The Boost Smart Pointer Library

Thomas Becker
Zephyr Associates, Inc.
BoostCon 2009

Overview

First two sentences of the Boost Smart Pointer Library documentation:

Smart pointers are objects which store pointers to dynamically allocated (heap) objects.

They behave much like built-in C++ pointers except that they automatically delete the object pointed to at the appropriate time.

Contents of the Boost Smart Pointer Library

Contents of the Boost smart pointer library (slightly augmented):

shared_ptr	Object ownership shared among multiple pointers
shared_array	Array ownership shared among multiple pointers
scoped_ptr	Simple sole ownership of single objects (non-copyable)
scoped_array	Simple sole ownership of arrays (non-copyable)
weak_ptr	Non-owning observers of an object owned by shared_ptr
intrusive_ptr	Shared ownership of objects with an embedded reference count
std:: auto_ptr (soon-to-be unique_ptr)	transferrable unique ownership

A Bit of History (Do Not Repeat)

In the 1990's, the definition of a smart pointer was: any class that overloads `operator*`, `operator->`, `operator++`, and `operator--` to do anything. Anything at all.

The cardinal sin of 1990's style OO programming: confusing encapsulation with obfuscation.

We have since learned: If it's called a pointer and looks like a pointer, then it better be a pointer, and the operators better behave like they do on "real" pointers.

Overloading operators `*`, `->`, `++`, and `--` has found its place in the world of iterators.

The iterator concept is an abstraction of the pointer concept with the addition of an underlying sequence —concrete or abstract—that the iterator traverses.

Here, the operators `*`, `->`, `++`, and `–` are overloaded to do what the iterator, by its nature, needs to do. That's expressive, as opposed to obfuscating.

The Problem

Traditional “management” of dynamically allocated memory in C++:

```
X* p = new X;

// use p, pass it around to other parts of the program

// somewhere, sometime, somebody does
delete p;
```

Common Problems:

1) Deletion happens too early. Someone’s still holding a copy of `p` and dereferences it: *dangling pointer*.

2) Deletion happens too late, or not at all: *memory leak*.

`boost::shared_ptr` saves

Avoid dangling pointers and memory leaks using `boost:shared_ptr`:

```
boost::shared_ptr<X> sp(new X);

// use sp, pass it around to other parts of the program

// under the hood, the destructor of the last copy of sp
// that goes out of scope calls delete on the raw pointer.
```

Deletion happens not too early, not to late.

In particular, this gives us all the benefits of RAII (think exceptions and multiple return statements).

Under the hood, `boost::shared_ptr` uses a reference count that is shared among all copies of a `shared_ptr` object.

Null Pointers

C/C++ raw pointers have a "built-in boolean":

```
X* p = new X;

// use p, perhaps pass it around

// Conditionally delete and null p
if(condition)
{
  delete p; // unless someone still has a copy
  p = NULL;
}

if(p)
{
  // use p
}
```

`boost::shared_ptr` can be nulled

Nulling `boost:shared_ptr`:

```
boost::shared_ptr<X> sp(new X);

// use sp, perhaps pass it around

if(condition)
{
  // Null sp, delete memory unless there are other copies
  sp.reset();
}

// Conversion to bool yields false if sp was
// default-constructed, or reset() has been called.
if(sp)
{
  // use sp
}
```

Reassigning a `shared_ptr`

```
boost::shared_ptr<X> sp(new X);

// use sp, perhaps pass it around

// Put sp in charge of a new X object, delete old X unless
// there are other copies
sp.reset(new X);
```

```
// The two resets are more efficient and concise versions
// of:
sp = boost::shared_ptr<X>();
sp = boost::shared_ptr<X>(new X);
```

Pragmatism First

There are two member functions in `boost:shared_ptr` that are somewhat controversial: pragmatists like them, purists don't.

```
boost::shared_ptr<X> sp(new X);

// use sp, pass it around to other parts of the program

// How many copies are out there?
long num_copies = sp.use_count();

// Am I the only one?
if(sp.unique()){
  // …
}
```

Use `boost::shared_ptr::use_count()` only for debugging purposes.

Understand that `boost::shared_ptr::use_count()` gets tricky to use in the presence of concurrency.

Pragmatism First (cont.)

```
boost::shared_ptr<X> sp(new X);

// Get the raw pointer
X* p = sp.get();
```

Use `boost::shared_ptr::get()` only to pass the raw pointer to library/system functions that expect a pointer and do not take ownership of the pointer.

Custom Deleter

`boost:shared_ptr` works with custom memory allocators.

Suppose you want to allocate objects of type X from a special, optimized heap.

```
// Gets memory from a custom heap and constructs an X at
// that location with placement new.
X* custom_allocate_x();

// Destructs the X and releases the memory
custom_deallocate_x(X* p);
```

You can now shield clients from the details of your allocation strategy with a factory function that returns a `boost::shared_ptr`.

```
boost::shared_ptr<X> make_x()
{
  return boost::shared_ptr<X>(
    custom_allocate_x(), &custom_deallocate_x
    );
}
```

Handling Handles

Custom deleters can be used to manage things such as operating system handles and descriptors.

```
// Win32 mutex handle managed by a boost::shared_ptr
// A Win32 HANDLE is a void*

boost::shared_ptr<void> hMutex(
  ::CreateMutex(NULL, TRUE, NULL),
  &CloseHandle
  );

if(! hMutex)
{
  throw processKaputt;
}
```

`boost::shared_ptr` as a Scope Guard

`boost::shared_ptr` can be used as a scope guard.

```
shared_ptr<void> guard(
  static_cast<void*>(0),
  boost::bind(foo, x, y, z)
  );
```

I personally much recommend using Andre Alexandrescu's `ScopeGuard` instead.

That's because `boost::shared_ptr` is *copyable*, and that very much expresses the wrong thing for a scope guard.

`boost::scoped_ptr`, on the other hand, does not allow custom deleters.

Hiding Implementation

```
class HiddenImplClass;
HiddenImplClass* createHiddenImpl();

HiddenImplClass* p = createHiddenImpl();
delete p; // problem: attempt to delete incomplete type!
```

```
class HiddenImplClass {
public:
  ~HiddenImplClass() {/*…*/}
  […]
};

HiddenImplClass* createHiddenImpl() {
  return new HiddenImplClass;
}
```

`boost::shared_ptr` and Incomplete Types

```
class SafeHiddenImplClass;
boost::shared_ptr<SafeHiddenImplClass>
  createSafeHiddenImpl();

  {
    boost::shared_ptr<SafeHiddenImplClass> ssp =
      createSafeHiddenImpl();
  } // fine: deletion is deferred to the ref count object
```

```
class SafeHiddenImplClass {
public:
  ~SafeHiddenImplClass() {/*…*/}
};

boost::shared_ptr<SafeHiddenImplClass>
createSafeHiddenImpl(){
  return boost::shared_ptr<SafeHiddenImplClass>(
    new SafeHiddenImplClass
  );
}
```

Thread Safety

When you use `boost:shared_ptr` in a multithreaded environment, imagine that your shared pointer objects were of a built-in type such as `double`.

```
boost::shared_ptr<X> sp(new X);

// Two threads concurrently execute
boost::shared_ptr<X> my_sp_copy = sp;
```

Ok: concurrent reads

```
boost::shared_ptr<X> sp1(new X);
boost::shared_ptr<X> sp2 = sp1;

// One thread executes
sp1.reset();

// Another thread executes concurrently
sp2.reset();
```

Ok: writes to different objects

```
boost::shared_ptr<X> sp(new X);

// Two threads execute concurrently
sp.reset();
```

Not ok: concurrent writes

Leaking Memory with Smart Pointers

True or false: If every occurrence of `new` in my program is in the constructor of a `boost:shared_ptr`, as in

```
Boost::shared_ptr<X> sp(new X);
```

Big fat false

then my program cannot leak memory.

There are two ways for memory leaks to occur.

Non-Exception-Safe Function Calls

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

// Good code
boost::shared_ptr<X> sp(new X);
f(sp, g());

// Bad code
//
// Problem if new X is evaluated first, then g(), and g()
// throws.
f(boost::shared_ptr<X>(new X), g());
```

Moral: avoid using unnamed temporaries of type `boost::shared_ptr`.

Cyclic References

```
struct B;
struct A {
  boost::shared_ptr<B> m_spB;
};

struct B {
  boost::shared_ptr<A> m_spA;
};

{
  boost::shared_ptr<A> spA(new A);
  boost::shared_ptr<B> spB(new B);

  spA->m_spB = spB;
  spB->m_spA = spA;
}
```

`spA` and `spB` keep each other alive forever.

Later, we will see how `boost::weak_ptr` solves this problem.

Miscellaneous

- `boost:shared_ptr` can be used in STL containers, including ordered ones.
- `boost::shared_ptr<Y> sp(new X);`

  is ok if `X*` converts to `Y*`.
- Conversions between raw pointers propagate to `boost::shared_ptr`:

  If `X*` converts to `Y*`, then `boost::shared_ptr<X>` converts to `boost::shared_ptr<Y>`.

Similarly, explicit casts between raw pointers translate to `boost::shared_ptr` via free functions. For example:

```
class Base{};
class Derived : public base {};

boost::shared_ptr<Base> spb(new Derived);
boost::shared_ptr<Derived> spd =
   boost::static_pointer_cast<Derived>(spb);
assert(spb == spd);
```

`boost::shared_array`

In C/C++, there are two versions of `new`/`delete`: one for objects, one for arrays.

The pointers returned by ordinary `new` and array `new` have the same type. Therefore, there is a separate version of `boost::shared_ptr` for array `new`/`delete`.

`boost::shared_array` is `boost::shared_ptr` for array `new`/`delete`.

`boost::shared_array` should be used very rarely, because:

C-style array should be used very rarely! Use `std::vector` instead. In rare cases, you may want to use a `boost::shared_ptr` to an `std::vector`.

boost::scoped_ptr

```
{
  boost::scoped_ptr<X> scp(new X);

  // Use scp. You cannot make copies of scp, nor can you
  // pass the ownership of the X object to any other smart
  // pointer.

  // When scp goes out of scope, the destructor calls
  // delete on the raw pointer.
}
```

`boost::shared_ptr` is very simple. Pretty much the only non-trivial thing in its interface is the `reset` method.

`boost::scoped_ptr` (cont.)

Use `boost::scoped_ptr`

- to keep a heap object alive for the duration of a scope (RAII)
- as a member of non-copiable objects

You could use `boost::shared_ptr` or `std::auto_ptr` wherever you use `boost::scoped_ptr`. The reason for wanting to use `boost::scoped_ptr` is to express intent.

`boost::scoped_array`

`boost::scoped_array` is `boost::scoped_ptr` for array `new`/`delete`.

`boost::scoped_array` should be used very rarely, because:

C-style arrays should be used very rarely! Use `std::vector` instead.

Cyclic References

Recall: cyclic references cause memory leaks even when using `boost::shared_ptr`:

```
struct B;
struct A {
  boost::shared_ptr<B> m_spB;
};

struct B {
  boost::shared_ptr<A> m_spA;
};
```

`boost::weak_ptr` solves this problem (and others).

Cyclic References and Symmetry

In practice, there is always an asymmetry in situations with cyclic references.

For example, a window may hold a pointer to child windows, and each child window may hold a backpointer to the window. But the ownership relation is from parent to child.

```
class window {
  std::list<boost::shared_ptr<window> > m_children;
  ??? m_parent; // backpointer to parent, if any
};
```

What type would we use for the backpointer? `boost::shared_ptr` would create a cyclic dependency. Moreover, it expresses the wrong thing: the back pointer is just for access, not for ownership.

Using a raw pointer wouldn’t be such a bad idea: no ownership implied.

However, a raw pointer could be dangling, e.g., when the window gets un-parented. A better solution is `boost::weak_ptr`.

`boost::weak_ptr` saves

`boost::weak_ptr` fills that need. It is a pointer is which is:

- created from a `boost::shared_ptr`,
- does not have ownership in the object pointed to, that is, it does not increment the reference count that the `boost:shared_ptr` maintains, and
- knows whether or not the object pointed to still exists, that is, it knows about the reference count that the `boost::shared_ptr` maintains and can ask if that reference count is zero.

In other words:

`boost::weak_ptr` is a pointer that observes an object that is already maintained by a `boost::shared_ptr`: if the object is still there, the weak pointer can access it, if not, the weak pointer knows that.

`boost::weak_ptr` is a pointer that can be dangling, but it knows whether it's dangling or not.

Using `boost::weak_ptr`

Use `boost::weak_ptr` to break cyclic dependencies:

```
class window {
  std::list<boost::shared_ptr<window> > m_children;
  boost::weak_ptr<window> m_parent;
};
```

Now the parent owns its children. A child can ask if it has a parent, and if so, access it.

`boost::weak_ptr` Creation and Resetting

A `boost::weak_ptr` must be created from a `boost::shared_ptr`.

```
boost::shared_ptr<X> sp1(new X);

// Construct a weak pointer that observes what sp points to
boost::weak_ptr<X> wp1(sp1);
//
// Alternatively, use assignment
boost::weak_ptr<X> wp2; // null weak pointer
wp2 = sp1;

// A weak pointer can be reset to null:
wp1.reset(); // Equivalent to wp1 = weak_ptr<X>();

// There is no reset that takes a weak pointer or shared
// pointer as an argument. Use assignment instead:
boost::shared_ptr<X> sp2(new X);
wp1 = sp2;
wp2 = wp1;
```

`boost::weak_ptr` Object Access

Recall: a `boost::weak_ptr` must ask whether or not it's dangling before it can be dereferenced. This is achieved by getting a `boost::shared_ptr` from the weak pointer.

```
color window::get_parent_bkgr_color()
{
  // We may not have a parent anymore
  boost::shared_ptr<window> safe_parent = m_parent.lock();
  if (safe_parent) {
    return safe_parent->get_bkgr_color();
  }
  else {
    throw no_parent;
  }
}
```

`boost::weak_ptr` Object Access (cont.)

Alternatively, you can use a `boost::shared_ptr` constructor instead of the weak pointer's `lock()` member.

```
color window::get_parent_bkgr_color()
{
  // We may not have a parent anymore
  try{
    boost::shared_ptr<window> safe_parent(m_parent);
    return safe_parent->get_bkgr_color();
  }
  catch(boost::bad_weak_ptr&) {
    throw no_parent;
  }
}
```

`boost::weak_ptr` Thread Safety

The fact that `boost::weak_ptr` uses a `boost::shared_ptr` to access the object pointed to guarantees that the proper level of thread safety is maintained.

```
color window::get_parent_bkgr_color()
{
  // We may not have a parent anymore
  try{
    // Getting the shared_ptr from the weak_ptr is a read
    // operation that is thread safe: other threads may
    // concurrently modify other shared_ptr or
    // weak_ptr objects that manage the same pointee.
    boost::shared_ptr<window> safe_parent(m_parent);

    // Using safe_parent is safe no matter what other
    // threads do: we now have ownership of the pointee!
    return safe_parent->get_bkgr_color();
  }
//...
```

Weak Pointers Relations Can Be Tricky to Set up

```
struct B;
struct A {
  boost::shared_ptr<B> m_spB;
};

struct B {
  boost::weak_ptr<A> m_wpA;
};

{
  boost::shared_ptr<A> spA(new A);
  boost::shared_ptr<B> spB(new B);       // 1

  spA->m_spB = spB;                      // 2
  spB->m_wpA = boost::weak_ptr<A>(spA); // 3
}
```

You may want to do the work of lines 1–3 in `A`'s constructor. That's not possible. More about this issue later.

Other Uses of `boost::weak_ptr`

Another use of `boost::weak_ptr` is to maintain a “weak” cache of objects, that is, a cache that expires when the cached object is no longer in use.

```
class widget_factory
{
  typedef std::map<std::string, boost::weak_ptr<widget> > cache;
  cache m_cache;

  public:
  boost::shared_ptr<widget> get_widget(std::string id){
    cache::const_iterator widget_pos = m_cache.find(id);
    if(widget_pos != m_cache.end()){
      boost::shared_ptr<widget> cached_widget =
        widget_pos->second.lock();
      if(cached_widget){
        return cached_widget;
      }
    }
    boost::shared_ptr<widget> wid(new widget(id));
    m_cache[id] = boost::weak_ptr<widget>(wid);
    return wid;
  }
};
```

`boost::intrusive_ptr`

`boost::intrusive_ptr` is a version of `boost::shared_ptr` that assumes that the owned object already maintains a reference count.

`boost::intrusive_ptr` allows you to leverage the existing reference count mechanism.

In practice, unfortunately, `boost::intrusive_ptr` is primarily a toy for people who love premature optimization:

"If I implement the reference count myself in my class and use `boost::intrusive_ptr` instead of `boost::shared_ptr`, then my program will be so much faster!"

NO!

std::auto_ptr

`std::auto_ptr` is like `boost::scoped_ptr` insofar as ownership is unique. But in the case of `std::auto_ptr`, that unique ownership can be transferred.

```
std::auto_ptr<X> ap1(new X);
std::auto_ptr<X> ap2 = ap1; // ownership transferred to ap2

ap2->foo(); // ok, ap2 owns the object
ap1->foo(); // crash, ap1 neither owns nor has the object
```

`std::auto_ptr` (cont.)

Use `std::auto_ptr`

1. to express that a function will take ownership of a heap object that you have created (sink):

   ```
   void x_sink(std::auto_ptr<X> ap);
   ```

2. to express that a function has created an object and gives ownership to the caller (source):

   ```
   std::auto_ptr<X> x_source();
   ```

When taking ownership of an object from an `std::auto_ptr` as in 2. above, you may for example want to create a `boost::shared_ptr` from it.

```
boost::shared_ptr<X> sp(x_source());
```

`boost::scoped_ptr`, because of its emphasis on simplicity, does not have a contstructor from `std::auto_ptr`. But you can transfer ownership explicitly:

```
boost::scoped_ptr<X>(x_source().release());
```

Why `std::unique_ptr`?

`std::unique_ptr`, which is part of C++0x, is a better `std::auto_ptr`. Here is one important improvement. Recall the following incorrect use of `std::auto_ptr`:

```
std::auto_ptr<X> ap1(new X);
std::auto_ptr<X> ap2 = ap1; // ownership transferred to ap2

ap1->foo(); // crash, ap1 neither owns nor has the object
```

Ok, that’s just bad code. Things get more subtle in generic code.

`std::auto_ptr` and Generic Code

```
template<typename T>
void foo(T arg){
  T temp(arg);
  // use temp
  // use arg
}
```

`std::auto_ptr` is obviously not suitable as a template argument for `foo()`. But it is not possible to express that in a natural way.

`std::auto_ptr` and Generic Code (cont.)

`std::unique_ptr` uses rvalue references to express its behavior properly. In the code above, the line

```
T temp(arg);
```

does not compile with `T == std::unique_ptr`.

You'd have to say

```
T temp(std::move(arg));
```

The use of `std::move` expresses that it is not ok to use `arg` again. Now using `arg` on a subsequent line would be an obvious mistake even in the generic code.

In summary, in C++0x you can write generic code in such a way that what doesn't work for `std::unique_ptr` won't compile for `std::unique_ptr`.

You can do even better with concepts: generic code that works for `std::unique_ptr` requires *movability*, as opposed to *copyability*.

Setting up Weak Pointer Relations

```
class window {
  std::list<boost::shared_ptr<window> > m_children;
  boost::weak_ptr<window> m_parent;

public:
  void add_child() {
    boost::shared_ptr<window> child(new window);
    m_children.push_back(child);
    child->m_parent = // need a shared_ptr to this ☹
  }
};
```

`boost::enable_shared_from_this` saves.

boost::enable_shared_from_this

```
class window : public boost::enable_shared_from_this<window>
{
  std::list<boost::shared_ptr<window> > m_children;
  boost::weak_ptr<window> m_parent;

public:
  void add_child() {
    boost::shared_ptr<window> child(new window);
    m_children.push_back(child);
    child->m_parent = shared_from_this();// ☺
  }
};
```

How is it done?

How to enable shared from `this`

If we were to implement `enable_shared_from_this` manually in our class, it would look like this:

```
class window {
  // …
  boost::weak_ptr<window> m_weak_this;
  window(){}

public:
  boost::shared_ptr<window> shared_from_this() {
    return boost::shared_ptr<window>(m_weak_this);
  }

  static boost::shared_ptr<window> create() {
    boost::shared_ptr<window> shared_this(new window);
    shared_this->m_weak_this = shared_this;
    return shared_this;
  }
};
```

How to enable shared from `this` (cont.)

`boost::enable_shared_from_this` does two things:

1. It abstracts the work shown on the previous slide out into a base class.
2. It makes the create method of the previous slide superfluous by putting its work into the constructor of `boost::shared_ptr` from raw pointer.

Note: `boost::enable_shared_from_this` makes getting a shared pointer from `this` very elegant. However, there remains one restriction that cannot be overcome: getting a shared pointer from `this` can never be done in an object’s constructor.