Using TR1's shared_ptr and weak_ptr Smart Pointers

David Kieras, EECS Department, University of Michigan

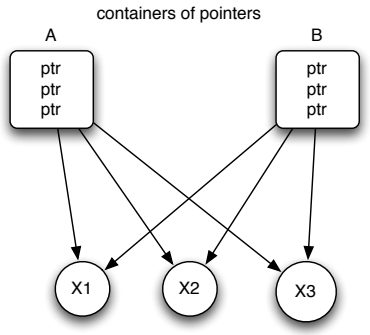
October 22, 2008

Since the first C++ Standard was approved in 1998, a group of C++ wizards have been working on the open-source Boost library (www.boost.org), with the goal of developing new library facilities good enough to be considered for addition to the C++ Standard Library. The Standard Library Technical Report No. 1 (TR1) specifies several Boost libraries that are scheduled to become part of the official C++ Standard Library in the next version of the Standard (expected to be completed in 2009). This note deals with a s*mart pointer* facility,which consists of `shared_ptr` and its partner, `weak_ptr`, and some additional functions and template classes. This document is a tutorial on this facility and how to use it . See the posted code examples for the examples presented here.

Concept of the TR1 Smart Pointers

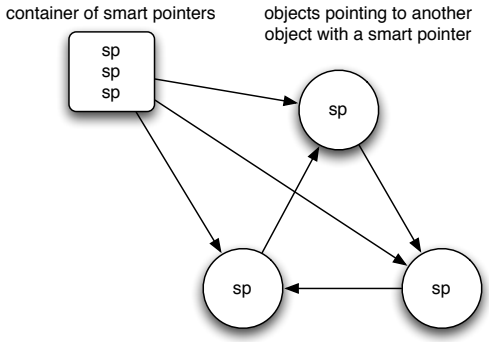
Smart pointers are class objects that behave like built-in pointers but also *manage* objects that you create with `new` so that you don't have to worry about when and whether to delete them - the smart pointers automatically delete the *managed object* for you at the appropriate time. The smart pointer is defined in such a way that it can be used syntactically almost exactly like a built-in (or "raw") pointer. So you can use them pretty much just by substituting a smart pointer object everywhere that the code would have used a raw pointer. A smart pointer contains a built-in pointer, and is defined as a template class so that it can point to a class object of any type.

The TR1 `shared_ptr` class template is a referenced-counted smart pointer; a count is kept of how many smart pointers are pointing to the managed object; when the last smart pointer is destroyed, the count goes to zero, and the managed object is then automatically deleted. It is called a "shared" smart pointer because the smart pointers all share ownership of the managed object - any one of the smart pointers can keep the object in existence; it gets deleted only when no smart pointers point to it any more. Using these can simplify memory management, as shown with a little example diagrammed below:

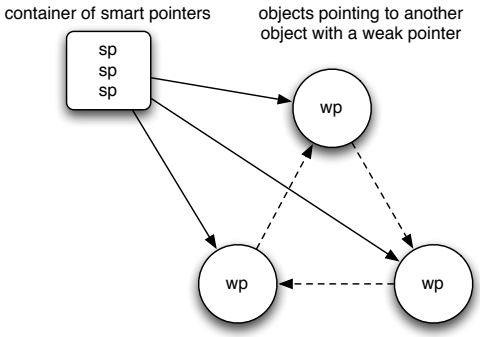


Suppose we need two containers (A and B) of pointers referring to a single set of objects, X1 through X3. Suppose that if we remove the pointer to one of the objects from one of the containers, we will want to keep the object if the pointer to it is still in the other container, but delete it if not. Suppose further that at some point we will need to empty container A or B, and only when both are emptied, we will want to delete the three pointed-to objects. Suppose further that it is hard to predict in what order we will do any of these operations (e.g. this is part of a game system where the user's activities determines what will happen). Instead of writing some delicate code to keep track of all the possibilities, we could use smart pointers in the containers instead of built-in pointers. Then all we have to do is simply remove a pointer from a container whenever we want, and if it turns out to be the last pointer to an object, it will get "automagically" deleted. Likewise, we could clear a container whenever we want, and if it has the last pointers to the objects, then they all get deleted. Pretty neat! Especially when the program is a lot more complicated!

However, a problem with reference-counted smart pointers is that if there is a ring, or cycle, of objects that have smart pointers to each other, they keep each other "alive" - they won't get deleted even if no other objects in the universe are pointing to them from "outside" of the ring. This cycle problem is illustrated in the diagram below that shows a container of smart pointers pointing to three objects each of which also point to another object with a smart pointer and form a ring. If we empty the container of smart pointers, the three objects won't get deleted, because each of them still has a smart pointer pointing to them.



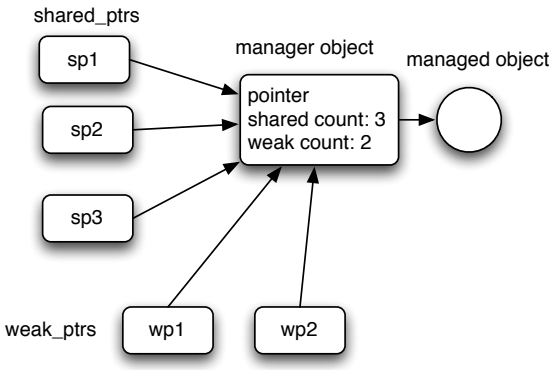
TR1 includes a solution: "weak" smart pointers: these only "observe" an object but do not influence its lifetime. A ring of objects can point to each other with `weak_ptrs`, which point to the managed object but do not keep it in existence. This is shown in the diagram below, where the "observing" relations are shown by the dotted arrows.



If the container of smart pointers is emptied, the three objects in the ring will get automatically deleted because no other smart pointers are pointing to them; like raw pointers, the weak pointers don't keep the pointed-to object "alive." The cycle problem is solved. But unlike raw pointers, the weak pointers "know" whether the pointed-to object is still there or not and can be interrogated about it, making them much more useful than a simple raw pointer would be. How is this done?

How they work

A lot of effort over several years went into making sure the TR1 smart pointers are very well-behaved and as foolproof as possible, and so the actual implementation is very subtle. But a simplified sketch of the implementation helps to understand how to use these smart pointers. Below is a diagram illustrating in simplified form what goes on under the hood of `shared_ptr` and `weak_ptr`.



The process starts when the managed object is dynamically allocated, and the first `shared_ptr` (`sp1`) is created to point to it; the `shared_ptr` constructor creates a *manager* object (dynamically allocated). The manager object contains a pointer to the managed object; the overloaded member functions like `shared_ptr::operator->` access the pointer in the manager object to get the actual pointer to the managed object.[1] The manager object also contains two reference counts: The shared count counts the number of `shared_ptrs` pointing to the manager object, and the weak count counts the number of `weak_ptrs` pointing to the manager object. When `sp1` and the manager object are first created, the shared count will be 1, and the weak count will be 0.

If another `shared_ptr` (`sp2`) is created by copy or assignment from `sp1`, then it also points to the same manager object, and the copy constructor or assignment operator increments the shared count to show that 2 `shared_ptrs` are now pointing to the managed object. Likewise, when a weak pointer is created by copy or assignment from a `shared_ptr` or another `weak_ptr` for this object, it points to the same manager object, and the weak count is incremented. The diagram shows the situation after three `shared_ptrs` and two `weak_ptrs` have been created to point to the same object.

Whenever a `shared_ptr` is destroyed, or reassigned to point to a different object, the `shared_ptr` destructor or assignment operator decrements the shared count. Similarly, destroying or reassigning a `weak_ptr` will decrement the weak count. Now, when the shared count reaches zero, the `shared_ptr` destructor deletes the managed object and sets the pointer to 0. If the weak count is also zero, then the manager object is deleted also, and nothing remains. But if the weak count is greater than zero, the manager object is kept. If the weak count is decremented to zero, and the shared count is also zero, the `weak_ptr` destructor deletes the manager object. Thus the managed object stays around as long as there are `shared_ptrs` pointing to it, and the manager object stays around as long as there are either `shared_ptrs` or `weak_ptrs` referring to it.

Here's why the `weak_ptr` is more useful than a built-in pointer. It can tell by looking at the manager object whether the managed object is still there: if the pointer and shared count are zero, the managed object is gone, and no attempt should be made to refer to it. If the pointer and shared count are non-zero, then the managed object is still present, and `weak_ptr` can make the pointer to it available. A `weak_ptr` member function creates and returns a new `shared_ptr` to the object; the new `shared_ptr` increments the shared count, which ensures that the managed object will stay in existence as long as necessary. In this way, the `weak_ptr` can point to an object without affecting its lifetime, but still make it easy to refer to the object, and at the same time, ensure that it stays around if someone is interested in it.

Notice how `shared_ptr` and `weak_ptr` have a fundamental difference: `shared_ptr` can be used syntactically almost identically to a built-in pointer. However, a `weak_ptr` is much more limited. You cannot use it like a built-in pointer at all - in fact, you can't use it to actually refer to the managed object at all! Almost the only thing you can do is interrogate it to see if the managed object is still there, or construct a `shared_ptr` from it. If the managed object

---

[1] To keep the language from getting too clumsy, we'll say that a smart pointer *is pointing to the managed object* if it is pointing to the manager object that actually contains the pointer to the managed object.

is gone, the `shared_ptr` will be an empty one (e.g. it will test as zero); if the managed object is present, then the `shared_ptr` can be used normally.

Important restrictions in using shared_ptr and weak_ptr

Although they have been carefully designed to be as fool-proof as possible, these smart pointers are not built into the language, but rather are ordinary classes subject to the regular rules of C++. This means that they aren't foolproof - you can get undefined results unless you follow certain rules. In a nutshell, these rules are:

- You can only use these smart pointers to refer to objects allocated with `new` and that can be deleted with `delete`. No pointing to objects on the function call stack!
- You must ensure that there is only one manager object for each managed object. You do this by writing your code so that when an object is first created, it is immediately given to a `shared_ptr` to manage, and any other `shared_ptrs` or `weak_ptrs` that are needed point to that object are all directly or indirectly copied or assigned from that first `shared_ptr`. The customary way to ensure this is to write the `new` object expression as the argument for a `shared_ptr` constructor.
- If you want to get the full benefit of smart pointers, your code must avoid using built-in pointers to refer to the same objects; otherwise you can still have problems with dangling pointers or double deletions.[2]

Using shared_ptr

How to access tr1::shared_ptr and tr1::weak_ptr.

gcc 4.x comes with Boost's version of the TR1 library. The following #includes and using directives make the facilities described in this handout available in gcc 4.x. Other compiler/library environments will differ.

```
#include <tr1/memory>

using namespace std::tr1;
```

Basic use of shared_ptr

Using a `shared_ptr` is easy as long as you follow the rules listed above. When you create an object with new, write the new expression in the constructor for the `shared_ptr`. Thereafter, use the `shared_ptr` as if it were a built-in pointer; it can be copied or assigned to another `shared_ptr`, which means it can be used as a call-by-value function argument or return value, or stored in containers. When it goes out of scope, or gets deleted, the reference count will be decremented, and the pointed-to object deleted if necessary. You can also call a `reset()` member function, which will decrement the reference count and delete the pointed-to object if necessary, and result in an empty `shared_ptr` just like a default-constructed one. Thus while you will still write `new` to create an object, if you always use a `shared_ptr` to refer to it, you will `never` need to `delete` the object. Here is a code sketch illustrating the basic usage:

```
class Thing {
public:
   void defrangulate();
};
ostream& operator<< (ostream&, const Thing&);
...
// a function can return a shared_ptr
shared_ptr<Thing> find_some_thing();
// a function can take a shared_ptr parameter by value;
shared_ptr<Thing> do_something_with(shared_ptr<Thing> p);

...
```

[2] There is no requirement that you use smart pointers everywhere in a program, just that you not use both smart and built-in pointers to the same objects.

```
void foo()
{
    // the new is in the shared_ptr constructor expression:
    shared_ptr<Thing> p1(new Thing);
    ...
    shared_ptr<Thing> p2 = p1; // p1 and p2 now share ownership of the Thing
    ...
    shared_ptr<Thing> p3(new Thing); // another Thing

    p1 = find_some_thing();    // p1 may no longer point to first Thing
    do_something_with(p2);
    p3->defrangulate();     // call a member function like built-in pointer
    cout << *p2 << endl;    // dereference like built-in pointer
    // reset not by assigning to zero, but with a member function:
    p1.reset();  // decrement count, delete if last
}
// p1, p2, p3 go out of scope, decrementing count, delete the Things if last
```

The design of `shared_ptr` helps prevent certain mistakes. For example, the only way to get a `shared_ptr` to take an address from a raw pointer is with the constructor, which makes it easier to get a `shared_ptr` into the picture right away, and not have stray raw pointers running around that might be used to delete the object or start a separate `shared_ptr` family for the same object.

```
Thing * bad_idea()
{
    shared_ptr<Thing> sp; // an empty pointer
    Thing * raw_ptr = new Thing;
    sp = raw_ptr; // disallowed - compiler error !!!
    ...
    return raw_ptr;  // danger!!! - caller could make a mess with this!
}

shared_ptr<Thing> better_idea()
{
    shared_ptr<Thing> sp(new Thing);
    ...
    return sp;
}
```

The only way you can get the raw pointer inside the manager object is with a member function, `get()`, - there is no implicit conversion. However, the raw pointer should be used with extreme caution - again you don't want to have stray raw pointers to refer to the managed objects:

```
Thing * another_bad_idea()
{
    shared_ptr<Thing> sp(new Thing);
    Thing * raw_ptr  = sp;  // disallowed! Compiler error!

    Thing * raw_ptr = sp.get();   // you must want it, but why?
    ...
    return raw_ptr;  // danger!!! - caller could make a mess with this!
}
```

Testing and comparing shared_ptrs

You can compare two `shared_ptrs` using the ==, !=, and < operators; they compare the internal raw pointers and so behave just like these operators between built-in pointers. In addition, a `shared_ptr` provides a conversion to a

bool, so that you can test for whether the internal raw pointer to the managed object is zero or not: So if sp is a shared_ptr, if(sp) will test true if sp is pointing to an object, and false if it is not pointing to an object, just like a built-in pointer.

Inheritance and shared_ptr

A difficult part of the design of shared_ptr is to make sure that you can use them to refer to classes in a class hierarchy in the same way as built-in pointers. For example, with built-in pointers you can say:

```
class Base {};
class Derived : public Base {};
...
Derived * dp1 = new Derived;
Base * bp1 = dp1;
Base * bp2(dp1);
Base * bp3 = new Derived;
```

The constructors and assignment operators in shared_ptr (and weak_ptr) are defined with templates so that if the built-in pointers could be validly copied or assigned, then the corresponding shared_ptrs can be also:

```
class Base {};
class Derived : public Base {};
...
shared_ptr<Derived> dp1(new Derived);
shared_ptr<Base> bp1 = dp1;
shared_ptr<Base> bp2(dp1);
shared_ptr<Base>(new Derived);
```

Casting shared_ptrs

One excuse for getting the raw pointer from a shared_ptr would be in order to cast it to another type. Again to make it easier to avoid raw pointers, TR1 supplies some function templates that provide a casting service corresponding to the built-in pointer casts. These functions internally call the get() function from the supplied pointer, perform the cast, and return a shared_ptr of the specified type. Again the goal is to emulate what can be done with built-in pointers, so they will be valid if and only if the corresponding cast between built-in pointers is valid. Continuing the above example:

```
shared_ptr<Base> base_ptr (new Base);
shared_ptr<Derived> derived_ptr;
// if static_cast<Derived *>(base_ptr.get()) is valid, then the following is valid:
derived_ptr = static_pointer_cast<Derived>(base_ptr);
```

Note how the casting function looks very similar to a built-in cast, but it is a function template being instantiated, not a built-in operator in the language. The available casting functions are static_pointer_cast, const_pointer_cast, and dynamic_pointer_cast, corresponding to the built-in cast operators with the similar names.

Using weak_ptr

Weak pointers just "observe" the managed object; they don't "keep it alive" or affect its lifetime. Unlike shared_ptrs, when the last weak_ptr goes out of scope or disappears, the pointed-to object can still exist because the weak_ptrs do not affect the lifetime of the object. But the weak_ptr can be used to determine whether the object exists, and to provide a shared_ptr that can be used to refer to it.

The definition of weak_ptr is designed to make it relatively foolproof, so as a result there is very little you can do directly with a weak_ptr. For example, you can't dereference it; neither operator* nor operator-> is defined for a weak_ptr. You can't access the pointer to the object with it - there is no get() function. The only comparison defined is operator< so that you can store weak_ptrs in an ordered container; but that's all.

Initializing a weak_ptr

A default-constructed `weak_ptr` is empty, pointing to nothing (not even a manager object). You can point a `weak_ptr` to an object only by copy or assignment from a `shared_ptr` or an existing `weak_ptr` to the object. In the example below, we create a new Thing pointed to by `sp`; then are shown the possible ways of getting a `weak_ptr` to also point to the new Thing. This makes sure that a `weak_ptr` is always referring to a manager object created by a `shared_ptr`.

```
shared_ptr<Thing> sp(new Thing);

weak_ptr<Thing> wp1(sp);   // construct wp1 from a shared_ptr
weak_ptr<Thing> wp2;       // an empty weak_ptr - points to nothing
wp2 = sp;                  // wp2 now points to the new Thing
weak_ptr<Thing> wp3 (wp2); // construct wp3 from a weak_ptr
weak_ptr<Thing> wp4
wp4 = wp2;                 // wp4 now points to the new Thing.
```

You can use the `reset()` member function to set a `weak_ptr` back to the empty state in which it is pointing to nothing.

Using a weak_ptr to refer to an object

You can't refer to the object directly with a `weak_ptr`; you have to get a `shared_ptr` from it first with the `lock()` member function. The `lock()` function examines the state of the manager object to determine whether the managed object still exists, and provides a empty `shared_ptr` if it does not, and a `shared_ptr` to the manager object if it does; the creation of this `shared_ptr` has the effect of ensuring that the managed object, if it still exists, stays in existence while we use it; it "locks" it into existence, so to speak (explaining the name). Continuing the above example:

```
shared_ptr<Thing> sp2 = wp2.lock();  // get shared_ptr from weak_ptr
```

Now that we have another `shared_ptr` for the new Thing, the previous one (`sp`) can go out of scope, and the Thing will stay in existence. However, in the normal use of a `weak_ptr`, it is possible that the managed object has already been deleted. For example, suppose we have a function that takes a `weak_ptr` as a parameter and wants to call Thing's `defrangulate()` function using the `weak_ptr`. We can't call the member function for a non-existent object, so we have to check that the object is still there before calling the function. There are three ways to do this:

First, we can go ahead and get the `shared_ptr`, but test for whether it is empty or pointing to something by testing it for true/false, analogous to what we would do with a built-in pointer that might be zero:

```
void do_it(weak_ptr<Thing> wp){
    shared_ptr<Thing> sp = wp.lock(); // get shared_ptr from weak_ptr
    if(sp)
        sp->defrangulate(); // tell the Thing to do something
    else
        cout << "The Thing is gone!" << endl;
}
```

Second, we can ask the `weak_ptr` if it has "expired" before bothering to get the `shared_ptr`:

```
void do_it(weak_ptr<Thing> wp) {
    if(wp.expired())
        cout << "The Thing is gone!" << endl;
    else {
        shared_ptr<Thing> sp = wp.lock(); // get shared_ptr from weak_ptr
        sp->defrangulate(); // tell the Thing to do something
        }
}
```

Third, we can construct a `shared_ptr` from a `weak_ptr`; if the `weak_ptr` is expired, an exception is thrown, of type `tr1::bad_weak_ptr`. This has its uses, but the above two methods are generally handier and more direct. Example:

```
void do_it(weak_ptr<Thing> wp){
   shared_ptr<Thing> sp(wp);  // construct shared_ptr from weak_ptr
   // exception thrown if wp is expired, so if here, sp is good to go
   sp->defrangulate(); // tell the Thing to do something
}
...
try {
   do_it(wpx);
}
catch(bad_weak_ptr&)
{
   cout << "A Thing (or something else) has disappeared!" << endl;
}
```

Special Case: Getting a shared_ptr for "this" Object

Why this is a problem

Suppose we have a situation where a Thing member function needs pass a pointer to "this" object to another function, for example an ordinary function of some sort. If we are not using smart pointers, there is no problem:

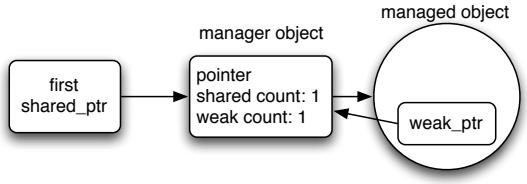
```
class Thing {
public:
   void foo();
   void defrangulate();
};

void transmogrify(Thing *);

int main()
{
   Thing * t1 = new Thing;
   t1->foo();
   ...
   delete t1;    // done with the object
}
...
void Thing::foo()
{
   // we need to transmogrify this object
   transmogrify(this);
}
...

void transmogrify(Thing * ptr)
{
   ptr->defrangulate();
   /* etc. */
}
```

Now say we want to use smart pointers to automate the memory management for Thing objects. To be reliable, this means we need to avoid all raw pointers to Things, and hand around only smart pointers. One would think all we need to do is change all the `Thing *` to `shared_ptr<Thing>`, and then the following code would work; but there is a big problem with it:

```
class Thing {
public:
    void foo();
    void defrangulate();
};

void transmogrify(shared_ptr<Thing>);

int main()
{
    shared_ptr<Thing> t1(new Thing); // start a manager object for the Thing
    t1->foo();
    ...
    // Thing is supposed to get deleted when t1 goes out of scope
}
...
void Thing::foo()
{
    // we need to transmogrify this object
    shared_ptr<Thing> sp_this(this); // danger! starts a second manager object!
    transmogrify(sp_this);
}
...

void transmogrify(shared_ptr<Thing> ptr)
{
    ptr->defrangulate();
    /* etc. */
}
```

When `main` creates the `shared_ptr` named `t1`, a manager object gets created for the new Thing. But in function `Thing::foo` we create a `shared_ptr<Thing>` named `sp_this` which is constructed from the raw pointer `this`. We end up with a second manager object which is pointed to the same Thing object as the original manager object. Oops! Now we have a double-deletion error waiting to happen - in this example, as soon as the `sp_this` goes out of scope, the Thing will get deleted; then when the rest of `main` tries to use `t1` it may find itself trying to talk to a non-existent Thing, and when `t1` goes out of scope, we will be deleting something that has already been deleted, corrupting the heap.

While one could tinker with any one chunk of code to work around the problem, a general solution is preferable. If we can ensure that the managed object contains a `weak_ptr` referring to the same manager object as the first `shared_ptr` does, then it is pointing to `this` object, and so we at any time we can get a `shared_ptr` from the `weak_ptr` that will work properly. The desired situation is shown in the diagram below:



Pulling this off in a reliable way is a bit tricky. A reasonable solution is provided in TR1 by a template class named `tr1::enabled_shared_from_this` which has a `weak_ptr` as a member variable and member function named `shared_from_this()` which returns a `shared_ptr` constructed from the `weak_ptr`.

The Thing class must be modified to inherit from the `enabled_shared_from_this` class, so that Thing now has a `weak_ptr<Thing>` as a member variable. When the first `shared_ptr` to a Thing object is created, the `shared_ptr` constructor uses template magic to detect that the `enable_shared_from_this` base class is present,

and then initializes the `weak_ptr` member variable from the first `shared_ptr`. Once this has been done, the `weak_ptr` in Thing points to the same manager object as the first `shared_ptr`. Then when you need a `shared_ptr` pointing to `this` Thing, you call the `shared_from_this()` member function, which returns a `shared_ptr` obtained by construction from the `weak_ptr`, which in turn will use the same manager object as the first `shared_ptr`.

The above example code would be changed to first, have the Thing class inherit from the template class, and second, use `shared_from_this()` to get a pointer to `this` object:

```
class Thing : public enable_shared_from_this<Thing> {
public:
    void foo();
    void defrangulate();
};

int main()
{
    // The following starts a manager object for the Thing and also
    // initializes the weak_ptr member that is now part of the Thing.
    shared_ptr<Thing> t1(new Thing);
    t1->foo();
    ...
}
...
void Thing::foo()
{
    // we need to transmogrify this object
    // get a shared_ptr from the weak_ptr in this object
    shared_ptr<Thing> sp_this = shared_from_this();
    transmogrify(sp_this);
}
...

void transmogrify(shared_ptr<Thing> ptr)
{
    ptr->defrangulate();
    /* etc. */
}
```

Now when `sp_this` goes out of scope, there is no problem - there is only the single, original, manager object for the Thing. Problem solved - the world is safe for using smart pointers everywhere!

The disadvantage is that to get a smart `this` pointer, we have to modify the Thing class, and carefully follow the rule of creating the Thing object only in the constructor of a `shared_ptr`. However, one should be following this rule anyway with `shared_ptr`. The more serious problem is that if you don't have access or permission to modify the Thing class, you can't use the `enable_shared_from_this` without wrapping Thing in another class, leading to some complexity. But only some class designs need to hand a `this` pointer around, and so the problem is not inevitable.

Conclusion

TR1's `shared_ptr` and `weak_ptr` work well enough that many software organizations have adopted them as a standard smart pointer implementation. So feel free to use them to automate or simplify your memory management, especially when objects may end up pointing to each other in hard-to-predict ways.

Where to Read More

To read more about TR1 and the Boost libraries, try the documentation on the boost web site; it ranges from quite cryptic to very clear and useful. The following books were available at the time of this writing:

Becker, P. *The C++ Standard Library Extensions: A Tutorial and Reference*. Addison-Wesley, 2007. Becker is a widely-respected C++ expert, specializing in library implementations. This book gets into how TR1 is implemented more than just how to use it, and so is pretty difficult.

Karlsson, Bjorn. *Beyond the C++ Standard Library; An Introduction to Boost*. This book surveys a large part of the Boost library from the perspective of why and how you should use it to improve your code. TR1 is well covered. Fairly clear and practical, but beware: not all of the examples will compile in gcc's version of TR1, so be sure to try out his exact examples and get them working before going on to use the facilities.