Classloading in C++:
Bringing classloading into the 21st century

stephan@s11n.net

3rd January 2005

Abstract

CVS info: $Id: classloading_cpp.lyx,v 1.8 2005/01/03 18:37:55 sgbeal Exp $

This paper discusses a type-safe, linkage-agnostic approach to dynamically loading objects (i.e., classloading) in C++. In particular, it shows that classloading can be easy to implement, 100% typesafe, and even trivial to use in client code. Here we will see how we can completely eliminate many of the troublesome details traditionally involved with writing classloaders.

This paper is targetted at intermediate-level C++ programmers, and makes many assumptions about the reader's previous knowledge. The techniques presented here, however, are suitable for use by novice C++ programmers, assuming they can absorb enough of the material to understand how to implement it.

Contents

1 Introduction

In application development, particularly in large applications, it is often desirable to break the project up into smaller, more manageable chunks. One possibility for doing so is the ability to load classes at runtime, as needed, as opposed to linking them all in with the application (as in so-called "monolithic" applications). Classloading introduces a number of advantages and a couple of disadvantages for a project. This paper attempts to show that the disadvantages can be nearly completely eliminated, and that getting at the advantages is essentially trivial to do, allowing programmers to reap the benefits implied by dynamic loading while not requiring them to bang their heads over the gory details of, e.g., looking inside a DLL to see if it contains the class they're looking for. That said, this paper is *full* of details, but understanding them *all* is not necessary for using the framework we will develop here.

The techniques we will show here require a fairly recent C++ compiler with good support for class templates. A subset of the features we will see here could potentially be implemented on older compilers, but such projects are outside the scope of this paper.

1.1 Change History

Newest at the top:

- 3 Jan 2005:
  - Minor text/code corrections.
  - Added notes about an alternative (and cleaner) approach to supporting abstract base types from `instantiator`.
- 20 Sept 2004: minor touch-ups and reorganizations
- 21 Aug 2004: initial publication

1.2 Disclaimer

i must admit that the vast majority of this paper was written over about a 4-hour period (i type quickly), so it may very well contain some mistakes, in particular copy/paste errors vis-a-vis the `instantiator` class and macro-based code. If you find a problem while using any of the code shown here, or using similar techniques with your own code, please don't hesitate to ask for help.

1.3 License/Distribution Policy

This document and all source code directly related to it are released into the Public Domain.

> "*... do as thou wilt shall be the whole of the law.*"
>
> Allister Crowley

1.4 Assumptions

Throughout the rest of the paper we will assume that the reader knows essentially what classloading is, at least on a theoretical level, and understands at least a couple of the cases where it can be useful. Also, we will assume that the reader knows what the system call `dlopen()` does[1], again at least on a theoretical level[2]. Practical experience with `dlopen()` is not required, as it will not directly affect client usage of this framework.

If you are unfamiliar with `dlopen()`, please go read some documentation on it (on Unix-like systems, try `man dlopen`) to get an idea of what it is for before reading this paper. You do *not* need to know *how* it works, but you should understand *what it does*. If you're in a hurry, i will try to give you a head-start by summarizing:

> `dlopen()` opens *Dynamically Loadable Libaries* in such a way as to allow the currently-running application to access data contained in those libraries.

Dynamically Loadable Libraries are commonly known as *DLL*s or, on most Unix-like systems, *Shared Objects* (or SOs).

If you don't know what a DLL (or ".so") is, i highly recommend doing some background reading before continuing, as we will not elaborate on them much here, and they play an important role in the dynamic loading of types. That said, the framework developed here can load types regardless of whether they live in a DLL or in the main application. There are in fact some very valid uses for classloaders even when a project is not using DLLs, some of which we will touch on later.

Another topic mentioned now and again in this paper is "name mangling." If you don't know what this means, you might be able to get through this paper by deducing or guessing, but i highly recommend searching the web for some information on it so that you understand why it exists and why it is a problem for developers when doing low-level work like DLL loading. (Errr... let me rephrase that: *was* a problem!)

Anonymous namespaces. If you don't know what these are, and especially if you don't understand their implications (*particularly* in header files), *please* go read up on them in your favourite C++ text. These are *crucial* to our eventual solution, and *must* be understood before some of the solutions to our problem will make much sense.

1.5 Motivation

Oh, boy... where to start? Answering this could easily turn into a paper by itself, but i'll attempt to summarize as concisely as possible:

Once upon a time, i needed the ability to dynamically load classes from C++ code. The only classloading code i had access to was based on the Qt library (www.trolltech.com), and did some embarassingly low-level tricks vis-a-vis name [de]mangling. In short, i didn't find it suitable for use outside of that particular project (qub.sourceforge.net), and set out to find a Better Way.

After reading up on `dlopen()` and it's related functions, i implented a basic classloader, but was appalled to find that it would allow me to cast one dynamically-loaded type to another (completely unrelated) type with the same signature and class size (e.g., identically-defined `foo::Foo` and `bar::Foo`). After this depressingly inadequate behaviour showed up i went on a crusade to find an Even Better Way of loading classes in C++.

While i have, in fact, found an Even Better Way, and have a working implementation which i am quite pleased with, this paper does not cover that implementation. Instead it covers the techniques that library uses, because they are generic enough to be used in a variety of contexts.

[1]For Windows platforms the equivalent is `LoadModule()`, but i have no personal experience with it so it won't be covered in this paper. Readers familiar with GNU's `lt_dlopen()` may mentally substitute that function for `dlopen()`, as it performs the same job and has essentially the same interface, and thus is compatible with our treatment of `dlopen()`.

[2]Assuming that you do know what `dlopen()` is, you may be surprised to learn that it's cousin, `dlsym()`, is, despite all common wisdom to the contrary, *100% unnecessary* for classloading. In fact, classloading becomes much simpler *and* much safer without it.

While i cannot claim to be a C++ guru of any sort, nor am i a ”bleeding edge” developer, i do believe that the techniques developed in this paper had not been used in C++ projects until the original implementation of my own classloader in September of 2003. If someone can show me prior art, i will happily concede the claim of having come up with this general technique to classloading.

1.6 Brief overview of problems with traditional classloaders

Name mangling. Argh. One of C++'s additions to it's C heritage is the advent of name mangling. While this was a necessary addition to the language, it poses no end of problems for people wanting to load classes from DLLs. Programmers have traditionally gotten around it by using C-based functions to inject functions and types into a DLL, as C does not mangle names. Or, if they chose not to work around it, but *with* it, by implementing demangling code, they have limited themselves and their code to a specific compiler and linker (or even specific versions of those tools).

Type safety. As one might assume, loading a type from a DLL requires knowing about a type which... *we don't know about*! Coders have traditionally used in-DLL ”factory” functions for accessing unknown types: they try to decipher function names (possibly mangled) which live in the DLL and call those to create new objects. Aside from the potential name mangling problem, there is a just-as-devious problem: we have traditionally used (`void *`) to get data back from a DLL, and we cast it to whatever type we *think* it is. i hesitate to say this, but relying on `void` pointers that way in an object-oriented world is just plain *Evil*.

The techniques developed in this paper completely eliminate not only the use of (`void *`) but also of casting - *no* casts will be used, neither in the back-end nor in client-side code. If you don't quite believe that statement, just bear with me for a while and we will prove that it can be (and has been) done.

1.7 The magic trick

We're going to jump ahead here a bit, in the hopes that doing so will make it clearer to the reader where we're headed. There is a ”magic trick” when can be used to eliminate the most notorious problems associated with classical classloading approaches:

- *All* uses of (`void *`).
- *All* name-mangling related problems.
- *All* type casts, while still retaining *100% type-safety*, even for types loaded via DLLs.

All? Really *all*? Yes - *all* of them.

If you don't believe that it is possible to achieve these goals:

1. i don't blame you. i didn't, either. In fact, i'm ashamed to admit that the type safety is a natural (but unanticipated) side-effect of ”the magic trick”, and is a property of this model which i only noticed *after* implementing it (and literally wondering, ”wait a minute... i need a dynamic cast around here somewhere...”).
2. Keeping reading - we will convince you by the end of this paper.

So what is this magic trick?

It is deceptively simple, and i am continually surprised that i haven't seen it published before:

> We set up a system where classes which live inside a DLL *register themselves* with the classloader when the DLL is opened (i.e., during its static data initialization phase).

The more perceptive readers may find that that description alone is enough to invoke ideas for an implementation. That is, some of you will be able to predict where that deceptively simple statement is leading us. Be aware, however, that the path which takes us to an implementation is *not* as straightforward as it may seem, due to largely to our need to avoid ODR (One Definition Rule) violations. However, it also is not difficult, and is *downright easy* to do once one finds an approach which can avoid the ODR-related problems. Read on...

1.8 Problems we will face

Here is an overview of the problems we will face while developing a classloading framework, listed roughly in the order in which we will address them[3]:

1. Non-intrusiveness. That is, how do we make a type classloadable without having to change it? We will achieve this via a combination of templates, anonymous namespaces, and limited use of macros to generate some back-end types (don't panic about the use of macros, please).
2. Registration of classes to be loaded. Ideally we should not require clients to do this, as that defeats the ability to load types without knowing their exact types (we must *always* know their base-most type, if we are to avoid stooping to `reinterpret_cast<>()`). This problem actually has a several sub-problems:
   (a) Mapping class *names* to classes. *No*, `typeid::name()` *is not acceptable*, as it *officially* provides *undefined results.* C++ unfortunately offers no in-language solution to this, so we will have to deal with it by ourselves.
   (b) Registering polymorphic types with their appropriate base-type classloader.
   (c) Handling abstract classes - we want them to be classloadable as well (even though we can't *actually* instantiate them, making them loadable simplifies things later on).
3. ODR violations. These are a PITA[4], but we will tame them through a combination of class templates and anonymous namespaces.
4. DLL symbols. While classloaders traditionally rely on symbols compiled in to a DLL, we will *completely* eliminate this requirement[5]. The solution to this does not actually require much conscious effort - it is actually a side-effect of the architecture.
5. Finding DLL files. While filesystem-related operations are inherently platform-specific, we will see that this aspect of classloading is trivial to separate from the core framework, and can easily be implemented on any platform, completely independently of any client-side classloader interface.

As we progress we will take on each of these problems. Understanding *why* these points are important to classloading is critical to understanding some of the implementation details of our framework.

We will take the above challenges and develop a project-neutral and, more interestingly, *linkage-neutral* approach which has, so far, proven to be useful in a wide variety of client applications without unduly imposing on them. By "linkage-neutral", we mean that class registration will be done is such a way that the detail of whether the class is actually in a DLL or linked in with the main application is irrelevant for classloading purposes. That is, the classloading process (from the perspective of client code and classloader registration) is 100% identical, regardless of whether we load classes via DLLs or which are linked in to our main application.

1.9 Reference implementations

The ideas developed here are a more generic treatment of the techniques discussed in the documentation for the s11n project's classloader framework, available from:

`http://s11n.net/class_loader/`

This paper refines the topics covered in that library's documentation into a standlone, project-neutral treatment, without actually providing a 100% complete framework. The reader is encouraged to study the `class_loader` documentation and implementation if he would like to see a working implementation of the techniques described here. (Keep in mind, however, that that project is a test-bed for these techniques, and is starting to show it's age.)

That framework is used *extensively* in the libs11n source tree and several other project trees i work on, and to good effect. Its `cllite` interface takes classloading to a whole new level of ease-of-use over traditional classloaders, and this paper will leverage from `cllite`'s implementation and apply tricks which have come about through its developement (after all, re-inventing the wheel is bad enough, but re-inventing a wheel you wrote yourself is downright sacrelidge! ;).

Also, the approach covered in this paper provides the factory layer of P::Classes 2.x:

[3]Or, more dramatically, the order in which we will beat the living shit out of them.

[4]Pain In The Ass.

[5]This is, in fact, one of my favourite features, if for no other reason than that it successfully defies all conventional wisdom on the topic. ;)

```
http://pclasses.com
```

An online coding colleague of mine, martin f. krafft (yes, he prefers it spelled lower-case), has also developed a related, though very different, classloading framework, available from:

```
http://sourceforge.net/projects/libfactory
```

Note that his framework is *significantly* different from the one we will develop here, but it also has some interesting features which we will not cover, such as the ability to use non-default constructors when creating new objects via factories. He also hasn't documented `libfactory` all that much, but we must forgive him:

- That's my job - i signed up for it. Instead i've been hacking on `s11n` and writing papers such as this one. :/
- He's been writing a book about Debian Linux, which is due to the publisher any day now, and this eats up his programming time.

That said, his code is clever and well worth a look.

2 The first step: creating a generic factory

How do we create new objects without necessarily having to know their exact types? As developers much wiser than myself have come to agree, so-called *Factories*[6] are a great solution to this problem, and we will leverage off of that idea to achieve this goal.

Our list of requirements for our factories and the types they create:

- We can load classes without having to modify them. That is, we do not want to have to require anything like a static member function called `MyType::create()`.
- We require a lookup key for loading types, and each base type/key pair is assumed to be unique. Traditionally, the key is the string form of a type's name, but it need not be, and in fact when working with template types (which have non-trivial names) using simplified names is... well, simpler.
- We *loosely* require that classloadable types of `struct` or `class` type and be *Default Constructable*, but the default constructor requirement can be gotten around by using custom factories.

2.1 An initial try

Okay, so we want to make factory functions. Here's what one might look like:

```
MyType * create_MyType() { return new MyType; }
```

While that may seem straightforward enough, it has a number of problems:

1. It will get name-mangled, so we cannot reliably know what it's name will be once it is compiled. i.e., we cannot reliably do symbol lookups for it. Yes, we could use an `extern "C"` block to get around this, but it turns out that we don't need to, so we won't.
2. It cannot polymorphically create and return subclasses to `MyType`, which is one of the most significant reasons for having a factory in the first place.
3. It hard-codes `MyType`, the implication of which is that we would need such a function for every type we want to create a factory for.
4. The above implementation will cause ODR violations if it is included in a type's header. If it included in an implementation file (i.e., if it compiled to an object file) then clients cannot see it, and therefor cannot use it. (This is where the practice of using `dlsym()` comes from, to fish out function names from DLLs. As we will come to see, that is now obsolete.)

[6]See Alexandrescu's *Modern C++ Design* for a complete treatment of this topic.

We won't dwell on problems #1 and #2, but we will show one of C++'s approaches to eliminating problem #3:

```
template <typename T> factory() { return new T; }
```

This function still has the problem of non-polymorphically creating a `T`, but it's a start, and we will work from there. We will consider this problem more very soon.

Solving problem #4 satisfactorily is tricky, but once the solution is seen it's likely to be one of those "oh, *duh!*" moments. (i know it was for me, anyway.)

2.2 An improved attempt

Without going into too much detail about why we want this, here we will look at a simple interface for polymorphically loading types. One possibility is something along the lines of:

```
template <typename T> classload( const std::string & classname );
```

Presumably, the function would have access to a map of classnames to factories, such that it could forward requests for classname to a factory which can return an object of that type. It is assumed that any classname passed to this function represents a subtype of `T`, or is `T` itself.

The implementation of such a name-to-factory map could be done any number of ways, so we won't go into specifics about it. One recommendation is to use something like the following:

```
typedef SOMETYPE * (*factory_type)(); // factory function
typedef std::map<std::string,factory_type> factory_map; // map class names to factories
```

So where do we get `SOMETYPE`? That's a detail we will address as we progress. Keep in mind that the above signatures are not set in stone - they are examples while we work through the details of out interface.

2.3 Attempt #3: a final proposal

Because i'm impatient, and because you probably want to start classloading your types, let's go ahead and steal a bit of code from `cllite`, and present a basic factory type which has proven to be useful for generic classloading.

```
template <typename BaseT, typename SubT = BaseT> // SubT must be-a BaseT.
struct object_factory {

    typedef BaseT result_type;
    typedef SubT actual_type;
    static result_type * new_instance() { return new actual_type; }
    static result_type * no_instance() { return 0; } // explained below

};
```

That should be pretty straightforward, except perhaps for the `no_instance()` function. We won't go into detail about that now, except to say that it's a convenience factory for "instantiating" abstract types (i.e., those which cannot be instantiated). We want to be able to classload subclasses of abstract types, and therefor we need a classloader which can deal with abstract types. The `no_instance()` function provides that feature to the classloader. If we reference `object_factory<MyAbstractType>::new_instance()` in our code compilation will fail because `new MyAbstractType` is not a valid expression. We will use `no_instance()` later on, but please don't think too much about it now, as it is actually a minor detail and does not deserve too much attention.

The above factory class doesn't do any classname-to-type mapping, but we don't want it to. It only acts as a basic factory, from which our classloader can create new objects. This type may be specialized (or partially specialized) to do things like create objects using non-default constructors, etc., but that is beyond the scope of what we are trying to accomplish, so those details are left to the implementor. It should be noted that we will not normally use `object_factory` instances, but will instead use it as a "type holder." Having a class, as opposed to free functions, gives us a convenient place to store type information, so we will take advantage of that capability. That said, our factory could also be implemented in terms of free functions with little effort.

Side-note: it might be useful to add a template parameter, `IsAbstract`, to `object_factory`, and specialize the type so that it's `new_instance()` function returns the same as `no_instance()`. We will not explore that option here, but it is potentially useful, especially if one has a template meta-programming technique which can automatically determine whether a type is abstract or not (i.e., an approach which keeps clients from having to specify that parameter).

Let's see how `object_factory` is used:

```
MyType * m = object_factory<MyType>::new_instance();
MyType * ms = object_factory<MyType,MySubType>::new_instance();
MyType * mss = object_factory<MyType,MySubSubType>::new_instance();
```

This brings us to an imporant aspect of our framework: classloaders are "keyed" to *base-most* types. Notice how we use `MyType` as the base-type parameter for `MySubSubType`, instead of using `MySubType` as it's base. This is important to keep in mind, so that we can be sure that we end up calling the proper factories from our classloaders. Note that "base-most" need not be the *real* base-most type, but is the base-most type *for purposes of our classloading framework.* To make an analogy against a commonly-used C++ toolkit: when classloading objects from the Qt toolkit (http://www.trolltech.com), we might want to use `QWidget` as our base type, instead of `QObject`, even though `QWidget` *is-a* `QObject`.

It must be stressed that `object_factory` is *not* the end-all, be-all of object factories: it is only a suggestion, and one we will use for demonstration purposes as we develop our framework. Implementors are encouraged to design their factory class (or functions) however they like.

We now have a generic factory type, but we're still far from the end. Let's continue...

2.4 Final note: object ownership

As a general rule, objects returned from a factory are *owned by the caller*, and it is the caller's responsibility to delete them. There are exceptions to this rule, of course, which is why we say, "as a general rule." For example, `cllite` offers a way to classload objects which are shared - all classloads of the chosen type return the same object, and the framework cleans them up when the application ends. We will not go into the details of that here, except to say that it is not difficult to add this feature once the basic framework is in place. (You guessed it: *an excercise for the reader*!)

3 Loading types by name

This section shows one potential approach to registering class names with factories, such that we can do the following:

```
MyType * m = classload<MyType>( "MySubType" );
```

and get a `MySubType` object.

3.1 A type for holding registrations

Again, we will go ahead and steal an object from the `cllite` framework - one which allows us to map string names to arbitrary object factories (not necessarily `object_factory`, though we will use that one here). The class presented here is simplified somewhat from the one found in `cllite`: the `cllite` version has a feature we won't be discussing here. If you're interested, see the original code and take a look at the documentation for the `ContextType` template parameter used by the `instantiator` type.

Before we show the code, let's go over our requirements for his type:

- It must be able to map keys to class names. The keys need not be strings, but non-strings are typically not useful for looking up DLLs, so in practice we almost always use `std::string` as our lookup key type. There are cases where mapping, e.g., `enum` entries as keys is remarkably useful. For example, mapping error numbers to error handler classes.

- Assuming the registered base-most type is polymorphic, `instantiator` must be able to polymorphically load sub-types of that base.

That should be simply review by now, so let's move on.

3.1.1 The `instantiator` class

The `instantiator` class is essentially a static classloader, with no built-in DLL lookup support, and will be used as the basis for our dynamic classloader later on. With this type in place we can already load types as if they were being loaded dynamically.

```
template < typename BaseType, typename KeyType = std::string >
class instantiator
{
public:
    typedef BaseType value_type; // base-most type
    typedef KeyType key_type; // lookup key type
    typedef value_type *( *factory_type ) (); // factory function signature
    typedef std::map < key_type, factory_type > object_factory_map; // key-to-factory map
    // Uses a mapped factory to return a new value_type object
    // polymorphically:
    static value_type * instantiate( const key_type & key )
    {
        typename object_factory_map::const_iterator it = factory_map().find( key );
        if ( it != factory_map().end() ) // found a factory?
        {
            return ( it->second ) (); // run our factory.
        }
        return 0; // no factory found :(
    }
    // Maps the given BaseType factory with the given key.
    // You may pass a factory which returns a subtype, but the
    // default factory will always return an actual value_type object
    // (assuming value_type is not abstract, in which case it returns
    // 0).
    // Note that by providing a default factory here we make a trade-off:
    // this code will not compile when value_type is abstract.  If we
    // force the user to specify a factory we can support abstract types,
    // but we almost double the amount of code needed to register types,
    // as demonstrated later.  Later in this paper we will mention a cleaner
    // solution which allows us to keep the default argument here and support
    // abstract types.
    static void register_factory( const key_type & key, factory_type fp = 0 )
    {
        if( !  fp ) fp = object_factory<value_type>::new_instance;
        factory_map().insert( object_factory_map::value_type( key, fp ) );
    }
    // Registers a factory for a subtype of base_type.
    // This is the same as calling register_factory( key, mySubTypeFactory ),
    // except that it installs a default factory rerturning a SubOfBaseType
    // if the client does not provide one.  SubOfBaseType must be a publically
    // accessible ancestor of base_type, or must be base_type.
    // Please see the notes in register_factory() regarding the second
    // parameter:  the same notes apply here.
    template <typename SubOfBaseType>
    static void register_subtype( const key_type & key, factory_type fp = 0 )
    {
            if( !  fp ) fp = object_factory<value_type,SubOfBaseType>::new_instance;
            register_factory( key, fp );
        }
        // Returns our factory map:
        static object_factory_map & factory_map()
        {
            static object_factory_map meyers;
            return meyers;
        }
        // Tells us if a given key is registered with a base_type
        // factory.  Only rarely useful, but here it is...
        static bool is_registered( const key_type & key )
        {
            return factory_map().end() != factory_map().find( key );
        }

    };
```

One immediate thing to notice is that `instantiator` only takes a *base type* as a template parameter, and does not use a sub-type except in it's registration function, `register_subtype()`. Also notice that all functions are static. While this might seem unnecessary, this decision has a history:

C++ will, for good reasons, only allow us to load a single definition of any given class. If we allow `instantiator` to use a private map, and thus allow it to return differently-created objects for a given key, we will probably only confuse users, who probably expect all loads of a given type to return objects which have been created in an identical manner.

Aside from that reason, this interface has shown to simplify client-side use, as opposed to an interface which requires that the client populate each instance of `instantiator` with all required factories.

That said:

- The `cllite` code allows a way to share `instantiators` within a "context type", so that all `instantiators` within a given context use shared factories, while not sharing factories between different contexts. See it's API documentation for details. (For more on *Context Types*, see `http://s11n.net/papers/`)
- There are valid uses for a non-shared `instantiator` factory maps, and readers are encouraged to implement their `instantiator` this way if they prefer to. (But when it becomes tedious to use in client code, kindly remember that i told you so. ;)

Note also that we did not add a facility to register abstract types in the above interface. Actually, we did, but it is hidden away in the ability to pass arbitrary `factory_type` functions to the various registration functions. We'll see how that's used later on.

3.2 Registering and loading classes via `instantiator`

The instantiator class, as presented above, is complete and ready to use. Here's how we can register types with it:

```
    typedef instantiator<MyType> IT;
    IT::register_factory( "MyType" );
    IT::register_subtype<MySubType>( "MySubType" );
    IT::register_subtype<MySubType>( "AliasForMySubType" );
    IT::register_subtype<MySubSubType>( "MySubSubType" );
    IT::register_subtype<UnrelatedType>( "Foo" ); // COMPILE ERROR: UnrelatedType is not a MyType
```

Seems simple enough, doesn't it? It *is* simple[7]. We will, later on, hide this process from client code, but it is instructive to see how it works. If we do not hide this from clients, they will be required to know the exact types of each `MyType` subtype, which is one of the things we are so desparately trying to avoid. Indeed, requiring clients to know about each concrete type eliminates (almost) the requirements for a classloader[8]. Manual registration

[7]If we allow a default 2nd argument to register_xxx() it becomes significantly simpler, as we could lose literally half of this code.

[8]i say "almost" because classloaders can ease lots of use cases even when all concrete types are known. Also, adding this support in advance means that dynamic loading comes "for free" once classes need to be dynamically loaded.

of types is sometimes helpful, e.g., for mapping one type to multiple keys. Consider, e.g., registering the key "DefaultDocClass" to an arbitrary Document factory, perhaps configurable by the user.

Note that we choose not to pass our own factories to the registrations, and thus rely on the default factory implementations. The `object_factory` is suitable for almost all cases. Some cases in which we might want to pass custom factories are:

- To enable "loading" of abstract types (as we will do later on). Note that we should remove the defaulted value for the second argument to `register_xxx()`, as explained in the comments for that function.
- If we need to construct objects using a non-default constructor, do accounting or logging work, pull objects from a shared pool, or similar tricks.

Now let's try to load some objects:

```
MyType * m = IT::instantiate( "MyType" ); // a MyType object
MyType * ms = IT::instantiate( "MySubType" ); // a MySubType object
MyType * msalias = IT::instantiate( "AliasForMySubType" ); // a MySubType object
MyType * mss = IT::instantiate( "MySubSubType" ); // a MySubSubType object
MyType * ohno = IT::instantiate( "NotRegistered" ); // ohno == 0
```

That's all there is to it. *We now have a basic classloader!* It includes all of the facilities we will need later on, with the exception of DLL lookups (which isn't as difficult as it sounds, and doesn't really belong in this interface, anyway).

Now that wasn't difficult, was it? The tricky parts are still to come, but that wraps up the development of the back-end for registering and loading classes.

3.3 Factory registration, part 2

Here we will develop a technique for registering classes in such a way as to avoid ODR violations, and which can be used in a non-client- and non-class-intrusive manner.

Without going into the long history about experimentation with different registration techniques (i don't want to re-live those grey-hair-inducing weeks!), we will go ahead and jump directly to one approach which has worked well for `cllite`.

Let it be said that the most notorious problem we face is ODR violations - duplication of function and class definitions which will show up either at compile-time or link-time (depending on many factors). C++'s greatest asset to us in this regard is the so-called *anonymous namespace*. These can be used to isolate a bit of object code within a single compilation unit and, by extension, within a library, such that like-named types which are encapsulated in anonymous namespaces do not cause ODR violations when the object files containing them are linked together. What does all that mean? It means "great!" Don't worry about gory the details just, just try to follow along ;). Readers who want to know the technicalities of it are encouraged to read up on the implications of using anonymous namespaces.

A secondary, but no less troublesome, problem is that we want our factory registrations to happen "automatically", without requiring client code to register the classes, nor requiring classes to register themselves.

In effect, what we need is to be able to create a *unique* type which is associated with the types to make classloadable and the *names* of those types. Templates provide most of the solution to this, but do not provide the whole solution: we still need to avoid some more ODR violations. To do this we will use anonymous namespaces.

Let's take a look:

```
namespace { // anonymous ns, important for linking reasons.

    template <class BaseT, class SubT = BaseT, class KeyT = std::string >
    struct classloader_reg_placeholder {
        static bool reg; // A no-meaning placeholder.
    };
    } // namespace
```

As you can see, this class simply holds one data member and has no functions. We will see what the placeholder variable is for soon. Just keep in mind that it has no meaning at all in and of itself, and could be of any type. We choose `bool` because we know it is small[9], thereby keeping the back-end overhead low.

Note also that we do not declare the static `bool` outside of the class, as we normally must do with static variables. We will provide these later, on a per-loadable-type basis.

Now we need a way of instantiating this "bogus" type for each class we want to make classloadable. The combination of template arguments and an anonymous namespace allows us to enforce uniqueness on a per-compilation-unit basis.

One straightforward solution to forcing instantiation is the definition of a simple set of macros, such as these (again, stolen from the `class_loader` tree):

```
    #define cl_CLASSLOADER_REGISTER(BaseT,SubT) cl_CLASSLOADER_REGISTER2(BaseT,SubT)
    #define cl_CLASSLOADER_REGISTER1(BaseT) cl_CLASSLOADER_REGISTER2(BaseT,BaseT)
    #define cl_CLASSLOADER_REGISTER2(BaseT,SubT) cl_CLASSLOADER_REGISTER3(BaseT,SubT,std::string)
    // #3 is only useful for KeyT == std::string.  It is here only for consistency.
    #define cl_CLASSLOADER_REGISTER3(BaseT,SubT,KeyT) cl_CLASSLOADER_REGISTER4(BaseT,SubT,KeyT,#
    SubT)
    #define cl_CLASSLOADER_REGISTER4(BaseT,SubT,KeyT,ThekeY) \
    namespace { \
        bool classloader_reg_placeholder< BaseT, SubT, KeyT >::reg = \
        (instantiator< BaseT, KeyT >::register_subtype< SubT >( ThekeY ),true); \
    }
```

Here's a pneumonic to remember which macro to call: the numeric suffixes in the names represent the number of arguments they take.

Let's summarize what these will do:

> During the static-initialization phase of the code which includes these macro calls, the placeholder variable will be initialized. As a side-effect, this initialization triggers a call to `instantiator::register_subtype()`. This, in turn, registers a factory.

Pretty straightforward, even if the details are a bit unsightly.

There are a number of limitations to this macro-based approach, and later on we will cover a more generic, but slightly more verbose, approach. For now let's just concentrate on what it does and it's implications.

Now, let's show a usage example, using our now-familiar `MyType`. From global-scope code we should call:

```
    cl_CLASSLOADER_REGISTER1(MyType);
    cl_CLASSLOADER_REGISTER2(MyType,MySubType);
    cl_CLASSLOADER_REGISTER2(MyType,MySubSubType);
```

Ideally these should be called from the header files containing the declarations (not necessarily the *definitions*) of the classes, but they could be in any header which has access to the types being registered. It might be useful, e.g., to dump the registrations into a single project-wide header, like `cl_registration.hpp`, and include that from any code which needs to do classloading.

That's all there is to it, at least for most classes - types with commas in the names break these macros, a problem we will fix later on.

The trick of it is that the placeholder type must be available to whatever code tries to classload the given type. We generally do this by calling this macros from some header file[10]. The important thing is that code trying to classload a given type must have a access to an *instantiation* (in template terms, not standard OO terms) of the placeholder class, which essentially boils down to having to put this registration code into header file. The main reason for this is that template types are not instantiated until they are *called*, which means if we put them in an implementation file and never instantiate them (again, in the template-based sense of the word), the registrations will never happen. (*Been there, done that!*)

[9]Historical note: gcc 2.95.2 sported a 4-byte `bool`, at least on Solaris platforms.

[10]Could be any header, or even an implementation file (under the right circumstances). Readers who want the full story are referred to the `class_loader` library manual, which covers this aspect in a gross amount of detail.

3.4 Done!

We're done with the most important parts of our classloading framework! We can't yet do DLL loading... actually... that's a lie. Let's reword that: we haven't yet seen how to do DLL loading. We now have everything we need in order to be able to register types and to instantiate them, using `instantiator::instantiate()`.

Before we move on to actually loading DLLs, though, let's tighten up the above macro interface into something more flexible, more maintainable and more robust...

3.5 Cleaning up the macro interface

As mentioned above, the registration macros we have seen have a number of limitations. One more generic approach is to use what i like to call "supermacros". Supermacros are discussed at length in a paper of their own, available at:

```
http://s11n.net/papers/
```

We won't go into detail here about what supermacros are, but we will create one and show how to use it to register any types, including those which have unusual characters in their names (like `std::map<foo,bar>`).

3.5.1 Registering via a supermacro

First, we write a small header file to take the place of the macro-generated code shown above. Before we show it's code, it is informative to see how it is used. So let's register `MyType` and it's subtypes:

```
#define CL_TYPE MyType // the type we want to register
#define CL_TYPE_NAME "MyType" // the name of the type (need not be the same as the type)
#include "cl_reg.hpp" // include the supermacro
```

If `MyType` is abstract we need to add one line before including `cl_reg.hpp`:

```
#define CL_ABSTRACT_BASE 1
```

That tells the supermacro to register a no-op factory for the type.

Registering a subtype is almost identical:

```
#define CL_TYPE MySubType
#define CL_BASE_TYPE MyType // associate MySubType with MyType's classloader.
#define CL_TYPE_NAME "MySubType"
#include "cl_reg.hpp"
```

Note, however, that `CL_ABSTRACT_BASE` is never required when registering a subtype, only when registering the base-most type.

Remember, these registrations would normally happen in their respective headers, though they may also be bundled up together in a separate file.

The `CL_TYPE_NAME` macro has an interesting implication. In particular, we can use any *string name* we want to! The only limitation is that we must not register two types with the same name *and same base class*, or else the last-registered factory will be the one which is actually registered. Since static initialization order is undefined in C++, we cannot rely on any particular factory registering before or after another. Thus, we should avoid using duplicate keys within the contexts of a single base type's classloader.

One implication of this property is that we can use, e.g., the name "map" for all `std::map<>` classloaders without having a problem! Why? Because each instantiation of `map<X,Y>` is actually unrelated to all other instantiations of `map<>`, which means that each instantiation has its own classloader. This is, like it or not, a property of templates.

3.5.2 Implementing a registration supermacro

Now let's look at the contents of the `cl_reg.hpp` supermacro, in all of it's glory:

```
#ifndef CL_TYPE
# error "You must define CL_TYPE before including this file."
#endif
#ifndef CL_TYPE_NAME
# error "You must define CL_TYPE_NAME before including this file."
#endif
#ifndef CL_BASE_TYPE
# define CL_BASE_TYPE CL_TYPE
#endif
namespace { // again, important for linking reasons

     # ifndef cl_CLLITE_REG_CONTEXT_DEFINED
     # define cl_CLLITE_REG_CONTEXT_DEFINED 1
         ////////////////////////////////////////////////////////////////
         // we must not include this bit more than once per compilation
         // unit...
         ////////////////////////////////////////////////////////////////
         // A unique (per Context/per compilation unit) space to assign
         // a bogus value for classloader registration purposes.
         template <typename Context>
         struct cllite_reg_context {
             static bool placeholder;
         };
         template <typename Context> bool
             cllite_reg_context<Context>::placeholder = false;
     # endif // !cl_CLLITE_REG_CONTEXT_DEFINED
     // The rest of the supermacro may be included multiple times...
     ////////////////////////////////////////////////////////////////////////
     // Register a factory with the classloader during static initialization:
     bool cllite_reg_context< CL_TYPE >::placeholder= (
     #ifdef CL_ABSTRACT_BASE
         // register a no-op factory:
         instantiator::register_factory< CL_TYPE >(
             CL_TYPE_NAME,
             object_factory< CL_TYPE >::no_instance
             ),
     #else
         // register the default factory:
         instantiator::register_subtype< CL_BASE_TYPE , CL_TYPE >( CL_TYPE_NAME ),
     #endif // CL_ABSTRACT_BASE
         true); // assign a no-meaning value to the placeholder var

} // anon namespace
// By convention, supermacros undefine all of their parameters after
// use so they can be called repeatedly without clients having to
// undef everything:
#undef CL_TYPE
#undef CL_BASE_TYPE
#undef CL_TYPE_NAME
#ifdef CL_ABSTRACT_BASE
# undef CL_ABSTRACT_BASE
#endif
```

You may want to look over that a few times to be sure of what it's doing. In short, it's doing the same thing as the `cl_CLASSLOADER_REGISTER` macros presented earlier, but it does so in a more generic way, which is useful for a wider variety of types. Note that there is no global include guard to prevent against multiple inclusion - in practice, supermacros generally do not use them (and, in fact, generally do not want them, as supermacros are designed to be included multiple times in succession, in the same way that classical macros often are). We do, however, block of one section of it from mulitple-inclusion.

That's essentially all there is to it. You may find that you want to partially specialize the placeholder type for special cases, but so far this has not proved to be necessary.

One major benefit of supermacros is that we can implement customized ones for special cases, keeping the same interface while swapping out the back end:

```
#define CL_TYPE SomeSpecialType
#define CL_TYPE_NAME ”SomeSpecialType”
#include ”special_registration.hpp”
```

Another property of supermacros is that if we decide to completely change the back-end implementation the client-side interface can remain the same, as it's ”parameters” are named, as opposed to positional.

3.5.3 Another approach to supporting abstract types

Let's briefly go back to the problem of the default second argument for `instantiator::register_xxx()` (as described above):

The supermacro code, as shown, uses `object_factory::no_instance` for abstract types. A better way to do this, which still allows us to keep the 2nd argument to `register_xxx()`, is to specialize `object_factory<CL_BASE_TYPE,CL_T`
such that `new_instance()` returns 0. We can easily do that from the supermacro code. This is in fact what we do in the P::Classes classloader.

3.6 Done! (Again!)

This concludes our coverage of registering types with the classloader. Keep in mind that the above is *one* potential solution to the factory registration problem. Implementors are encouraged to experiment.

4 DLLs

One of the primary benefits gained by using a classloader is the ability to load types which are not known to the main application when it is compiled. The application must know about common base types, but need not know about concrete implementations. In fact, the implementations classes need not even exist when the application is compiled, and they can still be loaded dynamically later on. Sound useful? You bet it is. Let's see how we might go about doing it...

4.1 Compiling and linking with `dlopen()`

The function `dlopen()` is normally part of the `libdl` library, which comes preinstalled on most Unix-like systems. The GNU project has a clone of `libdl`, called `libltdl`[11], but it's interface is identical to `dlopen()`, so readers who use that may simply mentally substitute `lt_dlopen()` for `dlopen()`. In fact, any function which can open a DLL can be substituted for `dlopen()` for our purposes, such was Win32's `LoadModule()`.

`dlopen()` is normally defined in the header file `dlfcn.h` and `lt_dlopen()` is defined in `ltdl.h`. Client code using these functions must of course include the appropriate headers. When linking an application or DLL which is to make use of our framework we must take care to heed the `dlopen()` documentation and link our application (or DLL) with the `-export-dynamic` flag (also called `-rdynamic` on some platforms, including Linux). If we don't do this the end result is that the class registrations will not take place (please don't ask me why - i don't understand the internals of `dlopen()`!).

It may (or may not) be necessary to initialize `dlopen()` with a call like this, one time from `main()`, or before `main()` (via using the static initialization trick):

[11] Many thanks to Roger Leigh for bringing `lt_dlopen()` to my attention.

```
    dlopen( 0, RTLD_NOW | RTLD_GLOBAL );
```

The `libltdl` equivalent is:

```
    lt_dlinit();
    lt_dlopen(0);
```

This ”opens” the main application. In theory this may not be necessary (any longer), but i have honestly been too lazy to test it, so including this is the safe thing to do.

4.2 Opening DLLs

This is so simple to do, we're going to jump right in:

```
    void * so_handle = dlopen( ”/path/to/my.so”, RTLD_NOW | RTLD_GLOBAL );
```

The variable `so_handle` is an opaque resource handle, similar in use to C's classic `FILE` type, which can be tracked by client code but need not be. It is intended to be passed, later on, to `dlclose()`, but *this should not be done*! Calling `dlclose()` can cause a DLL to be closed when it is in use by another piece of code, and very often results in a segfault. It is far, far more convenient (and far safer) to simply let the OS close the DLL when the application shuts down through it's normal exit routines. If you decide to be utterly pedantic and call `dlclose()`, heed these words: *You Have Been Warned.*

Thus we're going to ignore `so_handle`, except that it tells us one very useful thing: if it is `0` then `dlopen()` either couldn't find the DLL or opening it failed (which can happen for a number of reasons). Clients may call `dlerror()` (or equivalent) to get a string-form error message when `dlopen()` has failed.

Assuming that the handle is valid, we can ignore it and move on to our next steps.

So now how do we farm our classes from the DLL? The answer is so simple that i am almost ashamed to write it:

We don't!

Remember all of that registration mess we went through earlier? *That* code registered all classes contained in `my.so` in the instant that we opened the DLL, before the call to `dlopen()` even returned.

But how?!?

When the DLL is opened, the runtime environment initializes all static, namespace-/global-scope variables. When this happens, we have arranged for a call to `instantiator::register_factory()` to be triggered, feeding the registration into our classloader - the one running in the main application. This is one of the reasons that `instantiator` uses static functions and a shared factory map: so that the registrations feed back into the same factory pool which the rest of the application is using.

It is this property of our model which ensures that we have 100% type-safety, even for those types loaded via DLLs. Consider: a registration will never map a factory for the wrong base type into a given classloader - it *can't*, as doing so would not compile[12]. Thus, when a registration happens we know it is valid and are ensured that it goes to the proper classloader.

One implication of this general approach is that opening a DLL containing N classes will register all of those classes with their appropriate classloaders. This allows, e.g., easy bundling of several classes into a single DLL. Note also, however, that this means that classes living in DLLs may be register with classloaders which the main app never uses - effectively a waste of resources, though a rather small one.

This all may seem to be rather brute-force, backwards, or perhaps even anarchistic, but it is effective.

[12]We will ignore the possibility of someone writing ”malicious” factories or registrations using casts to fool the compiler.

4.3 When to search for a DLL

The technique used by the `cllite` framework when it is asked to classload a type follows:

1. If the factory map has a factory for the given lookup key, return what that factory gives us, else continue...
2. Try to open a DLL matching the name of the class. (Techniques for doing file searches are of course implementation specific.) If this fails, return 0 (i.e., class could not be found), else continue...
3. Go back to step 1, but if it fails, instead of continuing, return 0 (i.e., class not found) or throw an exception. Reasoning: if Step 2 works then *any* classloader registrations which live inside the DLL *registered themselves with our factory map*, so a re-check of Step 1 will tell us if the DLL contained the sought-after class or not.

We could potentially go through a number of loops of these steps by, e.g., iterating through different classname-to-DLL-file algorithms.

Implementors will have to address design questions such as, "when a factory lookup fails, do we flag it as bad and never do a lookup on that key again, or do we keep trying on each request?" and "how do we translate a class name into a DLL name?" The answers are necesarrily implementation specific.

`cllite` includes support for passing a functor to `classload()`, where the functor is responsible for translating a class name string to a DLL name (e.g., "`MyClass`" to "`myclass.so`"), and includes a simple implementation for a name translator, plus fully configurable search path support. Discussing these details are out of scope here, but curious readers are encouraged to examine the `cllite.hpp` header file available in that project's source tree. In fact, `cllite` provides a complete client-side interface based off of a model very similar to the one presented in this paper, and clients may wish to use it as a basis for designing their own.

4.4 Done!

That's all there is to loading classes from DLLs using the anonymous namespace/template magic trick.

While the `dlopen()` interface may be too primitive, or too platform-specific, for use in general client code, the above shows us everything we need in order to take advantage of it. In the closing section we will cover some potential simplifications which can be used to hide clients from having to use `dlopen()`.

5 In conclusion...

Now, that wasn't all that hard, was it?

We have not developed a whole framework here, but we have seen a back-end which provides everything we need to add classloading support to our applications. We will end this paper with some suggestions for improvements, both to make the interface easier to use and to isolate clients from platform-specific details such as called `dlopen()`.

5.1 Potential TODOs

5.1.1 Simplify the classloader interface.

The classloading front end should not intrude on client code, and should probably have an interface which looks something like:

```
template <typename T>
T * classload( const std::string & classname );
```

That function can do whatever it needs to find DLLs, e.g., using a configurable "classpath". On error, perhaps it returns 0 and perhaps it throws an exception: those are design desicions the implementor must answer[13].

A set of functions for registering factories might not be a bad idea, either. Ideally, factory-related classes like our examples, `instantiator` and `object_factory`, are safely tucked away out of client-side view. They are, after all, implementation details.

[13]Shameless plug: in `cllite` this is configurable: clients can toggle the use of exceptions at any time during runtime.

5.1.2 Hiding classloading from downstream clients

It is easy to hide the general classloading layer from downstream client code by inserting a small translation API, presumably one more suited to the specific application. For example, instead of requiring client code to do:

```
Document * d = classload<Document>( ”DefaultDocumentClass” );
```

He could instead call:

```
Document * d = createNewDocument( ”DefaultDocumentClass” );
```

or even:

```
Document * d = Document::create( ”DefaultDocumentClass” );
```

This allows the back-end to change considerably without affecting clients of the `Document` interface. It also allows, e.g., the insertion of GUI-based dialog boxes on classload errors, or similar application-specific features. From a design perspective, clients of `Document` shouldn't need to know how to classload `Documents`, nor should they even need to know that the requested `Documents` are even dynamically loaded.

5.2 That's all, folks!

This concludes our discussion of implementing a classloader in C++. If you have any feedback, questions, or suggestions for improvements, please feel free to get in touch with me at the address shown at the top of this document. i am always more than happy to address readers' emails, and always tickled pink (and very flattered) to hear that someone actually gets some use out of my papers.

—– stephan beal

21 August 2004