shared_ptr class template

Introduction

The `shared_ptr` class template stores a pointer to a dynamically allocated object, typically with a C++ *new-expression*. The object pointed to is guaranteed to be deleted when the last `shared_ptr` pointing to it is destroyed or reset.

> *Example:*
>
> ```
> shared_ptr<X> p1( new X );
> shared_ptr<void> p2( new int(5) );
> ```

`shared_ptr` deletes the exact pointer that has been passed at construction time, complete with its original type, regardless of the template parameter. In the second example above, when `p2` is destroyed or reset, it will call `delete` on the original `int*` that has been passed to the constructor, even though `p2` itself is of type `shared_ptr<void>` and stores a pointer of type `void*`.

Every `shared_ptr` meets the `CopyConstructible`, `MoveConstructible`, `CopyAssignable` and `MoveAssignable` requirements of the C++ Standard Library, and can be used in standard library containers. Comparison operators are supplied so that `shared_ptr` works with the standard library's associative containers.

Because the implementation uses reference counting, cycles of `shared_ptr` instances will not be reclaimed. For example, if `main()` holds a `shared_ptr` to `A`, which directly or indirectly holds a `shared_ptr` back to `A`, `A`'s use count will be 2. Destruction of the original `shared_ptr` will leave `A` dangling with a use count of 1. Use [weak_ptr] to "break cycles."

The class template is parameterized on `T`, the type of the object pointed to. `shared_ptr` and most of its member functions place no requirements on `T`; it is allowed to be an incomplete type, or `void`. Member functions that do place additional requirements (constructors, reset) are explicitly documented below.

`shared_ptr<T>` can be implicitly converted to `shared_ptr<U>` whenever `T*` can be implicitly converted to `U*`. In particular, `shared_ptr<T>` is implicitly convertible to `shared_ptr<T const>`, to `shared_ptr<U>` where `U` is an accessible base of `T`, and to `shared_ptr<void>`.

`shared_ptr` is now part of the C++11 Standard, as `std::shared_ptr`.

Starting with Boost release 1.53, `shared_ptr` can be used to hold a pointer to a dynamically allocated array. This is accomplished by using an array type (`T[]` or `T[N]`) as the template parameter. There is almost no difference between using an unsized array, `T[]`, and a sized array, `T[N]`; the latter just enables `operator[]` to perform a range check on the index.

Example:

```
shared_ptr<double[1024]> p1( new double[1024] );
shared_ptr<double[]> p2( new double[n] );
```

Best Practices

A simple guideline that nearly eliminates the possibility of memory leaks is: always use a named smart pointer variable to hold the result of `new`. Every occurence of the `new` keyword in the code should have the form:

```
shared_ptr<T> p(new Y);
```

It is, of course, acceptable to use another smart pointer in place of `shared_ptr` above; having `T` and `Y` be the same type, or passing arguments to `Y`'s constructor is also OK.

If you observe this guideline, it naturally follows that you will have no explicit `delete` statements; `try/catch` constructs will be rare.

Avoid using unnamed `shared_ptr` temporaries to save typing; to see why this is dangerous, consider this example:

```
void f(shared_ptr<int>, int);
int g();

void ok()
{
    shared_ptr<int> p( new int(2) );
    f( p, g() );
}

void bad()
{
    f( shared_ptr<int>( new int(2) ), g() );
}
```

The function `ok` follows the guideline to the letter, whereas `bad` constructs the temporary `shared_ptr` in place, admitting the possibility of a memory leak. Since function arguments are evaluated in unspecified order, it is possible for `new int(2)` to be evaluated first, `g()` second, and we may never get to the `shared_ptr`constructor if `g` throws an exception. See Herb Sutter's treatment (also here) of the issue for more information.

The exception safety problem described above may also be eliminated by using the `make_shared` or `allocate_shared` factory functions defined in `boost/make_shared.hpp`. These factory functions also provide an efficiency benefit by consolidating allocations.

Synopsis

```
namespace boost {

  class bad_weak_ptr: public std::exception;

  template<class T> class weak_ptr;

  template<class T> class shared_ptr {

    public:

      typedef see below element_type;

      shared_ptr(); // never throws
      shared_ptr(std::nullptr_t); // never throws

      template<class Y> explicit shared_ptr(Y * p);
      template<class Y, class D> shared_ptr(Y * p, D d);
      template<class Y, class D, class A> shared_ptr(Y * p, D d, A a);
      template<class D> shared_ptr(std::nullptr_t p, D d);
      template<class D, class A> shared_ptr(std::nullptr_t p, D d, A a);

      ~shared_ptr(); // never throws

      shared_ptr(shared_ptr const & r); // never throws
      template<class Y> shared_ptr(shared_ptr<Y> const & r); // never throws

      shared_ptr(shared_ptr && r); // never throws
      template<class Y> shared_ptr(shared_ptr<Y> && r); // never throws

      template<class Y> shared_ptr(shared_ptr<Y> const & r, element_type * p); // never throws

      template<class Y> explicit shared_ptr(weak_ptr<Y> const & r);

      template<class Y> explicit shared_ptr(std::auto_ptr<Y> & r);
      template<class Y> shared_ptr(std::auto_ptr<Y> && r);

      template<class Y, class D> shared_ptr(std::unique_ptr<Y, D> && r);

      shared_ptr & operator=(shared_ptr const & r); // never throws
      template<class Y> shared_ptr & operator=(shared_ptr<Y> const & r); // never throws

      shared_ptr & operator=(shared_ptr const && r); // never throws
      template<class Y> shared_ptr & operator=(shared_ptr<Y> const && r); // never throws


      template<class Y> shared_ptr & operator=(std::auto_ptr<Y> & r);
      template<class Y> shared_ptr & operator=(std::auto_ptr<Y> && r);

      template<class Y, class D> shared_ptr & operator=(std::unique_ptr<Y, D> && r);

      shared_ptr & operator=(std::nullptr_t); // never throws

      void reset(); // never throws

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

```
    template<class Y> void reset(shared_ptr<Y> const & r, element_type * p); // never throws

    T & operator*() const; // never throws; only valid when T is not an array type
    T * operator->() const; // never throws; only valid when T is not an array type

    element_type & operator[](std::ptrdiff_t i) const; // never throws; only valid when T is an ar

    element_type * get() const; // never throws

    bool unique() const; // never throws
    long use_count() const; // never throws

    explicit operator bool() const; // never throws

    void swap(shared_ptr & b); // never throws

    template<class Y> bool owner_before(shared_ptr<Y> const & rhs) const; // never throws
    template<class Y> bool owner_before(weak_ptr<Y> const & rhs) const; // never throws
  };

  template<class T, class U>
    bool operator==(shared_ptr<T> const & a, shared_ptr<U> const & b); // never throws

  template<class T, class U>
    bool operator!=(shared_ptr<T> const & a, shared_ptr<U> const & b); // never throws

  template<class T, class U>
    bool operator<(shared_ptr<T> const & a, shared_ptr<U> const & b); // never throws

  template<class T>
    bool operator==(shared_ptr<T> const & p, std::nullptr_t); // never throws

  template<class T>
    bool operator==(std::nullptr_t, shared_ptr<T> const & p); // never throws

  template<class T>
    bool operator!=(shared_ptr<T> const & p, std::nullptr_t); // never throws

  template<class T>
    bool operator!=(std::nullptr_t, shared_ptr<T> const & p); // never throws

  template<class T> void swap(shared_ptr<T> & a, shared_ptr<T> & b); // never throws

  template<class T> typename shared_ptr<T>::element_type * get_pointer(shared_ptr<T> const & p); //

  template<class T, class U>
    shared_ptr<T> static_pointer_cast(shared_ptr<U> const & r); // never throws

  template<class T, class U>
    shared_ptr<T> const_pointer_cast(shared_ptr<U> const & r); // never throws

  template<class T, class U>
    shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const & r); // never throws

  template<class T, class U>
    shared_ptr<T> reinterpet_pointer_cast(shared_ptr<U> const & r); // never throws


  template<class E, class T, class Y>
    std::basic_ostream<E, T> & operator<< (std::basic_ostream<E, T> & os, shared_ptr<Y> const & p);

  template<class D, class T>
    D * get_deleter(shared_ptr<T> const & p);
}
```

Members

element_type

```
typedef ... element_type;
```

`element_type` is `T` when `T` is not an array type, and `U` when `T` is `U[]` or `U[N]`.

default constructor

```
shared_ptr(); // never throws
shared_ptr(std::nullptr_t); // never throws
```

Effects: Constructs an *empty* `shared_ptr`.

Postconditions: `use_count() == 0 && get() == 0`.

Throws: nothing.

[The nothrow guarantee is important, since `reset()` *is specified in terms of the default constructor; this implies that the constructor must not allocate memory.]*

pointer constructor

```
template<class Y> explicit shared_ptr(Y * p);
```

Requirements: `Y` must be a complete type. The expression `delete[] p`, when `T` is an array type, or `delete p`, when `T` is not an array type, must be well-formed, must not invoke undefined behavior, and must not throw exceptions. When `T` is `U[N]`, `Y (*) [N]` must be convertible to `T*`; when `T` is `U[]`, `Y (*) []` must be convertible to `T*`; otherwise, `Y*` must be convertible to `T*`.

Effects: When `T` is not an array type, constructs a `shared_ptr` that *owns* the pointer `p`. Otherwise, constructs a `shared_ptr` that *owns* `p` and a deleter of an unspecified type that calls `delete[] p`.

Postconditions: `use_count() == 1 && get() == p`. If `T` is not an array type and `p` is unambiguously convertible to `enable_shared_from_this<V>*` for some `V`, `p->shared_from_this()` returns a copy of `*this`.

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

Exception safety: If an exception is thrown, the constructor calls `delete[] p`, when `T` is an array type, or `delete p`, when `T` is not an array type.

Notes: `p` must be a pointer to an object that was allocated via a C++ `new` expression or be 0. The postcondition that use count is 1 holds even if `p` is 0; invoking `delete` on a pointer that has a value of 0 is harmless.

[This constructor is a template in order to remember the actual pointer type passed. The destructor will call `delete` *with the same pointer, complete with its original type, even when* `T` *does not have a virtual destructor, or is* `void`*.]*

constructors taking a deleter

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

Requirements: `D` must be `CopyConstructible`. The copy constructor and destructor of `D` must not throw. The expression `d(p)` must be well-formed, must not invoke undefined behavior, and must not throw exceptions. `A` must be an *Allocator*, as described in section 20.1.5 (`Allocator requirements`) of the C++ Standard. When `T` is `U[N]`, `Y (*) [N]` must be convertible to `T*`; when `T` is `U[]`, `Y (*) []` must be convertible to `T*`; otherwise, `Y*` must be convertible to `T*`.

Effects: Constructs a `shared_ptr` that *owns* the pointer `p` and the deleter `d`. The constructors taking an allocator `a` allocate memory using a copy of `a`.

Postconditions: `use_count() == 1 && get() == p`. If `T` is not an array type and `p` is unambiguously convertible to `enable_shared_from_this<V>*` for some `V`, `p->shared_from_this()` returns a copy of `*this`.

Throws: `std::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.

Notes: When the the time comes to delete the object pointed to by `p`, the stored copy of `d` is invoked with the stored copy of `p` as an argument.

[Custom deallocators allow a factory function returning a `shared_ptr` *to insulate the user from its memory allocation strategy. Since the deallocator is not part of the type, changing the allocation strategy does not break source or binary compatibility, and does not require a client recompilation. For example, a "no-op" deallocator is useful when returning a* `shared_ptr` *to a statically allocated object, and other variations allow a* `shared_ptr` *to be used as a wrapper for another smart pointer, easing interoperability.*

The support for custom deallocators does not impose significant overhead. Other `shared_ptr` *features still require a deallocator to be kept.*

The requirement that the copy constructor of `D` *does not throw comes from the pass by value. If the copy constructor throws, the pointer would leak.]*

copy and converting constructors

```
shared_ptr(shared_ptr const & r); // never throws
template<class Y> shared_ptr(shared_ptr<Y> const & r); // never throws
```

Requires: `Y*` should be convertible to `T*`.

Effects: If `r` is *empty*, constructs an *empty* `shared_ptr`; otherwise, constructs a `shared_ptr` that *shares ownership* with `r`.

Postconditions: `get() == r.get() && use_count() == r.use_count()`.

Throws: nothing.

move constructors

```
shared_ptr(shared_ptr && r); // never throws
template<class Y> shared_ptr(shared_ptr<Y> && r); // never throws
```

Requires: `Y*` should be convertible to `T*`.

Effects: Move-constructs a `shared_ptr` from `r`.

Postconditions: `*this` contains the old value of `r`. `r` is *empty* and `r.get() == 0`.

Throws: nothing.

aliasing constructor

```
template<class Y> shared_ptr(shared_ptr<Y> const & r, element_type * p); // never throws
```

Effects: constructs a `shared_ptr` that *shares ownership* with `r` and stores `p`.

Postconditions: `get() == p && use_count() == r.use_count()`.

Throws: nothing.

weak_ptr constructor

```
template<class Y> explicit shared_ptr(weak_ptr<Y> const & r);
```

Requires: `Y*` should be convertible to `T*`.

Effects: Constructs a `shared_ptr` that *shares ownership* with `r` and stores a copy of the pointer stored in `r`.

Postconditions: `use_count() == r.use_count()`.

Throws: `bad_weak_ptr` when `r.use_count() == 0`.

Exception safety: If an exception is thrown, the constructor has no effect.

auto_ptr constructors

```
template<class Y> shared_ptr(std::auto_ptr<Y> & r);
template<class Y> shared_ptr(std::auto_ptr<Y> && r);
```

Requires: `Y*` should be convertible to `T*`.

Effects: Constructs a `shared_ptr`, as if by storing a copy of `r.release()`.

Postconditions: `use_count() == 1`.

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

Exception safety: If an exception is thrown, the constructor has no effect.

unique_ptr constructor

```
template<class Y, class D> shared_ptr(std::unique_ptr<Y, D> && r);
```

Requires: `Y*` should be convertible to `T*`.

Effects: Equivalent to `shared_ptr(r.release(), r.get_deleter())` when `D` is not a reference type. Otherwise, equivalent to `shared_ptr(r.release(), del)`, where *del* is a deleter that stores the reference `rd` returned from `r.get_deleter()` and `del(p)` calls `rd(p)`.

Postconditions: `use_count() == 1`.

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

Exception safety: If an exception is thrown, the constructor has no effect.

destructor

```
~shared_ptr(); // never throws
```

Effects:

- If `*this` is *empty*, or *shares ownership* with another `shared_ptr` instance (`use_count() > 1`), there are no side effects.
- Otherwise, if `*this` *owns* a pointer `p` and a deleter `d`, `d(p)` is called.
- Otherwise, `*this` *owns* a pointer `p`, and `delete p` is called.

Throws: nothing.

assignment

```
shared_ptr & operator=(shared_ptr const & r); // never throws
template<class Y> shared_ptr & operator=(shared_ptr<Y> const & r); // never throws
template<class Y> shared_ptr & operator=(std::auto_ptr<Y> & r);
```

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

Returns: `*this`.

Notes: The use count updates caused by the temporary object construction and destruction are not considered observable side effects, and the implementation is free to meet the effects (and the implied guarantees) via different means, without creating a temporary. In particular, in the example:

```
shared_ptr<int> p(new int);
shared_ptr<void> q(p);
p = p;
q = p;
```

both assignments may be no-ops.

```
shared_ptr & operator=(shared_ptr && r); // never throws
template<class Y> shared_ptr & operator=(shared_ptr<Y> && r); // never throws
template<class Y> shared_ptr & operator=(std::auto_ptr<Y> && r);
template<class Y, class D> shared_ptr & operator=(std::unique_ptr<Y, D> && r);
```

Effects: Equivalent to `shared_ptr(std::move(r)).swap(*this)`.

Returns: `*this`.

```
shared_ptr & operator=(std::nullptr_t); // never throws
```

Effects: Equivalent to `shared_ptr().swap(*this)`.

Returns: `*this`.

reset

```
void reset(); // never throws
```

Effects: Equivalent to `shared_ptr().swap(*this)`.

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

Effects: Equivalent to `shared_ptr(p).swap(*this)`.

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

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

```
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> void reset(shared_ptr<Y> const & r, element_type * p); // never throws
```

Effects: Equivalent to `shared_ptr(r, p).swap(*this)`.

indirection

```
T & operator*() const; // never throws
```

Requirements: `T` should not be an array type. The stored pointer must not be 0.

Returns: a reference to the object pointed to by the stored pointer.

Throws: nothing.

```
T * operator->() const; // never throws
```

Requirements: `T` should not be an array type. The stored pointer must not be 0.

Returns: the stored pointer.

Throws: nothing.

```
element_type & operator[](std::ptrdiff_t i) const; // never throws
```

Requirements: `T` should be an array type. The stored pointer must not be 0. `i >= 0`. If `T` is `U[N]`, `i < N`.

Returns: `get()[i]`.

Throws: nothing.

get

```
element_type * get() const; // never throws
```

Returns: the stored pointer.

Throws: nothing.

unique

```
bool unique() const; // never throws
```

Returns: `use_count() == 1`.

Throws: nothing.

Notes: `unique()` may be faster than `use_count()`. If you are using `unique()` to implement copy on write, do not rely on a specific value when the stored pointer is zero.

use_count

```
long use_count() const; // never throws
```

Returns: the number of `shared_ptr` objects, `*this` included, that *share ownership* with `*this`, or 0 when `*this` is *empty*.

Throws: nothing.

Notes: `use_count()` is not necessarily efficient. Use only for debugging and testing purposes, not for production code.

conversions

```
explicit operator bool() const; // never throws
```

Returns: `get() != 0`.

Throws: nothing.

Notes: This conversion operator allows `shared_ptr` objects to be used in boolean contexts, like `if(p && p->valid()) {}`.

[The conversion to bool is not merely syntactic sugar. It allows `shared_ptr`*s to be declared in conditions when using [dynamic_pointer_cast] or [weak_ptr::lock].]*

swap

```
void swap(shared_ptr & b); // never throws
```

Effects: Exchanges the contents of the two smart pointers.

Throws: nothing.

swap

```
template<class Y> bool owner_before(shared_ptr<Y> const & rhs) const; // never throws
template<class Y> bool owner_before(weak_ptr<Y> const & rhs) const; // never throws
```

Effects: See the description of `operator<`.

Throws: nothing.

Free Functions

comparison

```
template<class T, class U>
  bool operator==(shared_ptr<T> const & a, shared_ptr<U> const & b); // never throws
```

Returns: `a.get() == b.get()`.

Throws: nothing.

```
template<class T, class U>
  bool operator!=(shared_ptr<T> const & a, shared_ptr<U> const & b); // never throws
```

Returns: `a.get() != b.get()`.

Throws: nothing.

```
template<class T>
  bool operator==(shared_ptr<T> const & p, std::nullptr_t); // never throws
template<class T>
  bool operator==(std::nullptr_t, shared_ptr<T> const & p); // never throws
```

Returns: `p.get() == 0`.

Throws: nothing.

```
template<class T>
  bool operator!=(shared_ptr<T> const & p, std::nullptr_t); // never throws
template<class T>
  bool operator!=(std::nullptr_t, shared_ptr<T> const & p); // never throws
```

Returns: `p.get() != 0`.

Throws: nothing.

```
template<class T, class U>
  bool operator<(shared_ptr<T> const & a, shared_ptr<U> const & b); // never throws
```

Returns: an unspecified value such that

- `operator<` is a strict weak ordering as described in section 25.3 `[lib.alg.sorting]` of the C++ standard;
- under the equivalence relation defined by `operator<`, `!(a < b) && !(b < a)`, two `shared_ptr` instances are equivalent if and only if they *share ownership* or are both *empty*.

Throws: nothing.

Notes: Allows `shared_ptr` objects to be used as keys in associative containers.

[`Operator<` has been preferred over a `std::less` specialization for consistency and legality reasons, as `std::less` is required to return the results of `operator<`, and many standard algorithms use `operator<` instead of `std::less` for comparisons when a predicate is not supplied. Composite objects, like `std::pair`, also implement their `operator<` in terms of their contained subobjects' `operator<`.

The rest of the comparison operators are omitted by design.]

swap

```
template<class T>
  void swap(shared_ptr<T> & a, shared_ptr<T> & b); // never throws
```

Effects: Equivalent to `a.swap(b)`.

Throws: nothing.

Notes: Matches the interface of `std::swap`. Provided as an aid to generic programming.

[`swap` is defined in the same namespace as `shared_ptr` as this is currently the only legal way to supply a `swap` function that has a chance to be used by the standard library.]

get_pointer

```
template<class T>
  typename shared_ptr<T>::element_type * get_pointer(shared_ptr<T> const & p); // never throws
```

Returns: `p.get()`.

Throws: nothing.

Notes: Provided as an aid to generic programming. Used by mem_fn.

static_pointer_cast

```
template<class T, class U>
  shared_ptr<T> static_pointer_cast(shared_ptr<U> const & r); // never throws
```

Requires: The expression `static_cast<T*>( (U*)0 )` must be well-formed.

Returns: `shared_ptr<T>( r, static_cast<typename shared_ptr<T>::element_type*>(r.get()) )`.

Throws: nothing.

Notes: the seemingly equivalent expression `shared_ptr<T>(static_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice.

const_pointer_cast

```
template<class T, class U>
  shared_ptr<T> const_pointer_cast(shared_ptr<U> const & r); // never throws
```

Requires: The expression `const_cast<T*>( (U*)0 )` must be well-formed.

Returns: `shared_ptr<T>( r, const_cast<typename shared_ptr<T>::element_type*>(r.get()) )`.

Throws: nothing.

dynamic_pointer_cast

```
template<class T, class U>
  shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const & r);
```

Requires: The expression `dynamic_cast<T*>( (U*)0 )` must be well-formed.

Returns:

- When `dynamic_cast<typename shared_ptr<T>::element_type*>(r.get())` returns a nonzero value `p`, `shared_ptr<T>(r, p)`;
- Otherwise, `shared_ptr<T>()`.

Throws: nothing.

reinterpret_pointer_cast

```
template<class T, class U>
  shared_ptr<T> reinterpret_pointer_cast(shared_ptr<U> const & r); // never throws
```

Requires: The expression `reinterpret_cast<T*>( (U*)0 )` must be well-formed.

Returns: `shared_ptr<T>( r, reinterpret_cast<typename shared_ptr<T>::element_type*>(r.get()) )`.

Throws: nothing.

operator<<

```
template<class E, class T, class Y>
    std::basic_ostream<E, T> & operator<< (std::basic_ostream<E, T> & os, shared_ptr<Y> const & p);
```

Effects: `os << p.get();`.

Returns: `os`.

get_deleter

```
template<class D, class T>
    D * get_deleter(shared_ptr<T> const & p);
```

Returns: If `*this` *owns* a deleter `d` of type (cv-unqualified) `D`, returns `&d`; otherwise returns 0.

Throws: nothing.

Example

See shared_ptr_example.cpp for a complete example program. The program builds a `std::vector` and `std::set` of `shared_ptr` objects.

Note that after the containers have been populated, some of the `shared_ptr` objects will have a use count of 1 rather than a use count of 2, since the set is a `std::set` rather than a `std::multiset`, and thus does not contain duplicate entries. Furthermore, the use count may be even higher at various times while `push_back` and `insert` container operations are performed. More complicated yet, the container operations may throw exceptions under a variety of circumstances. Getting the memory management and exception handling in this example right without a smart pointer would be a nightmare.

Handle/Body Idiom

One common usage of `shared_ptr` is to implement a handle/body (also called pimpl) idiom which avoids exposing the body (implementation) in the header file.

The shared_ptr_example2_test.cpp sample program includes a header file, shared_ptr_example2.hpp, which uses a `shared_ptr` to an incomplete type to hide the implementation. The instantiation of member functions which require a complete type occurs in the shared_ptr_example2.cpp implementation file. Note that there is no need for an explicit destructor. Unlike `~scoped_ptr`, `~shared_ptr` does not require that `T` be a complete type.

Thread Safety

`shared_ptr` objects offer the same level of thread safety as built-in types. A `shared_ptr` instance can be "read" (accessed using only const operations) simultaneously by multiple threads. Different `shared_ptr` instances can be "written to" (accessed using mutable operations such as `operator=` or `reset`) simultaneously by multiple threads (even when these instances are copies, and share the same reference count underneath.)

Any other simultaneous accesses result in undefined behavior.

Examples:

```
shared_ptr<int> p(new int(42));

//--- Example 1 ---

// thread A
shared_ptr<int> p2(p); // reads p

// thread B
shared_ptr<int> p3(p); // OK, multiple reads are safe

//--- Example 2 ---

// thread A
p.reset(new int(1912)); // writes p

// thread B
p2.reset(); // OK, writes p2
```

```
//--- Example 3 ---

// thread A
p = p3; // reads p3, writes p

// thread B
p3.reset(); // writes p3; undefined, simultaneous read/write

//--- Example 4 ---

// thread A
p3 = p2; // reads p2, writes p3

// thread B
// p2 goes out of scope: undefined, the destructor is considered a "write access"

//--- Example 5 ---

// thread A
p3.reset(new int(1));

// thread B
p3.reset(new int(2)); // undefined, multiple writes
```

Starting with Boost release 1.33.0, `shared_ptr` uses a lock-free implementation on most common platforms.

If your program is single-threaded and does not link to any libraries that might have used `shared_ptr` in its default configuration, you can `#define` the macro `BOOST_SP_DISABLE_THREADS` on a project-wide basis to switch to ordinary non-atomic reference count updates.

(Defining `BOOST_SP_DISABLE_THREADS` in some, but not all, translation units is technically a violation of the One Definition Rule and undefined behavior. Nevertheless, the implementation attempts to do its best to accommodate the request to use non-atomic updates in those translation units. No guarantees, though.)

You can define the macro `BOOST_SP_USE_PTHREADS` to turn off the lock-free platform-specific implementation and fall back to the generic `pthread_mutex_t`-based code.

Frequently Asked Questions

Q. There are several variations of shared pointers, with different tradeoffs; why does the smart pointer library supply only a single implementation? It would be useful to be able to experiment with each type so as to find the most suitable for the job at hand?

A. An important goal of `shared_ptr` is to provide a standard shared-ownership pointer. Having a single pointer type is important for stable library interfaces, since different shared pointers typically cannot interoperate, i.e. a reference counted pointer (used by library A) cannot share ownership with a linked pointer (used by library B.)

Q. Why doesn't `shared_ptr` have template parameters supplying traits or policies to allow extensive user customization?

A. Parameterization discourages users. The `shared_ptr` template is carefully crafted to meet common needs without extensive parameterization. Some day a highly configurable smart pointer may be invented that is also very easy to use and very hard to misuse. Until then, `shared_ptr` is the smart pointer of choice for a wide range of applications. (Those interested in policy based smart pointers should read [Modern C++ Design](#) by Andrei Alexandrescu.)

Q. I am not convinced. Default parameters can be used where appropriate to hide the complexity. Again, why not policies?

A. Template parameters affect the type. See the answer to the first question above.

Q. Why doesn't `shared_ptr` use a linked list implementation?

A. A linked list implementation does not offer enough advantages to offset the added cost of an extra pointer. See [timings](#) page. In addition, it is expensive to make a linked list implementation thread safe.

Q. Why doesn't `shared_ptr` (or any of the other Boost smart pointers) supply an automatic conversion to `T*`?

A. Automatic conversion is believed to be too error prone.

Q. Why does `shared_ptr` supply `use_count()`?

A. As an aid to writing test cases and debugging displays. One of the progenitors had `use_count()`, and it was useful in tracking down bugs in a complex project that turned out to have cyclic-dependencies.

Q. Why doesn't `shared_ptr` specify complexity requirements?

A. Because complexity requirements limit implementors and complicate the specification without apparent benefit to `shared_ptr` users. For example, error-checking implementations might become non-conforming if they had to meet stringent complexity requirements.

Q. Why doesn't `shared_ptr` provide a `release()` function?

A. `shared_ptr` cannot give away ownership unless it's `unique()` because the other copy will still destroy the object.

Consider:

```
shared_ptr<int> a(new int);
shared_ptr<int> b(a); // a.use_count() == b.use_count() == 2

int * p = a.release();

// Who owns p now? b will still call delete on it in its destructor.
```

Furthermore, the pointer returned by `release()` would be difficult to deallocate reliably, as the source `shared_ptr` could have been created with a custom deleter.

Q. Why is `operator->()` const, but its return value is a non-const pointer to the element type?

A. Shallow copy pointers, including raw pointers, typically don't propagate constness. It makes little sense for them to do so, as you can always obtain a non-const pointer from a const one and then proceed to modify the object through it. `shared_ptr` is "as close to raw pointers as possible but no closer".

$Date$