival in this case, they could of course be non-trivial when using complex objects. Also, this approach provides modularity benefits which were not possible in the older code.

In addition to that trival case, i've replaced all off the Context Singletons in `libs11n` (that's no small number) with `phoenix`es, and to good effect: to-date `phoenix` has saved me from at least four separate post-`main()` dead references.

5 Final Summary

We've discussed Context Singletons... YES, i KNOW they're not *really* Singletons, but i don't have a better term for them. We've shown a couple uses for them, including a couple practical cases where they are helpful. We briefly touched on the post-`main()` issue which traditionally plagues globally-shared objects, and we've hopefully provided an adequate interface for bypassing that problem.

The Context Singleton idiom can be used in many cases where Singletons are traditionally used, without forcing object designers to write non-reusable Singleton-only classes. With the addition of `phoenix` they can be used more safely than non-phoenixing Singletons, and they can do so without having to subclass a Singleton type (nor implementing Singleton behaviour internally).

With the addition of Initialized Phoenixes (if i may now coin a term), we've also got phoenixes which can fully restore their state upon rebirth. (While the Dali Lama may have had this feature all along, most of us have not! ;)

The implementations presented here are provided to stir up ideas: they are not intended to provide "definitive implementations" of these concepts. i'm quite certain that you can come up with your own ways to add similar features to your toolboxes...

And, as a final note... `phoenix` is the first class i've written which has more than 100 times as much documentation as source code.