Document Number: SC22/WG21/N1681
J16/04-0121

Date: 01 September, 2004

Project: JTC1.22.32
Programming Language C++
Library Working Group

Reference: ISO/IEC IS 14882:1998(E)

Reply To: David B. Held
CodeLogic, LLC
110 10th Ave. N
St. Cloud, MN 56303 USA
dheld@codelogicconsulting.com

A Proposal to Add a Policy-Based Smart Pointer Framework to the Standard Library

I. Motivation

Smart pointers provide an automated memory management solution that aids in preventing resource leaks and ensuring exception safety. Further motivation and justification for smart pointers in the Standard Library can be found in N1450=03-0033 [DDC03]. Said document argues that shared ownership semantics are the most common need among smart pointer users. This proposal amplifies that notion with the insight that this goal may be achieved through multiple implementation designs, each offering various tradeoffs. Furthermore, this proposal will present a coherent framework by which most useful smart pointer designs may be assembled with as little effort as possible. While the proposal is based on an existing reference implementation, that implementation is not believed to be widely used, as it has not been officially released. On the other hand, the reference implementation is based on an earlier library that has been released and has been in use for about three years.

Multiple Roads to Shared Ownership

There are numerous strategies for implementing smart pointers with shared-ownership semantics, and even the `Boost.SmartPtr` library [CD99] from which `std::tr1::shared_ptr<>` was selected contains both an externally counted and an intrusively counted pointer. While `shared_ptr<>` tends to get most of the attention, `boost::intrusive_ptr<>` definitely has some users, judging by the occasional questions and comments on the Boost mailing list. Furthermore, the Microsoft® COM specification provides a large set of intrusively counted objects for which the safest, most efficient, and most natural usage would call for a wrapper of `IUnknown::AddRef()` and `IUnknown::Release()`. Finally, the OMG CORBA® architecture also provides numerous intrusively counted types that could benefit from a smart pointer wrapper.

Somewhat surprisingly, it seems that reference linking may, in fact, be one of the fastest non-intrusive ownership-sharing strategies for smart pointers in a single-threaded context. Tests conducted for a *C/C++ Users Journal* article [AH04] indicate that the minimization of conditional branches led to fairly fast code, if not for optimal pointer size. A scenario in which speed is strongly favored over size and the solution cannot be intrusive may benefit from a reference-linked implementation.

It is inevitable that some designs will result in pointer cycles, which a naïve implementation will fail to free properly. The current `Boost.SmartPtr` library solves this problem by introducing `boost::weak_ptr<>`, which breaks cycles. While this is an appropriate solution, it requires developers to explicitly track cycle-forming scenarios and use `weak_ptr<>`s in the appropriate places, which may be an onerous process in large projects. Thus, some developers, such as Greg Colvin [Colv99] and Larry Evans [Evan04], have developed smart pointers which detect cycles and clean them up correctly.

The number of shared ownership strategies is overwhelming, and the subject could fill an entire paper all by itself. But perhaps it is worth noting that there are gray areas between totally intrusive reference counting and purely external reference counting. Phillipe Bouchard explored a point in this region of the design space with his `shifted_ptr<>`[1] [Bou03], which was anticipated by an earlier Boost effort to evaluate various ownership techniques[2].

Non-Shared Ownership

While shared ownership may be the most common need, one can hardly argue that other ownership semantics are not used regularly, given the existence of `std::auto_ptr<>`. Furthermore, while `boost::scoped_ptr<>` was left out of N1450=03-0033, it is clear that it, too, has many users, given the somewhat frequent requests to modify it in various ways (such as by adding a custom deleter [Rame04]). This strategy might be called "no-copy semantics". The frequent suggestions to improve `std::auto_ptr<>` itself has led to better designs and implementations which utilize what we now call move semantics (c.f.: N1377=02-0035 [HDA02]). And occasionally, there is a request for a smart pointer with deep copy semantics[3].

Smart Resource?

Besides pointers to objects and pointers to arrays, it has been noted that a policy-based framework could generate arbitrary resource wrappers for everything from `FILE*` to sockets to Win32® handles. As an exercise, Held wrote a smart lock wrapper for a mutex. These would all be instances of custom storage policies.

[1] In this design, the reference count is non-intrusively grafted onto the pointee, and the pointer is "shifted" to make it look like an otherwise normal intrusive implementation.
[2] Check the History and Acknowledgements section of [CD99], especially the "Placement attached" configuration.
[3] Of course, Alan Griffith's [Grif99] `grin_ptr<>` has been in existence for quite some time. It can implement deep copy semantics and its motivating use-case is the pImpl idiom.

Checking For Null

C++ programmers must constantly make trade-offs between speed and safety. While safety is generally preferred, there are enough times when performance is absolutely necessary that it would be burdensome to write a custom smart pointer (or family of smart pointers!) that has all the safety features removed. For a monolithic design like `std::tr1::shared_ptr<>`, which already does a lot of work, it would probably not be appropriate to create a stripped-down version with all the non-essential checking removed. But for smaller, lightweight pointers that may get created thousands of times and accessed many times more than that, it makes sense to have a version that does not contain any safety features, once it has been shown to be correct. Furthermore, some programmers will refuse to switch from raw pointers to smart pointers if they perceive any significant loss of performance (whether this position is warranted or not). Demonstrating smart pointers that perform favorably compared to raw pointers is an important step in encouraging such programmers to adopt more modern techniques. Alexandrescu and Held [AH04] showed that dereference checking adds about a 40% performance penalty over raw pointers, while checking-free smart pointers perform identically to raw pointers on dereference. Both options should be available, and in a policy-based framework, they are. Not only is there a choice between checking or not, but one may also choose whether checking is reported via assertion, exception, or some other mechanism. Finally, there are different strategies for checking which result in different performance payoffs. Only checking for null during construction and copying avoids the somewhat costly dereference check, but prevents the default construction of pointers. Whereas, checking for null on every access is the safest approach, which is why it is the default checking policy.

On a more philosophical note, Alexandrescu comments [in private communication] that: "…it is in the grand tradition of C++ to offer efficient components for building safe components (the converse being impossible)... By trying to amass all of the useful features in one, `shared_ptr<>` inevitably fails that tradition. `smart_ptr<>` offers a range of flexible designs, with the safe ones building on the fast ones."

Implicit Conversion

While it is generally agreed that implicit conversion to the raw pointer type is a bad idea for smart pointers, some programmers will insist that any replacement for a raw pointer must interact with raw pointers seamlessly, and that means no calls to functions like `get()`[4]. Since this is an unsafe alternative, users must choose it explicitly with the `allow_conversion` policy.

Combinatorics and Boilerplate

A quick review of custom-written smart pointers will reveal a syntactic and semantic regularity that is awkward in its redundancy and ease with which implementation errors are introduced. It is inevitable that some functions (like `operator->()`) will be written in a similar manner. It is equally inevitable that some programmers will make subtle mistakes while designing such smart pointers. While a policy-based framework cannot eliminate all such mistakes, it can help to reduce them by allowing programmers to reuse portions that are already known to be correct. This reduces the burden of writing custom smart pointers in the way that the `Boost.Iterator` [ASW03] library reduces the burden of writing custom iterators and the way that the

[4] Or, as Alexandrescu suggests, "`leak()`".

Boost.Operators [Abra99] library reduces the burden of writing custom operators. Some functionality, such as comparison operators and casting operators, is almost always identical across configurations, thus saving the user considerable time in designing a new pointer type.

Furthermore, users can leverage work done by others by combining existing policies with their own custom ones, resulting in a combinatorial number of pointer configurations, rather than a linear selection. For example, one new checking policy can often be used with all the existing storage and ownership policies, multiplying the number of new pointer configurations available from which to select appropriate size, speed, and safety tradeoffs. Even if a custom ownership policy requires a specific custom storage policy, both can probably still benefit from the large number of checking policies available.

Summary
Is a policy-based framework justified, given the complexity for both users and implementers? The arguments above, together with an ever-increasing user demand for refined, specialized smart pointers, seem to support this conjecture. The steady stream of proposals for new smart pointer types and feature requests for existing ones on both the Boost mailing list and groups like comp.lang.c++.moderated strongly suggest that no small fixed set of smart pointers will satisfy the C++ community. The committee can serve developers well by providing a framework that offers the most common smart pointer designs while accommodating and aiding those who wish to construct non-standard designs.

II. Impact on the Standard

This proposal is essentially a pure extension. It does not by necessity depend on any other library, standard or non-standard, but implementations will benefit from the availability of facilities such as those provided by Boost. In particular, the reference implementation uses `Boost.TypeTraits` [Madd01], `Boost.Utility` [Abra03], and `Boost.MPL` [GAW02]. No other library or proposal depends on this framework, at the time of writing. No core language changes are required, but this framework would benefit significantly from template aliases as proposed in N1489=03-0032[5] [SD03]. It would not benefit from automated forwarding constructors à la N1583=04-0023 [Glas04] as previously assumed. This proposal would either modify `<memory>` or add a new header: `<smartptr>` (for details see section V).

There is certainly overlap between this proposal and N1450=03-0033. It is reasonable to believe that the smart pointers specified in [DDC03] could be implemented in terms of the framework presented in this proposal. The committee must decide whether this approach is mandated or allowed. As proof of concept, earlier versions of `boost::shared_ptr<>`, `boost::weak_ptr<>` and `boost::intrusive_ptr<>` were, in fact, implemented as policies within an earlier version of the current framework. However, `boost::shared_ptr<>` enjoys near continuous development, and it is not practical to update the policy definition whenever there is a change in the corresponding design. Nonetheless, it is strongly believed that `std::tr1::shared_ptr<>` and `weak_ptr<>` can, in fact, be implemented purely as policies within the framework being

[5] In fact, the `Loki::SmartPtr` library from which this proposal is derived was a motivating example in N1406=02-0064 [Sutt02].

proposed here. For the record, it is known that a standalone implementation of `std::tr1::shared_ptr<>` can peacefully coexist in the same project with the policy-based framework being proposed.

III. Design

Overview

The design of the library is derived from that specified in *Modern C++ Design* (*MC++D*), by Alexandrescu [Alex01]. The essence of the framework is summarized by a class parameterized over four policies, as illustrated below:

```
template <
   typename T,
   class StoragePolicy = scalar_storage,
   class OwnershipPolicy = ref_counted,
   class ConversionPolicy = disallow_conversion,
   class CheckingPolicy = assert_check
>
class smart_ptr;
```

These policies provide the majority of the functionality and all of the data defined by the `smart_ptr<>` class. The defaults are chosen to coincide with the most common use-case, although there is more to say on that later. The design of the framework attempts to accommodate several criteria deemed to be very important:

1. **Size**
   The framework should not produce types that are unreasonably large. This means that for simple configurations requiring no external ownership-tracking mechanisms (such as a `std::auto_ptr<>`-like, `boost::scoped_ptr<>`-like, or intrusively counted configuration), `sizeof(smart_ptr<T>)` should equal `sizeof(T*)`. Other configurations that provide similar capability to established smart pointer designs should be no larger than those extant types. E.g.: a typical externally reference-counted `smart_ptr<T>` should have size no larger than `sizeof(T*) + sizeof(unsigned*)`.

2. **Speed**
   Types produced by the framework should be as fast as equivalent hand-rolled smart pointer designs. Thus, any non-essential features that may impact performance must be user-configurable via the policy mechanism.

3. **Exception Safety**
   The framework should offer reasonable exception safety guarantees given constraints and requirements on policy classes. At least the basic guarantee must be offered for all functions[6].

[6] As Dave Abrahams notes, this should go without saying for all libraries. However, the conspicuous absence of exception safety analysis for many libraries indicates that this point is worth stressing.

4. **Configurability**
   In some respects, the cost of this framework is high. To justify that cost, the framework should make it relatively easy to produce virtually any smart pointer configuration desired. Whether that goal can be realistically achieved or not remains an open question.

Policy Integration

There are two obvious ways to integrate the policies into the `smart_ptr<>` type. One is to use the multiple-inheritance approach implemented by the Loki library which accompanies *MC++D*:

```
template <
   typename T,
   class StoragePolicy, class OwnershipPolicy,
   class ConversionPolicy, class CheckingPolicy
>
class smart_ptr
 : public StoragePolicy, public OwnershipPolicy,
   public ConversionPolicy, public CheckingPolicy
{ ... };
```

The other is to use a chained-policy approach as illustrated below:

```
template <
   typename T,
   class StoragePolicy, class OwnershipPolicy,
   class ConversionPolicy, class CheckingPolicy
>
class smart_ptr
 : public
      CheckingPolicy<
         ConversionPolicy<
            OwnershipPolicy<
               StoragePolicy<T>
            >
         >
      >
{ ... };
```

Aggregation is not desirable as it does not allow policies to contribute to the public interface of `smart_ptr<>`[7]. For the same reason, private inheritance is not considered. Each of the designs mentioned above implies various tradeoffs in simplicity, ease of implementation, and the policy interface presented to users who wish to write their own policies.

Multiple Inheritance

The multiple inheritance approach was initially selected because of its conceptual simplicity and implied orthogonality of policies. However, it was later realized that the policies cannot quite be made entirely orthogonal, and some communication among them is necessitated by the demand

[7] For a justification of why policies should be able to contribute to the public interface of `smart_ptr<>`, consider `operator[]` for configurations which support array access (deemed by the Boost community to be important, à la `boost::scoped_array<>` and `boost::shared_array<>`).

for exception safety. Nonetheless, this was the preferred implementation approach until the following issues were identified.

Poor Support for the Empty Base-class Optimization

The so-called “Empty Base-class Optimization” (EBO) is absolutely necessary to achieve the size criterion. Unfortunately, many compilers tested during the development of this library failed to suitably apply the EBO. Since the size criterion is deemed to be extremely important (due to the desire for low abstraction overhead), a workaround was sought. Alexandrescu proposed a solution he called `OptionallyInherit<>`, which is essentially the dual of `boost::compressed_pair<>`. Held implemented it as `optimally_inherit<>`, and its use led to this implementation:

```
template <
   typename T,
   class StoragePolicy, class OwnershipPolicy,
   class ConversionPolicy, class CheckingPolicy
>
class smart_ptr
 : public optimally_inherit<
      optimally_inherit<StoragePolicy, OwnershipPolicy>::type,
      optimally_inherit<ConversionPolicy, CheckingPolicy>::type
   >::type
{ ... };
```

While this solution does solve the size problem for most compilers, it is unsatisfying for several reasons. First, the `optimally_inherit<>` mechanism must forward all constructors, as well as `swap()`. Note that this would not be alleviated by N1583=04-0023 because the forwarding copy constructors perform casting to base classes so that non-templated constructors in those base classes will be properly called[8]. Second, the technique relies on partial template specialization (PTS), which at the time of original development, was still not widely supported (including on some popular compilers). However, it is possible to implement the mechanism without PTS for non-conforming compilers. Third, it adds compilation overhead to an already template-heavy library. While the overhead may be small compared to other features in the framework (such as the policy adaptor), if the template is instantiated over many types (which is to be expected in a large project), this overhead begins to impose a measurable compilation burden.

Policy Interaction

In discussing issues regarding this framework on the Boost mailing list, it became clear that the library was not exception safe. That is, it was possible for the `smart_ptr<>` class to take ownership of a newly allocated resource and then abort construction via an exception (such as while allocating an external reference count), thus leaking the resource. The solution to this problem was an intermediate template class called `resource_manager<>` which provided the cleanup function originally found in `~smart_ptr()`. By insinuating this class in the right position within the inheritance lattice (while imposing some reasonable requirements on policy authors), the basic guarantee was achieved. Conveniently, the `resource_manager<>` mechanism replaced one of the `optimally_inherit<>` instances. However, the lessons learned from this

[8] For a detailed explanation of this issue see note 1 in section VI.

minor debacle are that there is a necessary ordering relation for the initialization of `OwnershipPolicy` and `StoragePolicy` and that coordination between the two is required to obtain proper destruction. That, in turn, implies that these policies are not completely orthogonal.

This conclusion is also consistent with the observation by many policy designers that some interaction between `OwnershipPolicy` and `StoragePolicy` is desirable for various configurations. This led some to question, even from the very beginning, whether these two policies should be independent at all, or should rather constitute one policy. An argument for keeping the policies independent is that there are currently two `StoragePolicies` provided by the library: `scalar_storage` and `array_storage`. If these were merged into the currently provided `OwnershipPolicies`, there would be $N \times M$ such policies, rather than $N + M$ policies as it is now (where N and M are the number of `StoragePolicies` and `OwnershipPolicies`).

Furthermore, it is easily imagined that the framework could be used to construct resource managers for types other than heap-allocated objects. Such a configuration could be implemented by creating yet another `StoragePolicy`. For instance, consider a wrapper for `FILE*`. It is not appropriate to call operator delete on such an object. One solution would be to write a custom `StoragePolicy` which calls `std::fclose()` at the appropriate time. Note that this wrapper would benefit from the various ownership strategies already available, including external reference counting (keep the file open until all users are done with it), scoped semantics (no copying of the file handle), or move semantics (the handle can move from its original scope, but must retain exactly one owner at all times).

However, it is possible to retain some orthogonality and preserve the separation of concerns that somewhat independent `StoragePolicies` and `OwnershipPolicies` achieve while also allowing some interaction between the two policies. This is rather elegantly obtained by the chained-policy architecture.

Linear Inheritance

Despite initial suggestions that the Curiously Recurring Template Pattern (CRTP) be applied instead of multiple inheritance, the final design does not actually use CRTP because it is not necessary and there is nothing useful[9] that the `smart_ptr<>` template can provide to the policies (or, at least, it is far less useful than it might at first appear). Thus, the current reference implementation specifies a design by which each policy is parameterized on and inherits from the policy above it within the inheritance lattice (except for the `StoragePolicy`, which is parameterized on `T`, the pointee type passed to `smart_ptr<>`). That leads to a lattice of this form:

[9] Such as dependent types.

```
storage_policy<T>
   ↑
ownership_policy
   ↑
checking_policy
   ↑
conversion_policy
   ↑
smart_ptr
```

This design is straightforward and should be easy for policy writers to understand. While this solution deals nicely with some of the practical problems arising from the multiple inheritance architecture, it has issues of its own that need to be addressed.

Benefits
Because only single inheritance is used, EBO is far more effective; and most compilers will easily produce types with minimal size. This allows us to dispense with the somewhat ungainly `optimally_inherit<>` infrastructure. Since `OwnershipPolicy` is now derived from `StoragePolicy`, it has complete access to `StoragePolicy`'s interface, which provides the amount of coupling that most policies need, while still maintaining the factorization that leads to minimal code redundancy. This design also precludes the need for the `resource_manager<>` helper template, and simplifies the cleanup process over the original multiple inheritance design[10]. In those respects, this appears to be an ideal configuration.

Drawbacks
This design does not come without costs, however. The most obvious is the need for forwarding constructors in every policy class. As mentioned in section VI, note 1, inherited forwarding constructors do not help us here because of the presence of templated constructors (which are necessary for allowing policies to obtain arbitrary initialization data). Preprocessor metaprogramming[11] can alleviate the problem somewhat by generating many of the forwarding constructors, but it remains to be seen whether policy writers will want to take advantage of such an arcane device when a text editor will solve the same problem in much less time. Other issues are related to the conspicuous absence of template aliases (such as proposed in [SD03]). These issues will be explained in detail in the next section, which will describe the current workaround for the lack of template aliases.

Type Generator

A common criticism of a policy-based framework with a large number of template parameters is that users are required to name all non-default parameters with every declaration. One can certainly define a `typedef` for a pointer to a specific type, like so:

```
typedef smart_ptr<X, no_copy, no_check> X_ptr;
```

[10] For details, see note 2 in section VI.
[11] Using a tool like the Boost.Preprocessor library [KM02]. The current reference implementation does not use Boost.PP because it would replace an unwieldy solution with a solution that is unwieldy in another dimension.

However, it would be much more convenient to be able to define an alias for a particular configuration that is still parameterized on the pointee type, as in:

```
template <typename T>
using scoped_ptr = smart_ptr<T, no_copy, no_check>;
```

Unfortunately, template aliases do not yet exist in the language. Thus, Alexandrescu [in private communication] suggested a type generator approach whereby the policy parameters are passed to an outer class and the actual pointer type is a nested type parameterized over the pointee type. This design results in the following syntax, allowing `typedef` to come to the rescue:

```
smart_ptr<no_copy, no_check>::to<X> p;

typedef smart_ptr<no_copy, no_check> scoped_ptr;
scoped_ptr::to<X> q;
```

While this syntax is not as elegant as that possible with template aliases, it was decided that it is superior to writing the entire policy list for each pointer or forcing the user to create typedefs for concrete pointer types.

Unfortunately, it also creates a set of its own problems due to the rules for template argument deduction in C++. Before we examine these issues, let us take a look at the current design:

```
template <
   class StoragePolicy, class OwnershipPolicy,
   class ConversionPolicy, class CheckingPolicy
>
class smart_ptr
{
   template <typename T>
   class to
    : public
         CheckingPolicy<
            ConversionPolicy<
               OwnershipPolicy<
                  StoragePolicy<T>
               >
            >
         >
   { ... };
};
```

Here it is apparent that `smart_ptr::to<>` does the real work. However, it must make reference to `smart_ptr<>` in some very important contexts, such as the conversion constructor:

```
template <class SP, class OP, class CP, class KP, typename U>
to(smart_ptr<SP, OP, CP, KP>::to<U> const& rhs);
```

Unfortunately, the type of `rhs` will not be deduced, because `to<U>` creates a non-deducible context according to 14.8.2.4/4 [ISO98]. Nonetheless, the type of `rhs` is logically deducible because `to<U>` refers to a type that is entirely nested within `smart_ptr<>`. While there might be some merit to proposing a language change in which this special but important case were made deducible, the addition of template aliases would render this approach entirely unnecessary.

Policy Adaptor

Another common criticism of the library is that it has a large number of template parameters, and that specifying a non-default parameter requires specification of all parameters preceding it. This problem can be ameliorated by either named template parameters or a policy adaptor which automatically reorders policies as needed. It was decided that the latter approach would be easier for users, and was thus implemented. The policy adaptor uses `Boost.MPL` and policy category tags embedded in the policies to detect which non-default policies have been passed to the framework. It would be possible to use the TypeList facilities available in Loki to accomplish this task (or to even hand-roll a solution), but `MPL` enjoys broad development and strong support, making it fairly painless for implementers to use.

The resulting configuration is unspecified if a user passes two policies of the same category (for instance, two `OwnershipPolicies`). While it is possible to detect such errors programmatically, it seems unlikely that users who specify non-default policies will make such mistakes. Furthermore, such checking would only add to the compilation burden of a device that already weighs heavily on the compilation cost of the rest of the library. A more subtle kind of mistake is one in which the user specifies an invalid policy combination. This type of error could occur no matter what type of policy adaptor is used (including the trivial do-nothing adaptor). This type of problem is extremely difficult to solve programmatically, so this proposal suggests that the onus be placed on users to read the policy documentation carefully and make sure that a valid policy combination is selected. In most cases, it should be fairly obvious when a combination does not make sense[12], or when a custom policy set must be used together[13].

Provided Policies

Though there are an enormous number of policies which could be provided, the set included in the current framework is fairly representative of the types of configurations most users request. Following is a summary of those policies.

Storage Policies

- **`scalar_storage`**: This is the default policy for pointers to single objects allocated with `operator new()`. The reference implementation calls `boost::checked_delete()` to ensure that the type is complete at the time of destruction.

- **`array_storage`**: This policy supports builtin arrays, allowing emulation of `boost::scoped_array<>` and `boost::shared_array<>`. The array is disposed of with a call to `boost::checked_array_delete()`.

[12] Such as `array_storage<>` and `deep_copy<>`. Upon inspection, the user should quickly discover that it is impossible to define a member `clone()` function for built-in arrays, thus making this combination invalid.
[13] Such as the `shared_storage<>` and `boost_ref<>` policies which are used to emulate `boost::shared_ptr<>`.

Ownership Policies

- **`ref_counted`**: This is currently the default ownership policy. It enables a straightforward externally counted smart pointer. The count is allocated on the heap with the global `operator new()`, although an implementation could optimize this by using a custom allocator. However, [DDC03] argues that resources would be better spent optimizing the global `operator new()`.

- **`ref_linked`**: This is a reference-linked shared ownership policy that implements a doubly linked list for references. A singly-linked list was deemed too slow to be practical, though such an implementation might be appropriate for applications where the use count is known to be very low and/or heap allocation is to be avoided. Surprisingly enough, this policy exhibits very favorable performance characteristics among the non-intrusive ownership policies.

- **`com_ref_counted`**: This is a wrapper for Microsoft® COM pointers. It is the framework representative for intrusively counted pointers. No corresponding OMG CORBA® wrapper is provided, though there is no anticipated obstacles to providing such a policy. The Boost Smart Pointer emulation set also includes an emulation of `boost::intrusive_ptr<>`.

- **`deep_copy`**: This policy provides deep copying of the pointee through a `clone()` member function.

- **`no_copy`**: This policy enables `boost::scoped_ptr<>` emulation.

- **`move_copy`**: Although this policy is present in the current framework, a move pointer configuration is not currently functional due to various factors that are difficult to work around, including the absence of template aliases. See section V for details.

Checking Policies

- **`assert_check`**: This policy asserts on an invalid dereference. Null pointers are otherwise allowed. It also asserts on an attempt to `reset()` a pointer with itself, but all of the provided checking policies do so (and it is recommended that all user-defined policies do so as well).

- **`assert_check_strict`**: This policy asserts on invalid dereference as well as default initialization and initialization with a null pointer.

- **`reject_null`**: This policy is equivalent to `assert_check` except that it throws an exception instead of triggering an assertion.

- **`reject_null_strict`**: Analogously to `assert_check_strict`, this policy throws an exception on null initialization and invalid dereference, but allows default initialization.

- **`reject_null_static`**: This policy adds an assertion to `reject_null_strict` that disables default initialization.

Conversion Policies

- **`disallow_conversion`**: This is the default conversion policy which does exactly what it says (disallows implicit conversions to the raw pointer type).

- **`allow_conversion`**: This alternative is provided for completeness, but the user must select it explicitly.

Boost Emulation Policies
Though the Boost emulation policies have not been updated to work with the latest framework or to emulate the latest version of the Boost smart pointers, the earlier versions will be summarized to emphasize the point that emulation is possible.

- **`shared_storage`**: An interesting artifact of the `shared_ptr<>` architecture is that it does not neatly fit into the current framework's notion of separate storage and ownership policies. Nonetheless, by treating the different strategies as `StoragePolicies`, it is possible to provided the desired emulation. This policy enables emulation of `shared_ptr<>` itself.

- **`weak_storage`**: This policy enables emulation of `weak_ptr<>`.

- **`intrusive_storage`**: This policy enables emulation of `intrusive_ptr<>`.

- **`boost_ref`**: This policy is more or less a dummy `OwnershipPolicy`.

It should be noted that one way in which `shared_ptr<>` emulation fails is in providing support for initialization from a move pointer. Since `std::auto_ptr<>` emulation has not thus far been achieved in the current framework, this `shared_ptr<>` feature is also not fully operational. Another feature which has not been implemented is support for custom deleters. While part of the infrastructure exists to support this feature, it has not been completed, and there is no guarantee that it can be seamlessly integrated into the existing framework. There are no obvious hurdles, but the presence of a large number of templated constructors in `smart_ptr<>` tends to complicate matters. On the other hand, it may be possible in the current framework to support both runtime deleters and compile-time deleters, as requested in [Rame04].

IV. Proposed Text

Because the current implementation is designed as a workaround for the lack of template aliases, it would not be appropriate to describe a formal interface here based on that implementation. It would also be presumptuous to present an interface that assumed the presence of template aliases. Thus, the proposed text will be deferred to a future revision of this document, should the committee find favor with this proposal.

V. Unresolved Issues

Header Organization

The standard has established a precedent for placing memory management devices such as `auto_ptr<>` in the header `<memory>`. Proposal N1450=03-0033 has followed this precedent by adding smart pointer types to `<memory>`. However, there is a popular opinion within the C++ community (and Boost in particular) that headers should be fine-grained (to support the philosophy of not paying for what you don't use), and adding a sizeable framework to `<memory>` would contradict that opinion. The reference implementation is itself currently divided among several header files, with specific policy implementations being placed in separate files. So one possibility would be to reintegrate these files into `<memory>`, and another would be to create a new standard header `<smart_ptr>` (or `<smartptr>`, more in keeping with `<stdexcept>`, `<stdio>`, etc.). If the latter approach were chosen, it would still remain to be decided whether the standard policies should all appear in this header, or whether there should further be fine-grained separation of policy headers. If a fine-grained approach should be taken, it would make sense to put these headers in their own "header space", so to speak, like so:

```
#include <policy/array_storage>
#include <policy/deep_copy>
```

However, there is no precedent for such an organization in the Standard Library.

Move Semantics

At this time, a suitable implementation of a pointer having move semantics has not been achieved in the present framework. The primary obstacle is the need for a high degree of configurability. This results in a large number of templated constructors which makes overload resolution a very tricky business. The original mechanism, which was quite clever in its design, was found to be flawed due to the necessity of having both `const&` and non-`const&` conversion constructors.

Work by Sharoni [Shar03] on `std::auto_ptr<>` led to an SFINAE-based approach submitted by Daniel Wallin [in private communication]. Unfortunately, Daniel's technique relies on a partial template specialization that produces a non-deduced context for template arguments that must be deduced using the current nested design. One possible solution is to stay with the original mechanism and create a partial specialization of the entire framework for policy configurations that require move semantics. For many obvious reasons, this is an unattractive approach. There may yet be a workaround that will reasonably enable proper support for move semantics, but that goal has not yet been achieved in the current reference implementation. Yet another approach would be to simply exclude support for move semantics from the policy-based framework, and allow `std::auto_ptr<>` to remain a special case. Due to the occasional suggestions to improve `std::auto_ptr<>` in various ways, including adding features to it, this would not seem to be an ideal solution. Finally, if template aliases are added to the language, the nested design can be discarded and Mr. Wallin's approach will almost certainly work.

Non-Standard Policies
Some creative programmers have attempted to write policies for the proposed framework which implement unusual ownership strategies. They may well be exploring the limits of policy-based design, and it remains to be seen whether the framework can suitably accommodate them. Most notable among these attempts is Evans' `managed_ptr<>` [Evan04]. It would appear that he has mostly succeeded in writing a custom policy which implements a form of garbage collection. Hence, it may be possible that the proposed framework coupled with this policy set is one avenue to optional garbage collection in C++. A less successful attempt was Bouchard's `shifted_ptr<>` [Bou03], though it is unclear whether the failure was an inadequacy of the framework or a lack of real effort in writing a policy set. Thus the issue of whether the library delivers on the promise of accommodating a wide variety of custom policy configurations is as yet unresolved. The existence of the Boost emulation policies, which were not even anticipated when the framework was originally conceived, as well as Evans' `managed_ptr<>`, seems to bode well for this issue.

Default Configuration
The default policy configuration of the current library is an externally reference-counted smart pointer with assertion checking on dereference. That was the design decision chosen for the original Loki incarnation, and there was no immediate need to change it. This configuration reflects the observation stated earlier that shared ownership is the most common need and that safety should come first. It also presents a fairly light-weight pointer that has little memory overhead (either in the pointer itself or in associated heap-allocated resources) and reasonable performance. However, the adoption of `boost::shared_ptr<>` into the Library Technical Report brings up the issue of whether `shared_ptr<>` emulation should, in fact, be the default configuration should this proposal be accepted by the committee. There are no obvious obstacles to such an approach, and the wealth of experience which has gone into the development of `shared_ptr<>` may well recommend this as the preferred choice. It would also provide a smooth transition from the current TR to one which includes a policy-based framework. This proposal leaves it to the committee to make such a decision without offering a recommendation either way.

VI. Notes

1. Why implicit forwarding constructors don't help `smart_ptr<>`.
Due to the configurability requirements of the framework, many templated constructors exist in both `smart_ptr<>` and the policy classes. To ensure that the proper base class constructors get called in the presence of completely generic templated constructors, the object must be properly sliced before being passed to the base class. The code below illustrates:

```
smart_ptr(this_type const& rhs)
   : base_type(static_cast<base_type const&>(rhs))
{ ... }

template <typename U>
ref_counted_(U const& p)
   : base_type(p), count_(new counter_type(1))
{ }
```

```
ref_counted_(ref_counted_ const& rhs)
   : base_type(static_cast<base_type const&>(rhs)), count_(rhs.count_)
{ }
```

Suppose for a moment that `ref_counted<>` is the immediate base class of `smart_ptr<>` (that is, that `smart_ptr::base_type == ref_counted<>`). If the `smart_ptr<>` constructor did not cast `rhs` before passing it to the base initializer, the templated `ref_counted<>` constructor would get called instead of the copy constructor. This is a general problem not limited to the current framework, which perhaps reduces the utility of N1583=04-0023. That is, whenever a base class contains a generic templated constructor, a copy constructor in the derived class must cast its argument to the base type just to ensure that the base class copy constructor gets called rather than the templated constructor. Thus, there is no obvious way for such a derived class to inherit a useful copy constructor.

2. How cleanup occurs in the chained-policy design

The original `Loki::SmartPtr<>` design orchestrated initialization and cleanup like so:

```
template
<
   typename T,
   template <class> class OwnershipPolicy,
   class ConversionPolicy,
   template <class> class CheckingPolicy,
   template <class> class StoragePolicy
>
class SmartPtr
   : public StoragePolicy<T>
   , public OwnershipPolicy<typename StoragePolicy<T>::PointerType>
   , public CheckingPolicy<typename StoragePolicy<T>::StoredType>
   , public ConversionPolicy
{
   // ...
public:
   SmartPtr(const StoredType& p) : SP(p)
   {
      KP::OnInit(GetImpl(*this));
   }

   ~SmartPtr()
   {
      if (OP::Release(GetImpl(*static_cast<SP*>(this))))
      {
         SP::Destroy();
      }
   }
};
```

The `StoragePolicy` receives the resource, then the `Ownership`, `Checking`, and `Conversion` policies are constructed, and then the `CheckingPolicy`’s `OnInit()` function is called (e.g.: to test for null). When `~SmartPtr()` gets called, the `OwnershipPolicy` releases ownership of the resource and returns a bool specifying whether it is safe to destroy the resource. If it is, the `StoragePolicy` actually cleans it up. However, suppose that the `OwnershipPolicy` is `RefCounted<>` (which provides external reference counting). As one can see below, `RefCounted<>` performs a dynamic allocation to create the count:

```
RefCounted()
{
   pCount_ = static_cast<unsigned int*>(
      SmallObject<>::operator new(sizeof(unsigned int))
   );
   assert(pCount_);
   *pCount_ = 1;
}
```

In the following scenario:

```
SmartPtr<X> p(new X);
```

It is possible for the StoragePolicy to successfully acquire the resource, and then for the `OwnershipPolicy` (in this case, `RefCounted<>`) to throw an exception (such as `std::bad_alloc`). In that case, the acquired resource is leaked because the smart pointer is not fully constructed, and thus `~SmartPtr()` never gets called. To rectify this lack of exception safety, Held redesigned the cleanup scheme as follows:

- By default, the `StoragePolicy` destroys the resource in its destructor
- If the `StoragePolicy` should not destroy the resource, `StoragePolicy::release()` will be called to erase the `StoragePolicy`'s reference to the resource. Then, the `StoragePolicy` destructor will have no effect.
- By default, the `OwnershipPolicy` destroys any resources it allocates in its destructor
- If the `OwnershipPolicy` should not destroy its resources, `OwnershipPolicy::release()` will ensure that references to those resources are properly erased to cause the `OwnershipPolicy` destructor to have no effect.

The code below illustrates the new design with the multiple inheritance architecture (and thus, the `resource_manager<>` mechanism):

```
template
<
   typename T,
   class StoragePolicy, class OwnershipPolicy,
   class CheckingPolicy, class ConversionPolicy
>
class smart_ptr
 : public optimally_inherit<
      detail::resource_manager<
         StoragePolicy,
         OwnershipPolicy
      >,
      typename optimally_inherit<
         CheckingPolicy,
         ConversionPolicy
      >::type
   >::type
{ ... };

template <class StoragePolicy, class OwnershipPolicy>
class resource_manager
 : public optimally_inherit<StoragePolicy, OwnershipPolicy>::type
{
   // ...
public:
   ~resource_manager()
   {
      if (!ownership_policy::release(get_impl(*this)))
      {
         storage_policy::release();
      }
   }
}
```

The `resource_manager<>` destructor says: "Let the `OwnershipPolicy` release the resource. If it was not the last owner, then we must prevent the `StoragePolicy` from cleaning it up. Hence, call `StoragePolicy::release()`."

```
   template <typename T>
   class scalar_storage
   {
      // ...
   protected:
      ~scalar_storage()
      {
         boost::checked_delete(pointee_);
      }

      void release()
      {
         pointee_ = 0;
      }
   private:
      stored_type pointee_;
   };
```

Here one can see that by default, `scalar_storage<>` deletes the resource. When `release()` is called, `scalar_storage<>` is absolved of responsibility (its share of the ownership is released) and the destructor deletes the null pointer.

```
template <typename P>
class ref_counted
{
   // ...
protected:
   ~ref_counted()
   {
      delete count_;
   }

   bool release(P const&)
   {
      if (!count_ || !--*count_) return true;
      count_ = 0;
      return false;
   }
};
```

And the theme above is repeated again in the `OwnershipPolicy`. If `ref_counted<>` is the last owner, it signals that it is safe to delete the resource. In turn, `resource_manager<>` responds by doing nothing, and the `StoragePolicy` will clean up the resource. If it is not the last owner, it erases its reference to the count, and its destructor deletes the null pointer. On exit, `resource_manager<>` tells the `StoragePolicy` to release its ownership, thus preventing cleanup of the shared resource.

The user requirement to obtain the basic guarantee for the initialization of `smart_ptr<>` is that the `StoragePolicy` constructor (which gets called first) must either succeed or clean up the resource (that is, the `StoragePolicy` constructor itself must give the basic guarantee). For `scalar_storage<>`, the constructor is guaranteed to succeed. However, if the `OwnershipPolicy` throws an exception, the `StoragePolicy` subobject is fully constructed, so its destructor will properly clean up the resource. The `OwnershipPolicy` constructor must also provide the basic guarantee, and this is true for `ref_counted<>` (as well as all the other policies included in the reference implementation). If the `StoragePolicy` and `OwnershipPolicy` constructors succeed, it is still possible for the `CheckingPolicy` constructor to throw. If this occurs, everything will be cleaned up properly because the `resource_manager<>` subobject is fully constructed and possesses all knowledge required to properly dispose of the resource. Thus, the basic guarantee is obtained[14]. While this seems to be a fairly elegant solution to the exception safety problem, the chained policy architecture provides even tighter coupling:

```
template <class StoragePolicy>
class ref_counted_ : public StoragePolicy
{
   // ...
protected:
```

[14] It is assumed that the `CheckingPolicy` and `ConversionPolicy` are stateless and pure.

```
      ~ref_counted_(void)
      {
         if (!count_ || !--*count_)
         {
            delete count_;
         }
         else
         {
            storage_policy::release();
         }
      }
   }
```

The `StoragePolicy` remains as before, but the fact that the `OwnershipPolicy` now has access to the `StoragePolicy`'s interface allows for a very streamlined cleanup design. Since the polices need not be orchestrated by a third party (`smart_ptr<>` or `resource_manager<>`), there is no need to have `OwnershipPolicy::release()`. Instead, the `release()` code is merged directly with the destructor code. It should be apparent that this code is somewhat shorter, and avoids the additional comparison which tested the return value of `release()`. On the other hand, writers of custom `OwnershipPolicies` must now remember to call `StoragePolicy::release()` at the appropriate time, which was not formerly a requirement.

The astute reader may note that in some configurations, the `StoragePolicy` may know how to clean up the resource on failed initialization, but may not know whether that is the appropriate action to take. Consider an intrusively counted pointer. If the `OwnershipPolicy` were to throw an exception on construction, the default `StoragePolicy` would blithely destroy the resource, because it does not know about intrusive counts. In scenarios where proper initialization cleanup requires assistance from the `OwnershipPolicy`, this design requires that both the `StoragePolicy` and `OwnershipPolicy` constructors not fail. For most intrusive designs, this requirement is trivially met.

3. Objections

Due to the often parochial nature of smart pointer design, a policy-based framework is surprisingly controversial. It raises a number of objections, the most common of which are presented and addressed below.

1. **Multiple Inheritance (MI) should be avoided**
   This objection was primarily related to the fact that EBO is poorly supported for the MI case, and thus the resulting smart pointers were a less-than-optimal size. It was even debated whether EBO is always allowed in the presence of MI. This objection was addressed first with the `optimally_inherit<>` mechanism, and later with the chained-policy architecture.

2. **Parameterization discourages use**
   While this was perhaps true in 1998 when it was first put forth as an argument against a complex smart pointer type [CB99], the argument hardly carries the same force today with the existence of such popular libraries as `Boost.Function` [Greg01] (which even made it into the TR), `Boost.Graph` [SLL00], `Boost.Iterator` (also in TR1), `Boost.MPL` (which

makes the use of parameterization in the current framework seem almost quaint), `Boost.Operators`, `Boost.Python` [Abra02], etc. ad nauseum.

Nonetheless, there are ancillary issues related to this objection that were addressed. One is the specification of non-default policies, which was accommodated with a policy adaptor that detects the policy types and reorders them appropriately. This alleviates the burden of spelling out intervening default policies. Another is the need to abbreviate long policy configurations with typedefs or some other mechanism. While it cannot be stressed enough how much template aliases would benefit this library, this issue was handled by introducing a type generator architecture that allows users to typedef their favorite configurations while still parameterizing over the pointee type.

3. **The feature set is part of the type.**
This objection is raised primarily in contrast to `boost::shared_ptr<>`, in which optional features such as custom deleters and embedded counts are handled at runtime, leaving a unified type as an interface between libraries. From N1450=03-0033: "Following the 'as close as possible' principle, the proposed smart pointers have a single template parameter, the type of the pointee. Avoiding additional parameters ensures interoperability between libraries from different authors, and also makes *shared_ptr* easier to use, teach and recommend." [DDC03]

The interoperability principle can be retained by agreeing that the `shared_ptr<>` emulation of the current framework is the de facto inter-library configuration. Since some authors do not agree that interoperability is an essential criterion and write their own smart pointer types anyway, it seems sensible to accommodate those programmers with a policy-based framework. In other cases, it is apparent that the types of smart pointers ought to be different, as they should not be mixed any more than one should attempt to intermingle raw data pointers and function pointers.

For pedagogical purposes, one could again restrict discussion to the `shared_ptr<>` configuration, if it is deemed that custom policies would confuse students. The addition of template aliases would allow the `shared_ptr<>` emulation to look exactly like the standalone implementation. On the other hand, user configurability of libraries is fast becoming a popular trend in C++ and the current framework may actually turn out to be an aid in presenting this notion to students.

When recommending a smart pointer to a user that is new to the concept, the default should certainly make the most sense and offer the least resistance to adoption. If the `shared_ptr<>` emulation were the default configuration of the proposed framework, then the recommendation to use `shared_ptr<>` would apply equally well to the current proposal.

4. Future Directions

The following features are under consideration for addition to the proposed framework, but should not be considered part of this proposal itself. They are included merely to inform the committee that they may be present in a future revision of this proposal.

- Hinnant, in [Hinn03], demonstrates an elegant syntax for a smart pointer which also supports arrays. Namely, `move_ptr<T[]>`. It may be possible to support this syntax in the proposed framework, and that issue will be explored.

- Lovset suggests a means to protect inadvertent initialization of a smart pointer with a raw pointer in [Lov04]. Namely, initialization from a raw pointer would require a type of explicit cast to a proxy type, like so:

  ```
  smart_ptr<X> p = smart_ptr_cast<X>(new X);
  ```

  This would avoid unintentionally assigning a raw pointer to a smart pointer when transfer of ownership is not intended. The proposal also includes a means to allow `NULL` assignment to be a synonym for `p.reset()`. While the merits and complications of this proposal have not been fully evaluated, it is a possible direction that the proposed framework might take.

- Not only do custom deleters specified at runtime need to be supported by the proposed framework to achieve complete `shared_ptr<>` emulation, but static deleters that become part of the type should also be considered.

- While the original Loki implementation included a thread-safe externally reference counted `OwnershipPolicy`, unresolved issues regarding multithreading in C++ have caused that policy to be removed from the current offering. If and when multithreading is properly supported, issues of thread safety will be reconsidered and the appropriate policies will be added.

VII. Acknowledgements

This proposal would not be possible without the generous encouragement, support, and advice of Andrei Alexandrescu. The heart and soul of the library is his handiwork, and most of the major design decisions and philosophies are his. The refinements to the original Loki work were mainly the result of thoughtful criticism and suggestions from the Boost community, including, but certainly not limited to: Gennadiy Rozental, Beman Dawes, Dave Abrahams, Giovanni Bajo, Phil Nash, Peter Dimov, Greg Colvin, and Doug Gregor. Contributions to the reference framework were provided by Aleksey Gurtovoy, Paul Mensonides, Daniel Wallin, Larry Evans, and others. Porting assistance was provided by Jonathan Turkanis and others. Finally, Andrei graciously lent his insights and suggestions to help refine this proposal.

VIII. References

[Abra99] Abrahams, D., et al. 1999. Boost Operators Library Documentation. http://www.boost.org/libs/utility/operators.htm.

[Abra02] Abrahams, D. 2002. Boost Python Library Documentation. http://www.boost.org/libs/python/doc/index.html.

[Abra03] Abrahams, D., et al. 2003. Boost Utility Library Documentation. http://www.boost.org/libs/utility/utility.htm.

[ASW03] Abrahams, D., Siek, J., and Witt, T. 2003. Boost Iterator Library Documentation. http://www.boost.org/libs/iterator/doc/index.html.

[AH04] Alexandrescu, A., and Held, D. B. 2004. Smart pointers reloaded (iv): finale. *C/C++ User's Journal*, 22, 4.

[Alex01] Alexandrescu, A. 2001. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, New York, NY.

[Bou03] Bouchard, P. 2003. shifted_ptr documentation. http://fornux.com/personal/philippe/devel/shifted_ptr/libs/smart_ptr/doc/.

[Colv99] Colvin, G. Boost Mailing List. weak pointers and cycles. http://article.gmane.org/gmane.comp.lib.boost.devel/38518/match=cyclic+ptr.

[CB99] Colvin, G., Dawes, B., et al. 1999. Boost Smart Pointer Library Documentation. http://www.boost.org/libs/smart_ptr/smart_ptr.htm.

[DDC03] Dimov, P., Dawes, B., and Colvin, G. 2003. A proposal to add general purpose smart pointers to the library technical report. *ISO/IEC JTC1/SC22/WG21 Document*, N1450=03-0033.

[Evan04] Evans, L. 2004. managed_ptr documentation. http://cvs.sourceforge.net/viewcvs.py/boost-sandbox/boost-sandbox/libs/managed_ptr/doc/html/.

[Glas04] Glassborow, F. 2004. Inheriting constructors. *ISO/IEC JTC1/SC22/WG21 Document*, N1583=04-0023.

[Greg01] Gregor, D. 2001. Boost Function Library Documentation. http://www.boost.org/doc/html/function.html.

[Grif99] Griffiths, A. 1999. Octopull/C++. Ending with the grin. http://www.octopull.demon.co.uk/arglib/TheGrin.html.

[GAW02] Gurtovoy, A., Abrahams, D., and Winch, E. 2002. Boost MetaProgramming Library Documentation. http://www.boost.org/libs/mpl/doc/index.html.

[HDA02] Hinnant, H., Dimov, P., and Abrahams, D. 2002. A proposal to add move semantics support to the C++ language. *ISO/IEC JTC1/SC22/WG21 Document*, N1377=02-0035.

[Hinn03] Hinnant, H. 2003. move_ptr<> implementation. http://home.twcny.rr.com/hinnant/Utilities/move_ptr.

[ISO98] ISO/IEC 14882:1998(E). 1998. Programming languages – C++.

[KM02] Karvonen, V., and Mensonides, P. 2002. Boost Preprocessor Library Documentation. http://www.boost.org/libs/preprocessor/doc/index.html.

[Lov04] Lovset, T. 2004. Boost Mailing List. http://article.gmane.org/gmane.comp.lib.boost.devel/107435/match=nullptr.

[Madd01] Maddock, J., et al. 2001. Boost Type Traits Library Documentation. http://www.boost.org/libs/type_traits/index.html.

[Rame04] Ramey, R. 2004. Boost Mailing List. scoped_ptr<T, D>. http://thread.gmane.org/gmane.comp.lib.boost.devel/103937.

[Shar03] Sharoni, R. 2003. Usenet newsgroup: comp.std.c++. auto_ptr: improved implementation. http://groups.google.com/groups?q=g:thl1103834576d&dq=&hl=en&lr=&ie=UTF-8&safe=off&selm=3fc75686%40news.microsoft.com&rnum=1.

[SLL00] Siek, J., Lee, L., and Lumsdaine, A. 2000. Boost Graph Library Documentation. http://www.boost.org/libs/graph/doc/table_of_contents.html.

[SD03] Stroustrup, B. and Dos Reis, G. 2003. Template aliases for C++. *ISO/IEC JTC1/SC22/WG21 Document*, N1489=03-0072.

[Sutt02] Sutter, H. 2002. Proposed addition to C++: typedef templates. *ISO/IEC JTC1/SC22/WG21 Document*, N1406=02-0064.