Document Number: N1851=05-0111

Date: 2005-08-25

Improving Usability and Performance of TR1 Smart Pointers

Vladimir Kliatchko <vladimir@kliatchko.com>,
Ilougino Rocha <irocha@mac.com>,
August 25, 2005

Abstract

Over a period of two years we have accumulated significant experience in developing and then using a family of C++ components implementing smart pointers. While similar to the smart pointers described by TR1, our implementation has some important distinctions. Our experience shows that these distinctions often resulted in both improved usability and superior performance. In this paper we describe how our implementation differs from TR1 and propose a number of corresponding changes to the C++ standard.

Copyright and Disclaimer

Table of Contents

Introduction

Over the last two years, we have accumulated significant experience in developing and using a family of C++ components implementing smart pointers. These components have been successfully used in the development of a number of mission-critical applications on a variety of platforms (Sun Solaris, IBM AIX, HP HP-UX, Linux, and Windows). Our implementation has proven to be reliable, portable, and highly efficient (see Appendix A for performance comparison against Boost).

While our implementation of smart pointers is similar to that provided by Boost and described by TR1, it provides a number of additional features not found in Boost/TR1: allocator support, `shared_ptr` aliasing, housing of shared objects in `shared_ptr`, and support for smart pointers having destructive copy semantics (`managed_ptr`). We found these additional features to be essential for our applications, allowing for more straightforward design and improved performance. Gains in performance and usability were most noticeable in highly-optimized scalable multi-threaded applications. This class of applications is a traditional domain of the C++ programming language; hence, supporting such applications should be a priority, especially considering the rapidly growing importance of multithreaded programming.

The remainder of this paper describes the distinguishing features of our implementation of smart pointers and formulates corresponding proposals for the inclusion in the C++ standard.

The proposal is organized as follows:

Sections **"Allocator Support"**, **"Shared Pointer Aliasing"**, **"Housing Shared objects in shared pointers"**, and **"Smart Pointer with Destructive Copy Semantic (`managed_ptr`)"** each contain a proposal for inclusion in the standard of the corresponding feature of our smart pointer implementation.

The **"Conclusion"** section summarizes our proposals.

Appendix A contains benchmarks comparing performance of our implementation with that of Boost.

Appendix B contains a short sample program that illustrates both the motivation for and the use of the proposed features.

Allocator Support

Allocator support provided by our implementation of smart pointers allows the user of our `shared_ptr` to control memory allocations performed by each `shared_ptr` internally. More specifically, our `shared_ptr`'s interface includes two additional constructors and two matching `reset` methods, each taking an additional allocator argument. This allocator is used to supply memory for the reference count structure. Once the reference count structure is allocated, a copy of the allocator is stored in the structure to be used for the deallocation.

We can expect, and our experience confirms, that the ability to control memory allocation in `shared_ptr` will be crucial in a number of common scenarios. For example:

- It may be necessary to place instances of `shared_ptr` in a special memory region (e.g., shared memory).

- We may want to use a special pool to reduce contention and improve performance in a multithreaded application that creates a large number of `shared_ptr` instances concurrently.

It has been observed in [N1450] that using pools for reference count structures does not provide significant benefits. However, our experience with pooling reference count structures was radically different. The multithreaded applications that we built tended to use `shared_ptr` extensively and in performance-critical areas. In general we expect `shared_ptr` to be especially common in high-performance multithreaded applications since keeping track of each individual object's lifetime in these applications is particularly difficult: object ownership passes from thread to thread and objects are bound to asynchronous callbacks. In such applications we discovered that the allocation of the reference count structure was a common cause of contention and the cost of such allocation was prohibitive unless a special pool was used. High-performance lock-free pools combined with an ability to supply distinct pools to different instances of `shared_ptr` gave us the means necessary to reduce memory contention and significantly improve performance.

When incorporating support for allocators into `shared_ptr` we made several design decisions that are worth pointing out:

- **Templatazing individual methods rather than the entire class.**
  Other components in the standard library that support custom allocators do so via parameterizing the entire class on the type of the allocator. This approach results in generating a new C++ type for each distinct type of allocator used. This class-wide parameterization results in multiple incompatible types leading to interoperability problems that we felt compelled to avoid. Our solution for achieving interoperability with respect to allocator parallels that of Boost with respect to deleters. That is, we have chosen to parameterize on allocator type only those individual methods that take allocator parameters, thus avoiding the problem. The allocator type is then "baked" into the reference count structure just as it is done with the type of the deleter object.

- **Passing an allocator by value rather than by `const` reference or reference.**

TR1 specifies that a deleter is to be passed to `shared_ptr` constructors and `reset` methods by value. Passing the deleter by value results in an additional 'no throw' requirement on the deleter's copy constructor. Other alternatives are to pass the deleter by reference or `const` reference. [N1450] explains the trade-offs among these options. Passing the deleter by `const` reference would require the `operator()` to be declared as `const`. Passing the deleter by modifiable reference is rejected because that would prevent a caller from passing a temporary object. Given these considerations, we made the same choice and decided to pass an allocator by value, thus imposing a 'no throw' requirement on the allocator's copy constructor. However, to alleviate some of the pain associated with this requirement, we suggest that the standard strengthens its guarantee on the function template `bind` by specifying that `bind` shall not throw if the copy constructors of the arguments do not throw. Our experience implementing `bind` indicates that this additional constraint is easy to satisfy. On the other hand, this guarantee would allow such common idioms as constructing a `shared_ptr` with a deleter expressed as a member function bound to an object pointer.

- **Requiring an explicit deleter to be supplied when a custom allocator is used.** Although, it is not uncommon (in our experience) that a `shared_ptr` needs to use a custom allocator for internal memory, but does not require a custom deleter for the shared object, we have chosen not to provide a separate constructor for this usage case. Such a constructor would take two parameters – a pointer to a shared object and an allocator. This constructor, however, would have a signature identical to the constructor that takes a pointer to shared object and a deleter:

```
template<class Y, class D> shared_ptr(Y * p, D deleter);
template<class Y, class A> shared_ptr(Y * p, A allocator);
```

  In both constructors the type of the second parameter is templatized and therefore we cannot immediately distinguish between the constructor taking an allocator and the constructor taking a deleter. It is possible to resolve the ambiguity between the constructors by introducing a new trait (e.g., `is_allocator`) and requiring all the allocator implementations (including `std::allocator`) to support this trait. However, after considering the additional complexity that such a trait would introduce, we decided against supporting custom allocators without explicit deleters. Requiring an explicit deleter whenever an allocator is used will also help to eliminate potential programmer's errors when an allocator and a deleter might be confused. The provided `default_deleter` (that simply invokes the `delete` operator) can be use when only a custom allocator (and not a custom deleter) is required.

The following section describes the amendments to the C++ standard for allocator support in `shared_ptr`.

Proposed Text

Additions to header <memory> synopsis ([std] 2.4, [tr1] 2.2.1)

```
namespace std {
namespace tr1 {
    struct default_deleter;
}
}
```

Class template `default_deleter`

```
namespace std {
namespace tr1 {
    struct default_deleter {
           template<class T> void operator () (T *p) const;
    };
}
}
```

`template<class T> void operator () (T *p) const;`

Effects

Equivalent to `delete p`.

Additions to `shared_ptr` *constructors ([tr1] 2.2.3.1)*

`template<class Y, class D, class A> shared_ptr(Y * p, D d, A a);`

Requires

`p` is convertible to `T *`. `D` is ***CopyConstructible***. The copy constructor and destructor of `D` shall not throw exceptions. The expression `d(p)` shall be well-formed, shall have well-defined behavior, and shall not throw exceptions.

`A` is an ***allocator*** (see "allocator requirements", [std] 20.1.5). The copy constructor and destructor of `A` shall not throw exceptions.

Effects

Constructs a `shared_ptr` object that ***owns*** the pointer p and the deleter `d`.

Uses a copy of the allocator `a` to allocate memory for the `shared_ptr`'s internal use.

Postconditions

`use_count() == 1 && get() == p`.

Throws

`bad_alloc`, or an implementation-defined exception when a resource other than memory could not be obtained.

Exception safety:

If an exception is thrown, `d(p)` is called

```
template<class Y, class A> shared_ptr(auto_ptr<Y>& r, A a);
```

Requires

`r.release()` shall be convertible to T *. Y shall be a complete type. The expression `delete r.release()` shall be well-formed, shall have well-defined behavior, and shall not throw exceptions.

`A` is an ***allocator*** (see “allocator requirements”, [std] 20.1.5).

Effects

Constructs a `shared_ptr` object that ***owns*** `r.release()`.

Uses a copy of the allocator `a` to allocate memory for the `shared_ptr`’s internal use.

Postconditions

`use_count() == 1 && r.get() == 0`.

Throws

`bad_alloc`, or an implementation-defined exception when a resource other than memory could not be obtained.

Exception safety

No effect ,if an exception is thrown.

Additions to `shared_ptr` modifiers ([tr1] 2.2.3.4)

```
template<class Y, class D, class A> void reset(Y * p, D d, A a);
```

Effects

Equivalent to `shared_ptr(p, d, a).swap(*this)`.

```
template<class Y, class A> void reset(auto_ptr<Y>& r, A a);
```

Effects

Equivalent to `shared_ptr(r, a).swap(*this)`.

Shared Pointer Aliasing

While using our smart pointer components on a variety of projects, we discovered that it is often important to link the lifetime of one object with that of another. For example, suppose that you have a shared pointer to an object (e.g., a container) and that you need to obtain a shared pointer to some sub-object (e.g., an element of that container), say, to provide to an existing interface. When the shared pointer to the sub-object is destroyed, however, the shared pointer must not attempt to destroy the sub-object directly. The reference to the sub-object is implicitly a reference to the parent object. Once no part of the parent object is in use, the parent object should be destroyed instead.

This often useful feature, which we call smart pointer aliasing, lends itself to a simple and efficient implementation. In order to create a `share_ptr` alias, one constructs a `shared_ptr` that uses the same reference count structure as the original `shared_ptr` yet points to different object. In our implementation, as in the Boost implementation, the reference count structure retains the original pointer and a copy of the original deleter (when supplied at construction) so that the object can still be destroyed using the original pointer. This same mechanism is used in the implementation of various casting functions (e.g., `dynamic_pointer_cast`). In fact, our original version of smart pointer components (prior to the changes we made for compatibility with TR1) omitted the explicit cast functions since identical results were achievable by first obtaining the underlining raw pointer, performing the desired cast, and then aliasing that pointer with the original `shared_ptr`.

In fairness, we acknowledge that aliasing does introduce additional opportunities for misuse and programming errors. However, in our experience, such errors were not common and almost always detectable at compile time. On the other hand, not having the aliasing feature would have often resulted in complicated and error-prone work-around code with inferior performance.

The following section describes the amendments to the C++ standard for `shared_ptr` aliasing. Note, that our use of the term ***ownership*** when applied to an aliased shared pointer is somewhat imprecise since an object returned from a `shared_ptr`'s `get` method may be different from the one destroyed when its reference count reaches zero. This imprecision is exacerbated in the presence of conversions to and from `managed_ptr`. These conversions render our use of the term deleter imprecise as well. For consistency with the TR1 specification, however, we will continue to use these terms. We describe `managed_ptr` in the "Smart Pointer with Destructive Copy Semantic (managed_ptr)" section below.

Proposed Text

Additions to `shared_ptr` constructors ([tr1] 2.2.3.1)

```
template<class Y> shared_ptr(shared_ptr<Y> const& s, T *p);
```

Effects

Constructs a `shared_ptr` object that ***shares ownership*** with `s`. If `s` is ***empty***, requires `p==0` and constructs an ***empty*** `shared_ptr`.

Postconditions

`get() == p`

Throws

Nothing

Additions to `shared_ptr` modifiers ([tr1] 2.2.3.4)

```
template<class Y> void reset(shared_ptr<Y> const& s, T *p);
```

Effects

Equivalent to `shared_ptr(s, p).swap(*this)`.

Housing Shared Objects in Shared Pointers

We used the feature described in this section frequently in our applications for improving performance. This feature allows an object whose lifetime is controlled by a `shared_ptr` to be embedded directly within the reference count structure of the `shared_ptr`. This collocation of the reference count with the object improves locality of reference, reduces memory fragmentation, and, by avoiding an additional memory allocation, significantly accelerates the construction of shared object and alleviates contention in multithreaded applications. This feature, in combination with supplying `shared_ptr` with an efficient allocator (see "Allocator Support"), often allowed us to improve the scalability of multithreaded applications significantly.

Implementation of this feature is straightforward. We introduce four new free (non-member) functions: `make_shared_object`, `make_shared_object_with_alloc`, `make_shared_array`, and `make_shared_array_with_alloc`. These functions construct `shared_ptrs` with reference count structures of appropriately increased size. The extra room in the structure is then used to house an embedded object (or array of objects). Note that any extra arguments supplied to the `make_shared_object` and `make_shared_object_with_alloc` functions are automatically passed through to the constructor of the embedded object.

Providing free (rather than member) functions simplifies usage since free functions (that return a `shared_ptr` value) make it unnecessary to construct an instance of `shared_ptr` before creating a collocated shared object. In one case a member function might have had a slight performance advantage: when the result of one of the free functions is assigned to an existing `shared_ptr` variable no temporary instance of `shared_ptr` would have had to be created (and, thus, no additional updates to the reference would have been needed). However, this performance advantage is not significant and can be eliminated completely when rvalue-reference support is adopted into the language (see [N1377] for more information on rvalue references).

The following section describes the `make_shared_object`, `make_shared_object_with_alloc`, `make_shared_array`, and `make_shared_array_with_alloc` functions.

Proposed Text

Housing shared object in `shared_ptr`

```
template<class X, class T1, class T2, …, class TN>
shared_ptr<X> make_shared_object (T1 t1, T2 t2, …, TN tn);
```

Requires

`X` is a complete type. The expression `new ((void *)&s) X(t1, t2, …, tn)`, where `s` is of type `std::tr1::aligned_storage<size_of X,`

`std::tr1::alignment_of<X>::value>::type`, shall be well-formed and shall have well-defined behavior.

Returns

A newly constructed `shared_ptr` that houses an object of type `X`. The object of type `X` is constructed in memory owned by the `shared_ptr` internally passing the optional arguments `t1`, `t2`, … to the constructor of `X`. The `shared_ptr` ***owns*** this object of type `X` and the appropriate deleter.

Throws

`bad_alloc`, or an implementation-defined exception when a resource other than memory could not be obtained.

Exception safety

If an exception is thrown, has no effect.

```
template< class X, class A, class T1, class T2, …, class TN>
shared_ptr<X> make_shared_object_with_alloc (
                                A a, T1 t1, T2 t2, …, TN tn);
```

Requires

`X` is a complete type. The expression `new ((void *)&s) X(t1, t2, …, tn)`, where `s` is of type `std::tr1::aligned_storage<size_of X, std::tr1::alignment_of<X>::value>::type`, shall be well-formed and shall have well-defined behavior.

`A` is an ***allocator*** (see “allocator requirements”, [std] 20.1.5).

Returns

A newly constructed `shared_ptr` that houses an object of type `X`. The object of type `X` is constructed in memory owned by the `shared_ptr` internally passing the optional arguments `t1`, `t2`, … to the constructor of `X`. The `shared_ptr` ***owns*** this object of type `X` and the appropriate deleter.

Uses a copy of the allocator `a` to allocate memory for the `shared_ptr`’s internal use.

Throws

`bad_alloc`, or an implementation-defined exception when a resource other than memory could not be obtained.

Exception safety

If an exception is thrown, has no effect.

```
template<class X>
shared_ptr<X> make_shared_array (std::size_t n);
```

Requires

`X` is a complete type. The expression `new ((void *)&s) X`, where `s` is of type `std::tr1::aligned_storage<size_of X, std::tr1::alignment_of<X>::value>::type`, shall be well-formed and shall have well-defined behavior.

Returns

A newly constructed `shared_ptr` that houses an array of `n` object of type `X`. The array is constructed in memory owned by the `shared_ptr` internally. The `shared_ptr` ***owns*** this array and the appropriate deleter.

Throws

`bad_alloc`, or an implementation-defined exception when a resource other than memory could not be obtained.

Exception safety

If an exception is thrown, has no effect.

```
template<class X, class A>
shared_ptr<X> make_shared_array_with_alloc (A a, std::size_t n);
```

Requires

`X` is a complete type. The expression `new ((void *)&s) X`, where `s` is of type `std::tr1::aligned_storage<size_of X, std::tr1::alignment_of<X>::value>::type`, shall be well-formed and shall have well-defined behavior.

`A` is an ***allocator*** (see "allocator requirements", [std] 20.1.5).

Returns

A newly constructed `shared_ptr` that houses an array of `n` object of type `X`. The array is constructed in memory owned by the `shared_ptr` internally. The `shared_ptr` ***owns*** this array and the appropriate deleter.

Uses a copy of the allocator `a` to allocate memory for the `shared_ptr`'s internal use.

Throws

`bad_alloc`, or an implementation-defined exception when a resource other than memory could not be obtained.

Exception safety

If an exception is thrown, has no effect.

Smart Pointer with Destructive Copy Semantics (managed_ptr)

In our applications we discovered that it is common to employ object factories, object pools, and various kinds of custom allocators for object construction. When an object, so constructed, is no longer needed it should be either returned to the corresponding factory or pool, or destroyed, with the object's memory returned to the corresponding allocator. Using a smart pointer to control the lifetime of such an object requires that the smart pointer supports a custom deleter. While `shared_ptr` provides support for a custom deleter our experience showed that the reference counting semantics of `shared_ptr` often resulted in undesired overhead. A smart pointer with destructive copy semantics similar to `auto_ptr` would often be sufficient and more appropriate. This observation motivated us to construct a variant of `auto_ptr` with support for a custom deleter. We called this new type of smart pointer `managed_ptr`.

Using our `managed_ptr`, we were able to improve both the performance and the clarity of our application code. The performance benefits of `managed_ptr` come from its significantly lower construction costs (no memory allocation is required) and copying costs (no reference count needs updating) as compared with those of `shared_ptr`. The performance improvements were most noticeable in multithread applications, where memory allocation is a common cause of contention and reference count updates must be serialized.

The design and clarity of application code benefited because the use of `shared_ptr` where no actual sharing of ownership takes place would be misleading (and, therefore, confusing). Replacing `shared_ptr` with `managed_ptr` in these cases resulted in more straightforward easier to understand design.

In addition to destructive copying semantics, our implementation of `managed_ptr` has a number of important features that are worth pointing out:

- Since `managed_ptr`'s constructor does not need to allocate any memory, the constructor is able to provide a "no throw" guarantee. This guarantee made `managed_ptr` a useful tool in the development of exception safe code.

- Our `managed_ptr` is convertible to and from `shared_ptr`. Support for these conversions makes `managed_ptr` a more flexible and efficient "vocabulary" (interface) type than `shared_ptr`. The conversions enable a caller to choose whether or not to transfer ownership through an interface. If the caller of the interface wishes to transfer ownership, a `managed_ptr` is passed and the performance benefits of destructive copy semantics are realized. If the caller instead chooses to retain ownership, the caller passes a `shared_ptr` and can rely on an efficient conversion from `shared_ptr` to `managed_ptr`.

- Similar to `shared_ptr`, `managed_ptr` supports aliasing. Aliasing occurs when a `managed_ptr` is constructed from another `managed_ptr` and a new object different from the one pointed to by the original `managed_ptr` is supplied. This new object is assumed to have its lifetime linked to that of the original object. When an aliased `managed_ptr` is destroyed, instead of destroying the object pointed to by the `managed_ptr`, the object managed by the original `manage_ptr` is destroyed. The motivation for this important

feature is similar to the motivation for aliasing of `shared_ptr` instances (see section "Shared Pointer Aliasing").

The implementation of `managed_ptr` employs a number of techniques similar to those used by the implementations of `auto_ptr`, `shared_ptr`, and `tr1::function`. Each `managed_ptr` object is comprised of four data members:

- The "current" pointer: the pointer to be returned by the `get` method.
- The "original" pointer: the pointer to the original object to be destroyed when the `managed_ptr` itself is destroyed (stored as `void*`). Note that this pointer may be different from the "current" pointer due to conversions or aliasing.
- The pointer to the deleter (stored as `void*`).
- The pointer to a clean-up function that performs deletion using the original object pointer and the deleter pointer. This function is generated by instantiating a template function in the `managed_ptr`'s constructor so that this function is able to embed the knowledge of the original types of the object pointer and the deleter.

Although the `managed_ptr` is four words long, our experience shows that moving four words is significantly faster than the reference count change operation that would be performed when copying of a `shared_ptr`. Also the four word footprint of a `managed_ptr` instance is, in practice, much smaller than that of a `shared_ptr` instance.

Support for destructive copy semantics is implemented with the help of an auxiliary class `managed_ptr_ref` using a method analogous to that employed by `auto_ptr` [N1128R1].

The conversion from `shared_ptr` is implemented by treating the reference count structure as a special kind of deleter. That is, the reference count structure pointer is stored in the deleter pointer data member and a suitable clean-up function is generated to perform the same actions as the `shared_ptr`'s destructor.

The conversion to `shared_ptr` is implemented by housing a copy of the original `managed_ptr` in the `shared_ptr`'s reference count structure and invoking the `managed_ptr`'s destructor when the reference count (in the reference count structure) reaches zero. This conversion is supported via a `shared_ptr` constructor that takes a reference to a modifiable `managed_ptr`. Using a modifiable reference rather than a value in this constructor allows for the strong exception safety guarantee. Though modifiable references prevent rvalues from being passed to this constructor, once rvalue references are adopted, an additional constructor taking an explicit rvalue reference can be introduced to address this issue (see [N1377] for more information on rvalue references). The decision to pass `managed_ptr` by reference is consistent with how `auto_ptr` is passed to a `shared_ptr` constructor: `auto_ptr` is also passed by reference (also to provide the strong guarantee). For more information on construction of `shared_ptr` from `auto_ptr` see [N1450].

The conversion to `shared_ptr` from a managed_ptr that itself has been converted from a `shared_ptr` is treated as a special case. When this relatively common "reverse" conversion is performed, instead of constructing a new reference count structure housing

a copy of the original `manged_ptr`, the original reference count structure is recovered, thereby eliminating all runtime overhead.

The following section describes the amendments to the C++ standard for `managed_ptr`.

Proposed Text

Additions to header <memory> synopsis ([std] 2.4, [tr1] 2.2.1)

```
template<class T> class managed_ptr;
```

Class template *`managed_ptr`*

Template class `managed_ptr` stores a pointer to an object and a pointer to an object's ***manager.*** The ***manager*** is responsible for managing the lifetime of the object. When the `managed_ptr` is destroyed, `reclaim_managed_object` function is invoked with the manager pointer and the object pointer arguments, giving the ***manager*** an opportunity to destroy or, in some other way, reclaim the object (e.g., return the object into an object pool).

Template class `managed_ptr_ref` holds a reference to a `managed_ptr`. The `managed_ptr_ref` class is used by the `managed_ptr` conversions to allow rvalues of `managed_ptr` type to be passed to and returned from functions.

```
namespace std {
namespace tr1 {

      template<class T> class managed_ptr {
          template<class Y> struct managed_ptr_ref {};

  public:
      typedef T element_type;

      // constructors
      managed_ptr();
      template<class Y> explicit managed_ptr(Y *p);
      template<class Y, class M> managed_ptr(Y *p, M *m);
      managed_ptr(managed_ptr& r);
      template<class Y> managed_ptr(managed_ptr<Y>& r);
      template<class Y> managed_ptr(auto_ptr<Y>& r);
      template<class Y> managed_ptr(T *p, managed_ptr<Y>& r);

      // destructor
      ~managed_ptr();

      // assignment
      managed_ptr& operator =(managed_ptr& r);
      template<class Y> managed_ptr& operator =(managed_ptr<Y>& r);
      template<class Y> managed_ptr& operator =(auto_ptr<Y>& r);
```

```
      // modifiers
      void swap(managed_ptr& r);
      void reset();
      template<class Y> void reset(Y *p);
      template<class Y, class M> void reset(Y *p, M *m);
      template<class Y> void reset(T *p, managed_ptr<Y>& r);

      // observers
      T& operator *() const;
      T* operator ->() const;
      T* get() const;

      // conversions
      managed_ptr(managed_ptr_ref<T> r);
      template<class Y> operator managed_ptr_ref<Y>();
      template<class Y> operator managed_ptr<Y>();
};
```

managed_ptr constructors

managed_ptr();

Effects

Constructs an ***empty*** `managed_ptr`.

Postconditions

`get() == 0`.

Throws

Nothing

template<class Y> explicit managed_ptr(Y *p);

Requires

`P` is convertible to `T *`. `Y` is complete. The expression `delete p` shall be well-formed, shall have well-defined behavior, and shall not throw exceptions.

Effects

Constructs `managed_ptr` object that ***owns*** the pointer `p`.

Postconditions

`get() == p`

Throws

Nothing

```
template<class Y, class M> managed_ptr(Y *p, M *m);
```

Requires

`P` is convertible to `T *`. `Y` is complete. The expression `reclaim_managed_object(m, p)` shall be well-formed, shall have well-defined behavior, and shall not throw exceptions.

Effects

Constructs `managed_ptr` object that ***owns*** the pointer `p` and the manager `m`.

Postconditions

`get() == p`

Throws

Nothing

```
managed_ptr(managed_ptr& r);
```

Effects

If `r` is an ***empty*** `managed_ptr`, constructs an ***empty*** `managed_ptr`. Otherwise, makes `r` ***empty*** and constructs a `managed_ptr` that ***owns*** the pointer and, if `r` ***owned*** a manager, the manager that was previously ***owned*** by `r`.

Postconditions

`r.get() == 0`

Throws

Nothing

```
template<class Y> managed_ptr(managed_ptr<Y>& r);
```

Requires

`r.get()` is convertible to `T *`.

Effects

If `r` is an ***empty*** `managed_ptr`, constructs an ***empty*** `managed_ptr`. Otherwise, makes `r` ***empty*** and constructs a `managed_ptr` that ***owns*** the pointer and, if `r` ***owned*** a manager, the manager that was previously ***owned*** by `r`.

Postconditions

`r.get() == 0`

Throws

Nothing

```
template<class Y> managed_ptr(auto_ptr<Y>& r);
```

Requires

`r.get()` is convertible to `T *`.

Effects

If `r` is ***empty***, constructs an ***empty*** `managed_ptr`. Otherwise, calls `r.release()`and constructs a `managed_ptr` that ***owns*** the pointer returned by `release`.

Postconditions

`r.get() == 0`

Throws

Nothing

```
template<class Y> managed_ptr(managed_ptr<Y>& r, T *p);
```

Effects

Constructs a `managed_ptr` object that ***owns*** the pointer previously owned by `r`. `r` is made ***empty***.

Postconditions

`r.get() == 0 && get() == p`

Throws

Nothing

`managed_ptr` destructor

```
~managed_ptr();
```

Effects

No effect if `*this` ***is empty.*** Otherwise, if `*this` ***owns*** a pointer `p` and a manager `m`, `reclaim_managed_object(m, p)` is called. Otherwise, `delete p` is called.

Throws

Nothing

managed_ptr assignment

```
managed_ptr& operator =(managed_ptr& r);
```

Effects

Equivalent to `managed_ptr(r).swap(*this)`.

Throws

Nothing

```
template<class Y> managed_ptr& operator =(managed_ptr<Y>& r);
```

Effects

Equivalent to `managed_ptr(r).swap(*this)`.

Throws

Nothing

```
template<class Y> managed_ptr& operator =(auto_ptr<Y>& r);
```

Effects

Equivalent to `managed_ptr(r).swap(*this)`.

Throws

Nothing

managed_ptr modifiers

```
void swap(managed_ptr& r);
```

Effects

Exchanges contents of `*this` and `r`.

Throws

Nothing

```
void reset();
```

Effects

Equivalent to `managed_ptr().swap(*this)`.

Throws

Nothing

```
template<class Y> void reset(Y *p);
```

Effects

Equivalent to `managed_ptr(p).swap(*this).`

Throws

Nothing

```
template<class Y, class M> void reset(Y *p, M *m);
```

Effects

Equivalent to `managed_ptr(p, m).swap(*this).`

Throws

Nothing

```
template<class Y> void reset(managed_ptr<Y>& r, T *p);
```

Effects

Equivalent to `managed_ptr(r, p).swap(*this).`

Throws

Nothing

`managed_ptr` observers

```
T* get() const;
```

Returns

The stored pointer

Throws

Nothing

```
T* operator ->() const;
```

Returns

`get()`

Throws

Nothing

```
T& operator *() const;
```

Requires

`get() != 0`

Returns

`*get()`

Throws

Nothing

managed_ptr conversions

```
managed_ptr(managed_ptr_ref<T> r);
```

Effects

Constructs a `managed_ptr` object from the reference that `r` holds.

Throws

Nothing

```
template<class Y> operator managed_ptr_ref<Y>();
```

Returns

A `managed_ptr_ref<Y>` that holds `*this`.

Throws

Nothing

```
template<class Y> operator managed_ptr<Y>();
```

Returns

A `managed_ptr<Y>` that is constructed from `*this`.

Effects

Makes `*this` ***empty.***

Postconditions

`get() == 0`

Throws

Nothing

Additions to `shared_ptr` constructors ([tr1] 2.2.3.1)

```
template<class Y> shared_ptr(managed_ptr<Y>& r);
```

Requires

`r.get()` is convertible to `T *`.

Effects

Makes `r` ***empty*** and constructs a `shared_ptr` that ***owns*** the pointer previously ***owned*** by `r`.

Postconditions

`r.get() == 0`

Throws

`bad_alloc`, or an implementation-defined exception when a resource other than memory could not be obtained.

Exception safety

If an exception is thrown, `r` is made empty.

Additions to `shared_ptr` assignment ([tr1] 2.2.3.1)

```
template<class Y> shared_ptr& operator=(managed_ptr<Y>& r);
```

Effects

Equivalent to `shared_ptr (r).swap(*this)`.

Additions to `shared_ptr` observers ([tr1] 2.2.3.5)

```
template<class Y> operator managed_ptr<Y>() const;
```

Requires

`T *` is convertible to `Y *`.

Returns

If *this is ***empty***, an ***empty*** `managed_ptr<Y>`. Otherwise, an instance of `managed_ptr<Y>` that ***shares ownership*** with `*this`.

Postconditions

`get() == 0 || use_count() > 1`

Throws

Nothing

Conclusion

In this paper we have proposed several related features. The additional features are mostly independent and can be considered for inclusion into the standard individually. We hope that adopting these features will allow the greater C++ community to reap the benefits we have enjoyed achieving enhanced performance and clearer design on a variety of projects.

Appendix A. Performance comparison with boost::shared_ptr

Purpose

In this appendix we compare performance benchmarks of our shared pointer to Boost shared pointer for threaded builds on Sun and IBM/AIX platforms.

Summary Conclusions

We benchmarked Boost and our smart pointers on three platforms/configurations: (1) Sun, using `malloc`, (2) Sun using `SmartHeap`, and (3) IBM/AIX (with no observed difference for different allocation mechanisms). The overall results indicate that, when Boost and our implementation are used in exactly the same way to construct a shared pointer, there is no significant performance difference on Sun or IBM/AIX. However, our design is richer, supporting in-place storage (avoiding allocation) and effective use of pools. When either of these advantages can be exploited, our performance is between two and 39 times faster, depending on the specific benchmark and the platform. Examples include in-place payload storage (giving a two-fold improvement on Sun), in-place payload storage using a non-threaded pool (giving a four-fold improvement on Sun and AIX), and in-place payload storage using a threaded pool (giving a 39-fold improvement on AIX).

Benchmark Details

We have run benchmarks comparing the performance of our implementation of shared pointer and the Boost `shared_ptr` in several typical operations (e.g., basic construction), each in a program employing a single thread and a program employing two threads. In addition, our shared pointer tests were run using no pool, a non-threaded pool, and a threaded pool.

- Each test was performed in a loop of 5 million iterations.
- Each test was performed on Sun using `malloc`, on Sun using `SmartHeap`, and on IBM/AIX (there was no observed difference between allocation mechanisms on AIX).
- In all multi-threaded cases, two threads were used.
- In all cases, `BOOST_SP_USE_QUICK_ALLOCATOR` was defined prior to including boost `shared_ptr.h`.
- `BOOST_SP_ENABLE_DEBUG_HOOKS` was not defined for any of the tests.

The following sixteen test functions were run. The names of the test functions serve to document the test performed. For example, the function `boost_shared_ptr_Basic_ConstructorThreaded` uses the Boost shared pointer to benchmark the basic constructor in a program employing multiple threads.

- `boost_shared_ptr_Basic_Constructor`
- `boost_shared_ptr_Deleter_Constructor`

- `our_shared_ptr_Basic_Constructor`
- `our_shared_ptr_Deleter_Constructor`
- `our_shared_ptr_Inplace`
- `our_shared_ptr_Basic_Constructor_With_Pool`
- `our_shared_ptr_Inplace_With_ThreadedPool`
- `our_shared_ptr_Basic_Constructor_With_NonThreadedPool`
- `our_shared_ptr_Inplace_With_NonThreadedPool`
- `boost_shared_ptr_Basic_ConstructorThreaded`
- `our_shared_ptr_Basic_ConstructorThreaded`
- `our_shared_ptr_Inplace_With_ThreadedPoolThreaded`

Section “Graphical Summary” gives a graphical summary of the results on Sun using `SmartHeap`, which is the least advantageous for our implementation. Still, our implementation has clear advantages for all constructor tests that take advantage of in-place construction and/or pools.

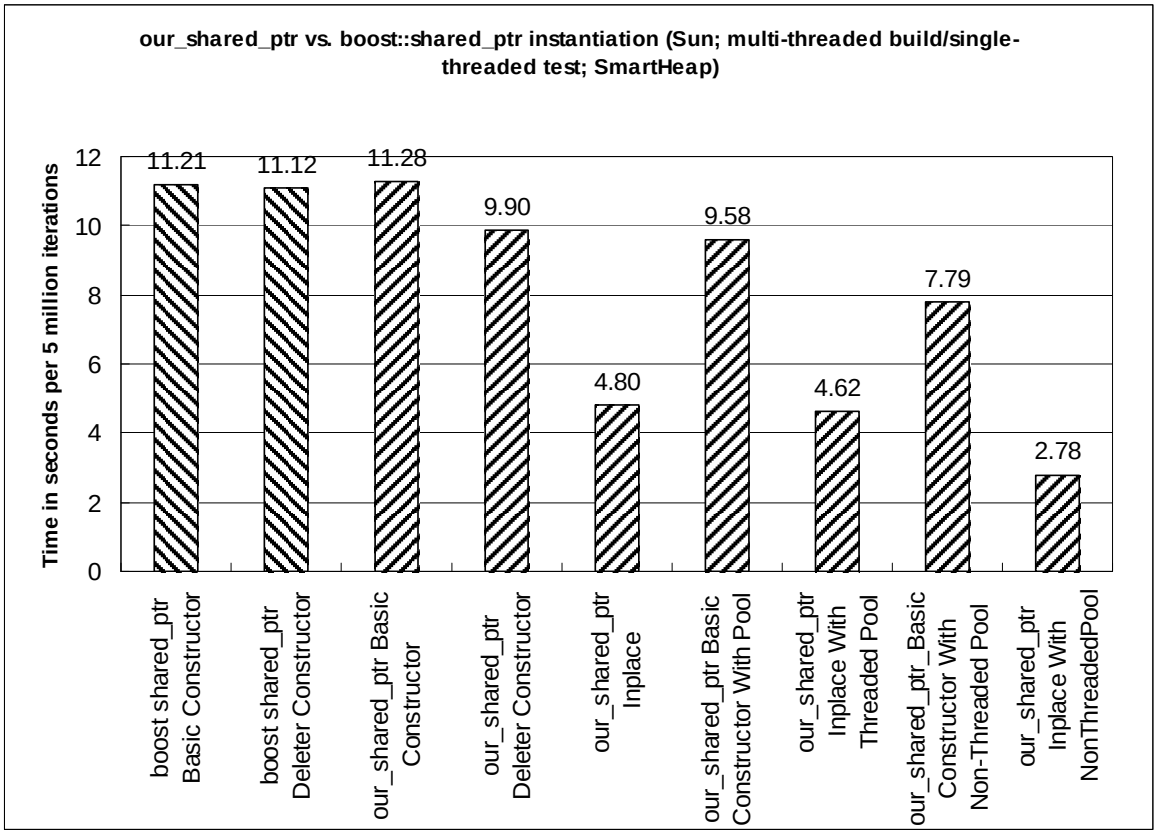
Section “Results” presents the full tabular results, expressed as execution times in seconds, for the sixteen tests on three platforms/allocation configurations.

Graphical Summary

In this section we present the graphical results of the benchmarks discussed in the main text. In all cases, the results are obtained on Sun using `SmartHeap`, which is the least favorable configuration for our implementation with respect to Boost.

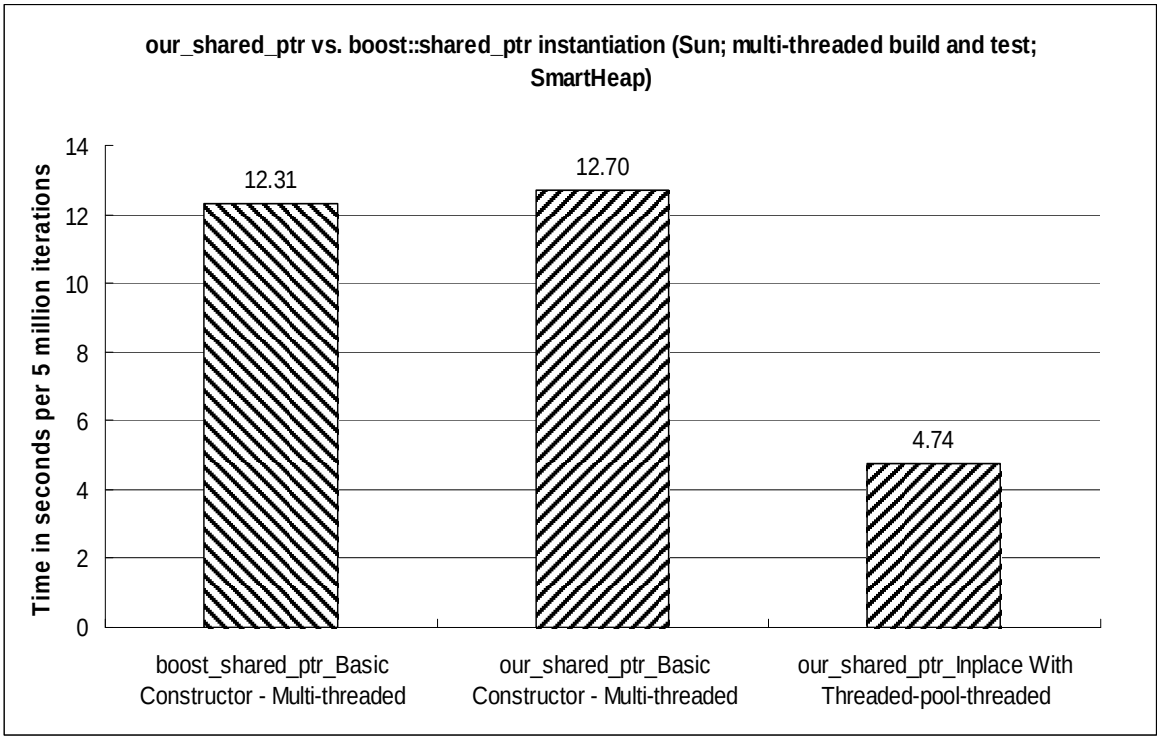
Pointer Instantiation, Applications Running a Single Thread

In applications running a single thread, our implementation's performance is comparable when not using in-place storage or pools. However, either for in-place storage or when using pools, performance can be as much as four times better.



Pointer Instantiation, Applications Running Two Threads

In applications running two threads, our implementation performance is comparable when not using in-place storage or pools. However, for in-place storage and when using pools, performance is 2.6 times better.



Results

In this section we present the full tabulated results of the benchmarks discussed in the main text.

Test	Average Execution Time (sec)		
	Sun, malloc	**Sun, SmartHeap**	**AIX**
Boost shared_ptr Basic Constructor	9.52	11.212	7.37
Boost shared_ptr Deleter Constructor	9.44	11.116	7.43
Our shared_ptr Basic Constructor	10.18	11.276	4.50
Our shared_ptr Deleter Constructor	9.80	9.897	4.21
Our shared_ptr Inplace	4.87	4.798	4.78
Our shared_ptr Basic Constructor With Pool	8.89	9.579	9.99
Our shared_ptr Inplace With Threaded Pool	4.48	4.623	6.04
Our shared_ptr Basic Constructor With Non-Threaded Pool	5.63	7.786	5.71
Our shared_ptr Inplace With NonThreadedPool	2.49	2.781	2.02
Boost shared_ptr Basic Constructor – Multi-threaded	57.96	12.314	321.15
Our shared_ptr Basic Constructor - Multi-threaded	56.71	12.705	199.95
Our shared_ptr Inplace With Threaded-pool-threaded	4.57	4.745	8.17

Appendix B. Example

The following sample code illustrates both the motivation for and the use of the proposed features. While this small example is deliberately constructed to employ all of the features, it is, in our experience, a realistic demonstration of how these features interact.

```
class HighPerformanceThreadSafeAllocator;
class WorkItem;

class AsynchProcessor {
    void processWorkItem(managed_ptr<const WorkItem> workItem);
                         // Process work item in a separate thread.
                         // Note use of the managed_ptr in the
                         // interface: AsynchProcessor requires
                         // ownership of the item but does not make
                         // any assumptions about sharing this
                         // ownership with the caller.

};

HighPerformanceThreadSafeAllocator alloc;
AsynchProcessor                    processor

// performance critical item processing loop
while (keepGoing)
{
     shared_ptr<vector<WorkItem> > itemsPtr;
                                // must be std::vector<WorkItem>
                                // for performance and
                                // "wire format compatibility"

     itemsPtr =
             make_shared_object_with_alloc< vector<WorkItem> >(alloc);
                                // housing a vector in the shared_ptr
                                // while also using the allocator to
                                // optimize memory allocation
                                // performance

     int numWorkItems;
     workStream >> numWorkItems;
     itemsPtr->reserve(numWorkItems);
     ReadWorkItems(workStream, *itemsPtr, numWorkItems);

     for (vector<WorkItem>::iterator i = itemsPtr->begin();
          i!=itemsPtr->end(); ++i)
     {
        shared_ptr<WorkItem> item(itemsPtr, &*i); // aliasing
        processor.processWorkItem(item); // share to managed conversion
     }
     // The vector held by itemsPtr is destroyed and deallocated when
     // when the last WorkItem has been processed asynchronously (and,
     // possibly, out of order).
}
```

References

[N1128R1] Bill Gibbons, Greg Colvin, ***Fixing auto_ptr***, C++ committee document J16/97-0090R1= WG21/N1128R1, November 1997

[N1377] Howard E. Hinnant, Peter Dimov, Dave Abrahams, ***A Proposal to Add Move Semantics Support to the C++ Language,*** C++ committee document N1377=02-0035, September 2002

[N1450] Peter Dimov, Beman Dawes, Greg Colvin, ***A Proposal to Add General Purpose Smart Pointers to the Library Technical Report,*** C++ committee document N1450=03-0033, 2003

[N1809] Library Extension Technical Report – Issues List, Revision 9: post-Lillehammer mailing, C++ committee document N1809=05-0069, April 2005