Using C++11's Smart Pointers

David Kieras, EECS Department, University of Michigan

May 2015

This tutorial deals with C++11's s*mart pointer* facility, which consists `unique_ptr`, `shared_ptr` and its partner, `weak_ptr`, and some associated functions and template classes. See the posted code examples for the examples presented here.

Concept of the C++11 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 whose type parameter is the type of the pointed-to object, so you can declare smart pointers that point to a class object of any type.

When it comes to dynamically-allocated objects, we often talk about who "owns" the object. "Owning" something means it is yours to keep or destroy as you see fit. In C++, by ownership, we mean not just which code gets to refer to or use the object, but mostly what code is responsible for deleting it. If smart pointers are not involved, we implement ownership in terms of where in the code we place the `delete` that destroys the object. If we fail to implement ownership properly, we get memory leaks, or undefined behavior from trying to follow pointers to objects that no longer exist. Smart pointers make it easier to implement ownership correctly by making the smart pointer destructor the place where the object is deleted. Since the compiler ensures that the destructor of a class object will be called when the object is destroyed, the smart pointer destruction can then automatically handle the deletion of the pointed-to object. The smart pointer owns the object and handles the deletion for us.

This tutorial first presents `shared_ptr`, which implements *shared ownership*. Any number of these smart pointers jointly own the object. The owned object is destroyed only when its last owning smart pointer is destroyed. In addition, a `weak_ptr` doesn't own an object at all, and so plays no role in when or whether the object gets deleted. Rather, a `weak_ptr` merely *observes* objects being managed by `shared_ptrs`, and provides facilities for determining whether the observed object still exists or not. C++11's `weak_ptrs` are used with `shared_ptrs`. Finally, `unique_ptr` implements *unique ownership* - only one smart pointer owns the object at a time; when the owning smart pointer is destroyed, then the owned object is automatically destroyed.

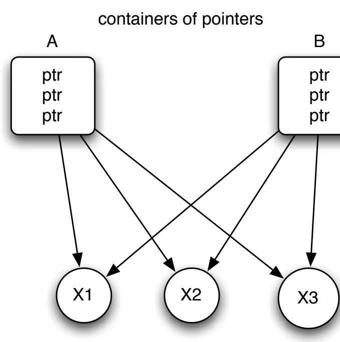
How to Access the C++11 Smart Pointers.

In a C++11 implementation, the following #include is all that is needed:

```
#include <memory>
```

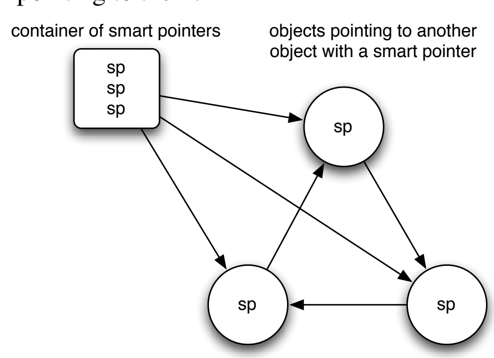
Shared Ownership with shared_ptr

The `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:

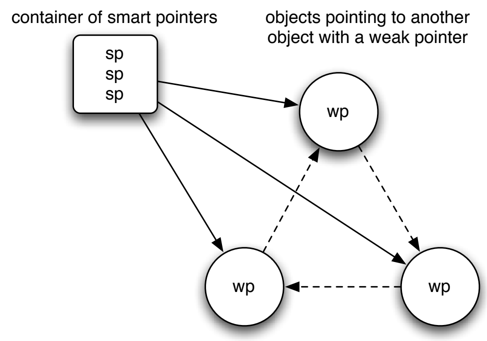


Suppose we need two containers (A and B) of pointers referring to a single set of objects, X1 through X3. Suppose that if we remove the pointer to one of the objects from one of the containers, we will want to keep the object if the pointer to it is still in the other container, but delete it if not. 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.



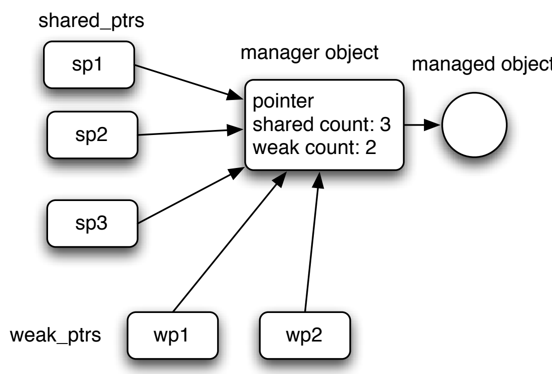
C++11 includes a solution: "weak" smart pointers: these only "observe" an object but do not influence its lifetime. A ring of objects can point to each other with `weak_ptrs`, which point to the managed object but do not keep it in existence. This is shown in the diagram below, where the "observing" 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 by the Boost group (boost.org) went into making sure the C++11 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.

[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.

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_ptr`s pointing to it, and the manager object stays around as long as there are either `shared_ptr`s or `weak_ptr`s 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/or 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. This is done by a `weak_ptr` member function that 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.

But `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 — in fact, you can't use it to actually refer to the managed object at all! Almost the only things you can do are to interrogate it to see if the managed object is still there, or construct a `shared_ptr` from it. If 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 that the compiler can't enforce. 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! Trying to delete them will cause a run-time error!
- 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_ptr`s or `weak_ptr`s that are needed to 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, or use the `make_shared` function template described below.
- If you want to get the full benefit of smart pointers, your code should avoid using raw pointers to refer to the same objects; otherwise it is too easy to have problems with dangling pointers or double deletions. In particular, smart pointers have a `get()` function that returns the pointer member variable as a built-in pointer value. This function is rarely needed. As much as possible, leave the built-in pointers inside the smart pointers and use only the smart pointers.[2]

---

[2] There is no requirement that you use smart pointers everywhere in a program, it just recommended that you do not use both smart and built-in pointers to the same objects. Actually it is more subtle than that - having some built-in pointers in the mix might be useful, but only if you are hyper-careful that they cannot possibly be used if their objects have been deleted, and you never, ever, use them to delete the object or otherwise exercise ownership of the object with them. Such mixed code can be very hard to get right. My recommendation is to point to a set of objects with either raw pointers (where you manage the ownership directly), or smart pointers (which automate the ownership), but never mix them in pointing to the same set of objects.

Using shared_ptr

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 appropriate, and result in an empty `shared_ptr` that is just like a default-constructed one. You can also reset a `shared_ptr` by assigning it the value `nullptr`, which is converted to an empty `shared_ptr` before the assignment. 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 explicitly `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);

...
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 with a member function or assignment to nullptr:
    p1.reset();  // decrement count, delete if last
    p2 = nullptr; // convert nullptr to an empty shared_ptr, and decrement count;
}
// 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 to the raw pointer type. However, the raw pointer should be used with extreme caution - again you don't want to have stray raw pointers that 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!
}
```

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_ptr`s 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> bp3(new Derived);
```
[3]

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, C++11 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

---

[3] This usage is correct, but has an odd effect. The manager object stores a pointer of the *original* type, `Derived*`, but `get()` and the dereferencing operators will convert the stored pointer to a `Base*`. But the managed object will get deleted through the stored `Derived*` pointer, not through a `Base*` pointer. So if you forget the recommendation to declare `~Base()` as virtual, deletion will still start with `~Derived()`, unlike what would happen if you delete through a built-in `Base*` pointer, and furthermore, in this situation, correct destruction would then not happen if there are classes further derived from `Derived` unless you happened to declare `~Derived()`as virtual. Rather than risk this confusion, you should still follow the recommendation to declare `~Base()` as virtual, which also insures that all derived class destructors are virtual, so that correct destruction will happen no matter whether or how the client code is using smart pointers.

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.

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. [4]

Getting better memory allocation performance

A problem with `shared_ptr` is that if you create an object with `new`, and then create a `shared_ptr`, as in the usual scenario:

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

There are actually *two* dynamic memory allocations that happen: one for the object itself from the `new`, and then a second for the manager object created by the `shared_ptr` constructor. Since memory allocations are slow, this means that creating a `shared_ptr` is slow relative to using either a raw pointer, or a so-called "intrusive" reference-counted smart pointer where the reference count is a member variable of the object. To address this problem, C++11 includes a function template `make_shared` that does a *single* memory allocation big enough to hold both the manager object and the new object, passing along any constructor parameters that you specify, and returns a `shared_ptr` of the specified type, which can then be used to initialize the `shared_ptr` that you are creating (with efficient move semantics). So instead of:

```
shared_ptr<Thing> p(new Thing); // ouch - two allocations
```

you would write:

```
shared_ptr<Thing> p(make_shared<Thing>()); // only one allocation!
```

Why does `<Thing>` appear twice in this statement? The `make_shared` function returns a `shared_ptr` of its specified type, which doesn't have to be the same as the type of the `shared_ptr` that we are initializing (as long as they can be converted — see above). This enables you to specify the pointer type of the `shared_ptr` and the type of the constructed object separately, as in:

```
shared_ptr<Base> bp(make_shared<Derived1>());
```

This creates an object whose type is `Derived1` (that inherits from `Base`), and returns a `shared_ptr<Derived1>` that is used to initialize the `shared_ptr<Base>`.

Finally, if `Thing`'s constructor had parameters, you would place their values in the argument list of `make_shared`:

---

[4] The Standard says shared_ptrs can have only an explicit conversion to a `bool` type must be `explicit` - which is a good idea. The compiler automatically applies this conversion in a conditional or logical expression. However, assigning a `shared_ptr` to a `bool` variable, or returning it from a function that returns `bool` requires you to explicitly convert it with a `static_cast` or a constructor-style cast. Do not try to avoid the cast using an implicit conversion with the result of the `get()` function - this can look like an exposure of the raw pointer, which really does need to be avoided if at all possible.

```
shared_ptr<Thing> p (make_shared<Thing>(42, "I'm a Thing!"));
```

Thanks to the magic of C++11 variadic templates and perfect forwarding, you can write anything in the function argument list that you could write in an ordinary constructor argument list, and they will get correctly passed to the constructor.

Using `make_shared` also avoids explicit use of `new`, promoted in the slogan "no naked new!" That is, if you are using `shared_ptrs` throughout a project, never writing `new`, and always using `make_shared`, is an easy way to make sure that all allocated objects are managed by `shared_ptrs`.

Because only a single memory allocation is involved when you use `make_shared` to initialize a `shared_ptr`, you can expect improved performance over the separate allocation approach; this can be valuable if there are a lot of `shared_ptrs` being created. As usual, the object's destructor will be called when the last `shared_ptr` is destroyed, but there is a possible downside that if there are still `weak_ptrs` referring to the object, the entire chunk of memory will not be returned to the allocation pool until the last `weak_ptr` is destroyed.

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 - they have no ownership rights. 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. There is a comparison function defined 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.[5]

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
```

[5] Unlike `shared_ptr`, you can't reset a `weak_ptr` by assignment to `nullptr`.

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:

1. 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;
}
```

This is the most useful and common way to use a weak_ptr to access the object.

2. We can ask the weak_ptr if it has "expired":

```
bool is_it_there(weak_ptr<Thing> wp) {
    if(wp.expired()) {
        cout << "The Thing is gone!" << endl;
        return false;
        }
    return true;
}
```

This approach is useful as a way to simply ask whether the pointed-to object still exists. Notice that if after calling expired(), the code goes on to use lock() to get a shared_ptr to the object, testing first for expired() is redundant and may actually be problematic.[6]

3. We can construct a shared_ptr from a weak_ptr; if the weak_ptr is expired, an exception is thrown, of type std::bad_weak_ptr. This has its uses, but the first method is 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;
}
```

[6] The C++11 smart pointers are designed to be thread-safe (at least on most platforms). But notice that in a multithreaded environment, some other thread may have been holding the shared_ptrs to the object we are pointing to with the weak_ptr, so if the weak_ptr is not expired, and we go ahead and acquire the shared_ptr with .lock(), we need to check the shared_ptr again in case the object got deleted between the expired() call and the lock() call.

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. */
}
```

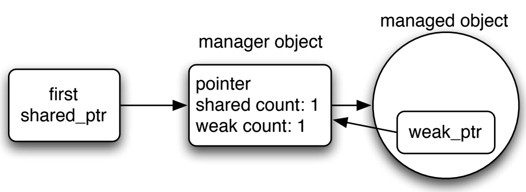
Now say we want to use smart pointers to automate the memory management for `Thing` objects. To be reliable, this means we need to avoid all raw pointers to `Things`, and hand around only smart pointers. One would think all we need to do is change all the `Thing *` to `shared_ptr<Thing>`, and then the following code would compile; 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_for_this(this); // danger! a second manager object!
    transmogrify(sp_for_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_for_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_for_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 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:



Getting this done in a reliable way is a bit tricky. Rather than DIY code, you should use the solution provided in the C++11 Library. There is a template class named `std::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 `enabled_shared_from_this<Thing>`, 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!

There are three problems with this solution. First, 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`. Second, you can't use `shared_from_this()` in the constructor of the Thing class. The `weak_ptr` member variable has to be set to point to the manager object by the `shared_ptr` constructor, and this can't run until the Thing constructor has completed. You'll have to do some kind of work-around, like a calling another member function on the constructed Thing that completes the setup involved that needs the `shared_from_this()`. Third, 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

C++11'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.

Unique Ownership with unique_ptr

At first glance, `unique_ptr` looks like it gives you the same thing as `shared_ptr`. With a `unique_ptr`, you can point to an allocated object, and when the `unique_ptr` goes out of scope, the pointed-to object gets deleted, and this happens regardless of how we leave the function, either by a return or an exception being thrown somewhere. For example:

```
void foo ()
{
    unique_ptr<Thing> p(new Thing); // p owns the Thing
    p->do_something();  // tell the thing to do something
    defrangulate();     // might throw an exception
}  // p gets destroyed; destructor deletes the Thing
```

Note that to use `unique_ptr` reliably, you need to follow the same basic rules as those presented earlier for `shared_ptr`. So why is `unique_ptr` worth having, since you could use a `shared_ptr` to do the same thing? There are two reasons:

First, the basic mechanism of `unique_ptr` is so simple that it costs *nothing* to use. It has a pointer member variable of type `Thing*` that either points to an object, which means the `unique_ptr` owns it, or it is zero (`nullptr` in C++11), meaning that the `unique_ptr` doesn't own any object. Either way, all the destructor has to do is a `delete` on this pointer variable and if the `unique_ptr` owns an object, it is gone, or nothing happens if it doesn't own an object[7]. Compare this to the overhead of how shared_ptrs dynamically allocate a manager object and then increment/decrement and test one or two reference counts. This simplicity means that there is really zero overhead of using of unique_ptr compared to a built-in pointer. So the automatic cleanup costs nothing! It has already become recommended practice to use unique_ptr in this sort of scenario.

Second, unique_ptr implements a *unique ownership* concept - an object can be owned by only one unique_ptr at a time - the opposite of *shared ownership*. This means unique_ptr is very different from shared_ptr; this special feature is explained in the rest of this handout.

What makes the ownership unique?

An object is owned by exactly one unique_ptr. The unique ownership is enforced by disallowing (with =delete) copy construction and copy assignment. So unlike built-in pointers or shared_ptr, you can't copy or assign a unique_ptr to another unique_ptr. If you follow the basic rules for using smart pointers, this means that you can't have two unique_ptrs that contain the same raw pointer value and so claim ownership to the same object and cause double deletion problems if both go out of scope. Forbidding copy takes care of this:

```
unique_ptr<Thing> p1 (new Thing); // p1 owns the Thing
unique_ptr<Thing> p2(p1);  // error - copy construction is not allowed.
unique_ptr<Thing> p3;      // an empty unique_ptr;
p3 = p1;                   // error, copy assignment is not allowed.
```

Notice that since copy construction is disallowed, if you want to pass a unique_ptr as a function argument, you have to do it by reference.

A couple of additional goodies: You can test a unique_ptr to see if it owns an object (e.g. with if(p)); there is a conversion to bool that supplies the value of the pointer member variable as true/false.[8] If you want to delete the object manually, call the reset() function or assign to nullptr – this does the delete and then resets the internal pointer to nullptr.

Transferring ownership

While unique_ptr is defined to help you avoid ambiguous ownership, it might be handy to transfer ownership of an object from one unique_ptr to another. This allows one to directly represent the idea of having several potential owners of an object, but only one of them can own the object at a time. This is easily done with move semantics: the move constructor and move assignment operator are defined for unique_ptr so that they transfer ownership from the original owner to the new owner. After move construction, the newly created unique_ptr owns the object and the original unique_ptr owns nothing. After move assignment, the object previously owned by the left-hand unique_ptr has been deleted, and the left-hand unique_ptr now owns the object previously owned by the right-hand unique_ptr, which now owns nothing. The implementation is fast and simple – the unique_ptr moving code just copies and zeroes the pointer member variables.

Remember that the returned value of a function is a rvalue, so the presence of move construction and assignment means that we can return a unique_ptr from a function and assign it to another unique_ptr; the effect is that ownership of the object is passed out of the function into the caller's unique_ptr. A little example:

```
//create a Thing and return a unique_ptr to it:
unique_ptr<Thing> create_Thing()
{
    unique_ptr<Thing> local_ptr(new Thing);
    return local_ptr;  // local_ptr will surrender ownership
}
void foo()
{
    unique_ptr<Thing> p1(create_Thing()); // move ctor from returned rvalue
```

---

7 Recall that the delete operator is defined as doing nothing if it is given a zero or nullptr value.

[8] This conversion operator is explicit - see the previous footnote about converting shared_ptr to bool.

```
    // p1 now owns the Thing
    unique_ptr<Thing> p2;  // default ctor'd; owns nothing
    p2 = create_Thing();    // move assignment from returned rvalue
    // p2 now owns the second Thing
}
```

Thus ownership can be transferred implicitly from an rvalue `unique_ptr`, but not from an lvalue `unique_ptr`.

Explicit transfer of ownership between unique_ptrs

If you really want to transfer ownership from one `unique_ptr` to another, you can use move semantics to do it; all you have to do is treat the original `unique_ptr` as an rvalue using `std::move()`, which casts its argument to an rvalue reference. Then the move version of construction or assignment will be invoked. For example:

```
unique_ptr<Thing> p1(new Thing); // p1 owns the Thing
unique_ptr<Thing> p2; // p2 owns nothing

// invoke move assignment explicitly
p2 = std::move(p1); // now p2 owns it, p1 owns nothing

// invoke move construction explicitly
unique_ptr<Thing> p3(std::move(p2)); // now p3 owns it, p2 and p1 own nothing
```

Using unique_ptr with Standard Containers

You can fill a Standard Container with `unique_ptrs` that own objects, and the ownership then effectively resides in the container. Here are several key things to remember in making use of this interesting concept:

- You must fill the container by supplying rvalue `unique_ptrs`, so the ownership transfers into the `unique_ptr` in the container. Either use an unnamed rvalue `unique_ptr` as the argument for the container inserting/filling function, or use `std::move` with a named `unique_ptr`.
- If you erase an item in the container, you are destroying the `unique_ptr`, which will then delete the object it is pointing to.
- If you empty, clear, or destroy the container, all of the pointed-to objects will be deleted because all the `unique_ptrs` will be destroyed.
- If you transfer ownership out of container items, the empty `unique_ptrs` stay in the container. If you leave empty `unique_ptrs` in the container, your code will need to check for empty `unique_ptrs` before dereferencing them.
- You can refer to an object by referring to the `unique_ptr` in the container without trying to copy or take it out of the container (you may have to test it for `nullptr` first - see above). For example:

```
std::vector<unique_ptr<Thing>> v;
...
if(v[3])                    // check that v[3] still owns an object
    v[3]->defrangulate();   // tell object pointed to by v[3] to defrangulate
```

See the course examples directory for some demo code that uses unique_ptr with Standard Containers.

Avoiding explicit `new`

For symmetry with `shared_ptr`, and to support the "no naked new" slogan, C++14 added a `make_unique` template analogous to `make_shared`, so that you can create both the `unique_ptr` and the pointed-to object without explicitly invoking `new`. However, there is no performance improvement in this case because there is no need to allocate a manager object with `unique_ptr`.

Conclusion

The most common use of `unique_ptr` is as a pretty fool-proof way of making sure an object allocated in a function (or class constructor) gets deleted. However, there are situations in which ownership of objects needs to be transferred around but always remains in one place at a time; `unique_ptr` gives you way to represent this concept directly.