Generic Cleanup in C++

stephan@s11n.net

September 28, 2005

Abstract

This article develops a project-neutral approach to cleaning up standard containers in C++. We will show how the cleanup of any container can be reduced to a single API call, no matter how deeply nested the container is, nor how many subcontainers it may contain, nor with client-side regard to whether the contained items are pointer- or value-types. This opens up the door to providing exception-safe, pointer-holding containers without the use of smart pointers. The approach is suitable for a wide variety of types, but we will demonstrate with standard containers because those are a particular problem when it comes to cleaning up unmanaged pointers in the face of exceptions.

Revision History:
10 July 2005: initial revision

Contents

1 Introduction

1.1 Preliminaries

This article assumes much prior knowledge regarding C++. In particular, we will make many assumptions about your understanding of how *template specializations* and *partial template specializations* work. If these

terms mean nothing to you, this article isn't likely to, either, but you can certainly find some good tutorials on the topic on the net, or in the classic book *C++ Templates, the Complete Guide*, by Nicolai Josuttis.

Just as we will make many assumptions about *your* knowledge, we're also going to make some about mine: in particular the knowledge that i am often mistaken! If you find a bug in this article or the accompanying source code, please report it. Any and all feedback are welcomed, and your feedback may be used to improve future versions of this article and the source code. You can reach me via email:

`stephan@s11n.net`

The home page for this paper is:

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

This article is dedicated to martin f. krafft (http://libfac.sourceforge.net) and Christian Prochnow (http://www.pclasses.com) both of whom provided very influential feedback while i was working through libs11n's exception handling rules. That work was the main inspiration for this article.

1.2 What is this all about?

Have you ever had a container like this:

```
map<int,T*>
```

or this:

```
list< map<int,T*> >
```

or even this:

```
deque<list<multimap<int,T*>*>*>
```

???

You probably have, and you are probably well aware the fact that in none of those cases are the pointers owned by their containers. What does that mean? It means, if we destroy the containers with transfering ownership of the contained pointers, the pointers will *leak*.

There are several approaches to handling this type of cleanup:

- Manually walk the container, or use `std::for_each()` and appropriate functors, and delete the pointers as you go. This requires knowing the structure of the contained elements, and probably requires having some idea of their type(s), and thus normally requires at least some small amount of hand-written, container-specific code per cleanup operation.
- Use proxy objects to manage access to your containers, such that the proxies clean up the containers when the proxies are destroyed.
- Fundamantally the same as proxying, write your own containers which manage pointers. It is not uncommon to see a `PtrList` class template in utility libraries.
- Use smart pointers, so that when a container holding the pointers is destroyed, the pointers actually clean up themselves. The only significant down-side to this approach is that it imposes a specific smart pointer implementation on the client. As there isn't yet a standardized smart pointer implementation, the choice of implementation is still very much a personal issue, and not one i feel is worth imposing on users of a given library unless the smart pointers plays a significant role in the encompassing library (e.g., as in Boost [www.boost.org]). While we cannot discount smart pointers as a viable solution, we will not consider smart pointers here.

There are certainly other approaches, but those are the ones i can think about off the top of my head.

We're going to try a different route. The goal is, in effect, having a single function with which we can clean up arbitrary containtainers, irrespective of nesting level, pointerness off the contained types, and the actual contained types themselves (provided they are compatible with our framework).

1.3 Motivating problem

The original motivation for the API we will develop here came about while working on libs11n 1.1.3. While experimenting with the exception conventions, i realized that there were exception/failure cases in which the library would leak *sometimes*. The behaviour was predictable and definable, but entirely dependent on what templatized types were being used. The fundamental problem was, for the places at which this happened there was literally no logical course of action the code could take. The only thing it could do was call delete and hope for the best. And that *works* in many cases... but not in the case of containers holding pointers, or nested containers where one of the sub-containers holds unmanaged pointers. i unfortunately failed to see that early on, as i didn't fully consider the full implications of deleting any given type. At that level of the library failures rarely happen (they happen up-stream, in i/o), so the bug went unnoticed until a full review of the core source code a long time (a bit over a year) after the code was introduced into the library.

In any case, this was completely unacceptable behaviour on the library's part, and needed to be fixed. So, i sat down to implement an idea i'd been tossing around for almost a year, and had tried experimentally in another tree at one point. This paper covers that approach, and shows that it can significantly simplify the cleanup of not only containers, but arbitrary types which hold unmanaged pointers. We will develop a small library which provides such support to arbitrary client code.

To provide a concrete example let's go over the case in the s11n library so we can make it clear why we needed something more flexible than `delete()`. Rather that use the s11n code for demonstration, which would require a significant deal more background knowledge than necessary for our purposes, we will abstract it a bit.

Assume we have the following free function:

```
template <typename T> T * create_object( const data_source & src );
```

Assume that `data_source` is a DOM-like container, but that's not at all important for our purposes. i point it out only to explain why the function argument is const (whereas an input *stream* would not be const).

This function's purpose is to create an object, deserialize it using the given data source, and give it back to the user. How it does this is unimportant for purposes of solving the cleanup problem, so we won't go into that level of detail.

The conventions of the function are that on succcess it returns a non-null pointer (which the caller owns), and on error it returns 0 (or may propagate an exception from elsewhere).

It performs the following operations:

1. Try to create a `T` object. If that fails, we can safely bail out with no chance of a leak. (This *can* fail because we use a classloader to load new types, polymorphically if needed.)
2. Use some algorithm to populate the object from `src` (i.e., to "deserialize" it).
3. If the operation succeeds, return the new object, else...
4. `T` is in an undefined state - we need to destroy it. ***How to safely destroy this object is the sole topic of this article!***

The problem is, that approach will work fine for any `T` which does not contain "unmanaged pointers" - pointers nobody (yet) owns, in the sense of "who's going to delete them?" Once `T` is a `container<X*>`, that logic breaks down considerably. What we need is a way to walk containers without having to know their types nor underlying structure, such that we can deallocate any such pointers, even in the case of nested containers.

This paper explains how we can satisfactorily solve this problem for any `T` which meets a minimal set of requirements. Namely:

1. If `T` is a pointer-qualified type, this code must be legal: `delete anInstanceOfT;`
2. The destructor must not throw. This is a general C++ guideline, and types with destructors that throw are categorically forbidden from use with the STL [CCS2005].

That's it, really, as far as concrete requirements go. Some types, namely containers, will have some "indirect" requirements, and we will show how to accommodate those as our framework is fleshed out.

1.4 Source code

The complete source code developed for this article should be available at the article's home page:

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

The source code should build on any C++ compiler supporting partial template specialization. Compilers without this feature are implicitely not supported, because this feature is (in my opinion) essential to solving this problem satisfactorily.

2 Constructing the implementation

Now let's build a library capable of handling our motivating problem...

2.1 A client-side API

First let's lay down our "client API" - the public interface which serves as the core entry point into our eventual functionality. Let's try:

```
template <typename T> void cleanup( T & obj ) throw();
```

The job of the function is to "clean up" the object, with the exact definition of "clean up" being left a bit hazy because it is inherently type-specific. In brief "clean up" essentially means "delete pointers," but might also include type-specific behaviours. We will see examples of this soon, so don't worry too much about these details just yet.

The `throw()` (i.e., throws no exceptions) qualification would seem to be justified. As throwing exceptions from destructors is normally considered a bad idea in C++, by extension we can conclude that `cleanup()`'s logical role in the destruction process warrants the same guaranty. This justification is admittedly philosophical in nature, so implementors should feel free to change the throw specification to suit them. [A few days after writing this i bought a copy of [CCS2005], and its Item 51 seems to back up this decision.]

We're going to jump the gun just a small bit: as it turns out, it simplifies some of our algorithms later if we have the following overload for our cleanup function:

```
template <typename T> void cleanup( T * & obj ) throw();
```

The difference between this and the first form is that this one deletes the object and assigns it to zero after passing the call on to `cleanup<T>(*obj)`. Why we want this will become clear later on. The assignment to zero is not mandatory, but seems reasonable and helps us test the code for proper functionality. For example, the following code demonstrates the effect of the second form:

```
T * t = new T;
cleanup<T>( t );
assert( 0 == t );
```

The assertion will pass if all has gone well (and if `NDEBUG` isn't defined, which disables `assert()`).

With those two functions, we have the *complete public API* for the functionality we need. What we need now is some way of translating specific requirements for specific types into calls to different handlers. For our purposes, templates fill this role nicely, so we will pursue a solution based upon templates and "compile-time polymorphism."

2.2 `cleanup_traits<T>`

Now we jump to the "middle part" of the problem and define a traits type. The type has only one purpose: to map a given `T` to a set of rules (a functor) which knows how to clean up a `T` object. The type looks something like this:

```
template <typename T> cleanup_traits {
```

```
    typedef some_functor cleanup_functor; // cleanup rules for T
    typedef T cleaned_type; // for use with algos/functors
};
```

We will use the `cleanup_traits` type to translate calls to `cleanup<T>()` through the proper (installed) functor.

Above i said this was the "middle" of the problem. Let's see how we can connect this part with the first part, our public API. Here are potential implementations of our `cleanup()` functions:

```
template <typename T>
void cleanup( T & t ) {
    typedef typename cleanup_traits<T>::cleanup_functor CF;
    CF cf;
    cf(t);
}
template <typename T>
void cleanup( T * & t ) {
    cleanup(*t);
    delete t;
    t = 0;
}
```

Though the first variant can be implemented as one long line, i have broken it down into smaller steps, first for clarity, and secondly because some compilers don't appear to like:

```
typename cleanup_traits<T>::cleanup_functor()(t);
```

Regarding the second form: if you aren't familiar with the reference-to-pointer syntax, don't be alarmed. While odd-looking, it is perfectly valid and allows us to do some things to a pointer which we cannot do to a pointer passed in to a function, like assign it to zero.

Before we go on, let's make one highly arguable addition which eases my mind a bit:

```
template <typename T> cleanup_traits<T*> :  public cleanup_traits<T> {};
```

i hope to be able to explain/justify this fully at some point. The main implication of it is that `cleanup_traits<T*>::cleaned` does not have a pointer qualifier. This simplifies some algorithm code later on, but is otherwise not essential to the framework.

If your project already uses some sort of traits type for storing type information, you might consider adding `cleanup_functor` to your existing traits type, rather using `cleanup_traits`. Whether this is appropriate or not depends on your project and the scope of your traits type.

2.3 `cleanup_functor`

Remember that `some_functor` type we declared in `cleanup_traits`? Well, we need to define it. In fact, we need a default implementation we can specify in the base `cleanup_traits` definition. As it turns out, a reasonable implementation does exist for arbitrary types:

- For pointer types, delete them.
- For reference/value types, *do nothing*. Let the normal destruction of stack-allocated objects do its thing.

That is a bit oversimplified, but that's essentially what it boils down to. Note that we have shifted the pointer-handling into `cleanup(T*&)`, so the specific cleanup functors do not know whether the object they are cleaning up is itself a pointer or a reference.

Here is what the default implementation of the cleanup functor looks like:

```
template <typename T>
default_cleanup_functor {
    typedef T cleaned_type;
    void operator()( cleaned_type & ) const {
        // NOTHING!
    }
};
```

Why on earth do we want to do *nothing* there? Because we cannot apply any given set of rules to a reference of any given type, so the default rule (i.e., the default implementation) is to do nothing. Before moving on, let's show that this is really the behaviour we want via examining a function like the one in our motivating example:

```
template <typename T>
T * create_object( SomeType input ) {
    T * t = new T;
    if( !  restore_state(input,*t) ) {
        cleanup( t ); // t is deleted and assigned 0
    }
    return t;
}
```

(Note that we have a potential leak in the case of an exception, but we will cover that later on.)

Let's mentally substitute some various types for `T` and verify what `cleanup(t)` does:

- `T` is a POD type: `t` is cleaned up (a no-op) then deleted.
- `T` is a client-side type: t is cleaned up (no-op unless special `cleanup_traits<T>::cleanup_functor` defined) and deleted. This is *normally* correct for client-side classes.
- `T` is a container: this is where we need to take care. Read on...

Now, if we're comfortable with the conventions we've laid out so far (they seem reasonable enough to me), we are actually done with the first and second layers of the framework. The final layer is in the type-specific cleanup rules.

What we need now is to install rules for specific containers, which should walk the containers and call `cleanup()` on each item. This can be done in one of two ways:

1. Specialize, or partially specialize, `cleanup_traits<T>` for the container type.
2. Specialize, or partially specialize, `default_cleanup_functor<T>` for the container type. (This is why the template parameter for the default functor is specified at the class level, not function level.)

The approaches are equivalent, and which you use is probably a question of taste and existing project conventions (if any).

2.4 Cleaning up a `list<T>`

Given our core API and a default cleanup functor, we fundamentally have everything we need to clean up nearly any structure. As the main motivation for this article is standard containers, let's start with a simple one: `list<T>`, where `T` may optionally be pointer-qualified.

Above we listed two ways to install rules for a type with the core framework. For this example we will specialize the default functor, though this approach is fundamentally no different than specializing `cleanup_traits` and specifying a different functor.

Before we start, let's jump a bit ahead (again) and write a small functor which we will use very soon to simplify the list-walking code:

```
struct do_cleanup {

    template <typename T> void operator()( T & t ) throw() {
        cleanup( t );
    }
    template <typename T> void operator()( T * & t ) throw() {
        cleanup<T>( t );
    }

};
```

Now let's fix `cleanup<list<T*>>()` so that it works properly:

```
template <typename VT>
struct default_cleanup_functor< std::list< VT > > {

    typedef std::list< VT > cleaned_type;
    void operator()( cleaned_type & c ) throw() {
        std::for_each( c.begin(), c.end(), do_cleanup() );
        c.clear();
    }

};
```

What we've done is actually ensured two things: that both `cleanup<list<T*>>()` and `cleanup<list<T>>()` will work as expected for any `T` which has a valid cleanup functor installed.

The above specialization of the default template works for *all* standard list-like containers, not just `std::list`: the only thing which needs to be changed is the `std::list` text. This includes `vector`, `deque`, `set` and `multiset`, plus your own types which are conventions-compatible with those.

Now let's look again at the behaviour of this call:

```
typedef list<T *> ListT;
ListT list;
...  populate list ...
cleanup( list );
```

Let's assume `T` is:

- a POD: each will be cleaned up (a no-op) and then deleted.
- `list<int>`: each sublist will be recusively walked and cleaned up, then deleted.
- `list<list<list<X*>*>*>`: same as above. So far so good.
- `list<map<int,X*>>`: the pointers in the map will be leaked.

We know how to fix that last case, so let's do it...

2.5 Cleaning up a `map<K,V>`

Clearing a map is almost like cleaning a list. There is one glaring problem, however: the keys of maps are constant objects. In short, this means we cannot apply cleanup rules to them without violating their constness. Given this, and the rarity of using unmanaged pointers as keys in maps, we will chicken out and declare that map *keys* are not cleaned up by our rules.

Here's what a cleanup functor for all standard maps might look:

```
template <typename KT, typename MT>
struct default_cleanup_functor< std::map< KT, MT > > {
```

```
    typedef std::map< KT, MT > cleaned_type;
    void operator()( cleaned_type & c ) throw() {
        typedef typename cleaned_type::iterator IT;
        IT b = c.begin();
        IT e = c.end();
        if( e == b ) return;
        typedef typename cleanup_traits<MT>::cleaned_type MTBase; // stripping
        pointer part
        for( ; e != b; ++b ) {
            cleanup<MTBase>( (*b).second ); // this is why we wanted to strip
            any pointer part
        }
        c.clear();
    }
};
```

As for the list-based algorithm, this exact same code will also work with multimaps: simply replace `std::map` with `std::multimap`. It is also ignorant of the pointerness of the contained types: they are handled identically regardless of which they are. The only difference in pointer-vs-reference handling is in `cleanup()`, where the pointer-based overload will delete the pointers, whereas stack-allocated objects will be destroyed in the call to `c.clear()`.

Now let's go look again at the cleanup of our infamous `list<T>` when `T` is a map type. When the list is cleaned up, the map will be walked and any pointers in the "value part" of the map are freed (again, keys are not because they are const). So the following will work as expected:

```
typedef map< int, list< map < string, X *> *> > MapT;
MapT map;
...  populate map ...
cleanup( map );
```

Through the recursive application of the `cleanup()` algorithms, the containers are each walked and any pointer entries deleted. Non-pointer entries are either skipped (via the no-op default functor) or walked (if containers), but not deleted (which they can't be). Any stack-allocated objects will be destroyed either by their container going out of scope or via an explicit call to `clear()` in the cleanup functor.

It might be desireable to use a template metaprogramming technique to emit a warning, or even throw an exception, if the key part of the map is a pointer. Remember that throwing is likely to cause the program to abort, because `cleanup()` is declared as no-throw. This may very well be preferred over a leak of those pointers, however. For the very brave, feel free to cast away the constness and clean up the keys - my respect for constness prohibits me from doing so.

2.6 Protecting against leaks during exceptions

Given our above, API, we have all that we really need to protect against a leak in our motivating example. Let's look at it again:

```
template <typename T>

T * create_object( SomeType input ) {
    T * t = new T;
    try {
        if( !  restore_state(input,*t) ) {
            cleanup( t ); // t is deleted and assigned 0
        }
    catch(...)  {
        cleanup( t );
```

```
        }
        return t;
    }
```

While sufficient, it's a bit ugly. As it turns out, we can simplify the implementation with the use of a `std::auto_ptr`-like type. The source code distributed with this paper contains a `cleanup_ptr` class template, which is used like a `std::auto_ptr` but is intended specifically for handling cases covered by our cleanup framework. An example:

```
cleanup_ptr<T> c( new T );
if( operation fails ) return error or throw; // c will clean up the object
return c.release(); // transfer ownership of object to caller
```

With that simple mechanism in place we can simplify cleanup during exception/error handling significantly and, in our case, provide some leak-safety guarantees which simply couldn't be made without this, or a similar, feature.

3 Wrapping up

The previous section showed us everything we need to know to apply type-specific cleanup rules using a trivial framework. Let's leave with a few parting notes...

3.1 Re-examining the motivating problem

Let's take another look at the motivating problem described at the top of this article, and show how our cleanup framework approach allows the algorithm to safely recover from errors, instead of "sometimes" admitting a leak. In that case, we had the following function:

```
template <typename T> T * create_object( const data_source & src );
```

After the creation of the cleanup framework it can predictably, reliably destroy nested objects of near-arbitrary types. It now performs the following operations:

1. Try to create a `T` object. If that fails, we can safely bail out with no chance of a leak.
2. Pass `src` to the new object so the object can populate itself.
3. If the operation succeeds, return the new object, else...
4. The object might be in an undefined state: `cleanup(obj)`

Using the `cleanup_ptr<>` mentioned above, the error-handling code becomes trivial to write.

3.2 Simplifying creation of cleanup functors or `cleanup_traits`

One feature which would certainly simplify using the library is to allow the creation of cleanup functor specializations, or partial specializations, via macros. The libs11n code uses this approach to create partial specializations for the standard containers.

3.3 It ain't just for containers...

The model shown here works not only for containers. Containers are an important consideration, indeed the motivating consideration, for the framework, but it can also be used for other purposes. The original prototype for this code was used to clean up items from an underlying database-like store. Types which participated in the db called the cleanup functor from their dtor, passing it their unique db identifier (instead of their pointer/reference). The functor then removed any data associated with that instance of that type from the db.

3.4 What about cleaning up `void` pointers and arrays?

This article has specifically avoided the handling of `void` pointers and arrays during cleanup because, quite frankly, i never use them. They are artefacts from C, and don't have a place in most modern C++ code. i am also not certain of the implications of generically freeing a `void` pointer: should we use `free()` or `delete`? The `std::vector` class is compatible with C arrays and superior in every way (except that it's a tiny bit larger than a raw array), so there is no reason not to switch from arrays to vectors.

3.5 Ciao!

Thanks for taking the time to read this article. :)
—— stephan@s11n.net

References

References

[CCS2005] *C++ Coding Standards*, Herb Sutter and Andrei Alexandrescu, 2005