Being Smart About Pointers

Michael VanLoon

Adapted from CPPcon 2014

m.vanloon@F5.com

A Quick Diversion

- Let’s talk about RAII…

RAII:
Resource Acquisition is Initialization

- Scope-Based Resource Management
- An idiom for managing resources based on container object lifetime
- Most often used in scoped variable scenarios
- Automatic and low maintenance
- Reduces the opportunity for negligence
- How? Deterministic, Encapsulation, Locality, Exception Safety

RAII

- Deterministic:
  - Deallocation of resources happens immediately, during container object destruction
  - Standard scoping and object lifetime rules apply
  - Standard ordering rules apply
  - No delayed garbage collection

RAII

- Encapsulation:
  - Cleanup code is kept with other logically-related object management code
  - Cleanup code is not sprinkled throughout your program wherever an object might exit scope

RAII

- Locality:
  - Constructor, Destructor, other object management code, and container methods are all logically and physically grouped together

RAII

- Exception Safety:
  - Stack-based variables will exit scope as stack is unwound following exception
  - SBRM-based variables will automatically clean up resources as they exit scope

(lack of) RAII example

old and busted >> new hotness

```
class C
{
public:
    C(int i): x(i) { }
    ~C() { }
private:
    int x;
};
```

```
int
main(int argc, const char* argv[])
{
    for (int i = 0; i < 100; ++i)
    {
        C* c = new C(i);
        /* do something with c */
        delete c;
    }
    return 0;
}
```

RAII example

old and busted >> **new hotness**

```
class C;

class SBRM
{
public:
    SBRM(C* c): pc(c)
        { }
    ~SBRM()
        { delete pc; }
    operator C*()
        { return pc; }
private:
    C* pc;
};
```

```
int
main(int argc, const char* argv[])
{
    for (int i = 0; i < 100; ++i)
    {
        SBRM cc(new C(i));
        C* c = cc;
        /* do something with c */
    }
    return 0;
}
```

Smart pointers, the toolbox

- Boost::scoped_ptr and std::unique_ptr (C++11)
- shared_ptr, with make_shared and enable_shared_from_this
- weak_ptr

Smart pointers, what they can do

- Features
  - RAII, with all its benefits
  - Custom deleters – not just new/delete!
  - Reference counted in some cases
  - Guarantee of deterministic delete
- Benefits
  - No more memory leaks or multiple frees
  - Code easier to read, easier to maintain, with less bugs
  - Simpler code logic; no messy ownership tracking

Pointer Best Practices

- Avoid explicit new/delete whenever possible
  - e.g. make_shared, make_unique, or custom factory
- Avoid delete except for exceptional circumstances
  - It's a sign you need to re-think pointer handling code
  - It's easy to forget
  - It's easy to accidentally maintain around
- Avoid handling raw pointers whenever reasonable
  - Put them into a container as soon as you get them
  - Even better: use make_shared or make_unique
- How?
  - Boost::scoped_ptr or std::unique_ptr
  - shared_ptr
  - Custom container of your own design

boost::scoped_ptr and std::unique_ptr

- Both adhere to RAII principles
- Both appropriate for managing local, scope-based resources
- Both actively prevent unintentional sharing of resource
- Both are faster and simpler than shared_ptr
- Prefer unique_ptr if you have it. If not, scoped_ptr and shared_ptr solve many of the same problems.

boost::scoped_ptr

- Available in boost
- scoped_ptr has neither Move nor Copy semantics; it is truly local scope only
- Separate scoped_array for new [] / delete []
- Does not support custom deleter

std::unique_ptr

- C++11 improvement on scoped_ptr
- unique_ptr has Move semantics but not Copy semantics (i.e. can return out of function and be embedded in standard containers)
- unique_ptr natively supports delete []
- unique_ptr allows custom deleter
- In C++14, has make_unique which is more exception safe and supports avoidance of explicit new and pointer handling

boost::scoped_ptr example

```
class C {
public:
    C() { cout << "C!" << endl; }
    ~C() { cout << "~C" << endl; }
};

C* Cfactory() {
    return new C;
}

void spFunc() {
    boost::scoped_ptr<C> c(Cfactory());
    // Do something with c...
}
```

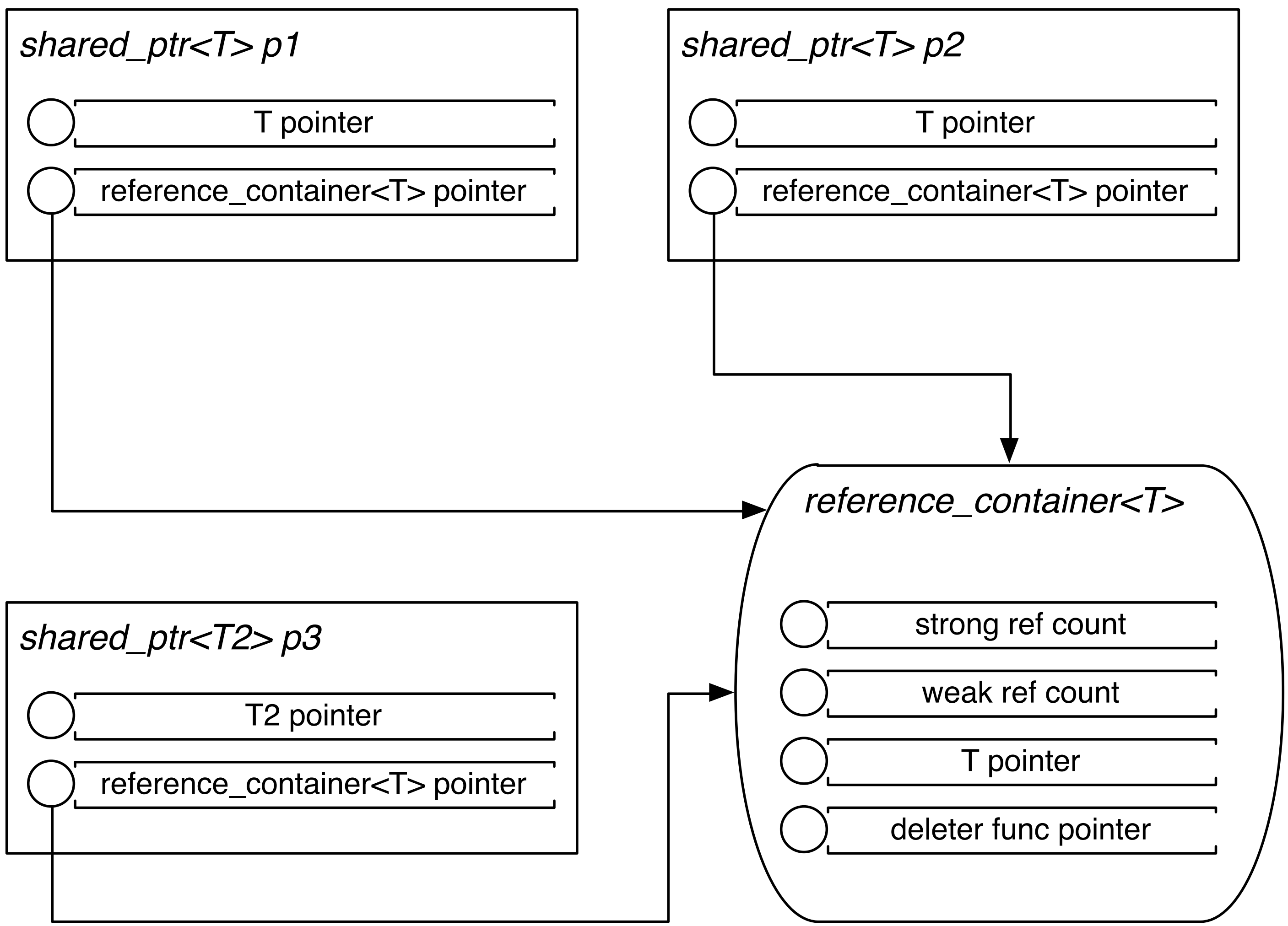
std::unique_ptr example

```
unique_ptr<C> returnUP() {
    unique_ptr<C> c(new C);
    // Do some magic with c, but we're not final owner
    return c;
}

void upFunc() {
    unique_ptr<C> c(returnUP());
    // Do something with c; we are the final owner
}

void upVecFunc() {
    vector<unique_ptr<C>> cVec;
    for (int i = 0; i < 5; ++i)
        cVec.emplace_back(new C);
    // cVec goes out of scope and cleans up nicely
}
```

shared_ptr, make_shared and enable_shared_from_this

- shared_ptr adheres to RAII principles
- Reference-counted container
- Copyable
- Custom deleter supported for cleaning up resource allocations other than new
- make_shared allocates object more efficiently, is exception safe, and avoids explicit new
- enable_shared_from_this allows object to return shared_ptr containing itself
- Prefer unique/scoped_ptr over shared_ptr

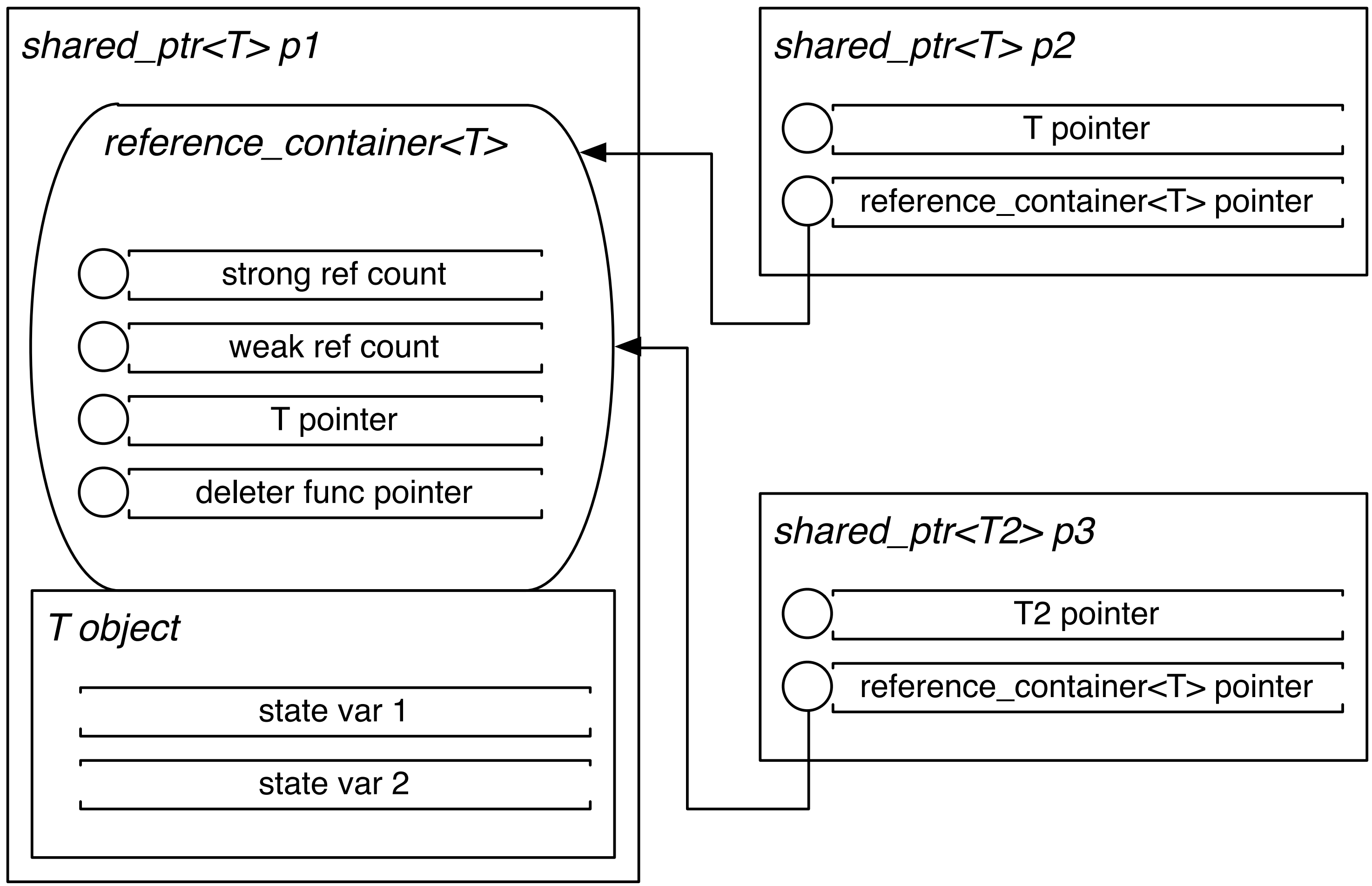
Example shared_ptr implementation



shared_ptr construction/assignment and make_shared

```
class C {
public:
    C() { cout << "C!" << endl; }
    ~C() { cout << "~C" << endl; }
};

void spAssign() {
    shared_ptr<C> p1 = make_shared<C>();
    assert(p1.use_count() == 1);
    shared_ptr<C> p2 = p1;
    assert(p2.use_count() == 2);
    shared_ptr<C> p3(new C);
    assert(p3.use_count() == 1);
    p2 = p3;
    assert(p1.use_count() == 1 && p3.use_count() == 2);
}
```

Example make_shared optimization



shared_ptr copy/move and embedding in containers

```
shared_ptr<C> Cfactory() {
    return make_shared<C>();
}

void spVec() {
    vector< shared_ptr<C> > cVec;
    cVec.emplace_back(Cfactory()); // first C
    cVec.emplace_back(Cfactory()); // second C
    // Do something with cVec...
    // everything cleans up nicely
}
```

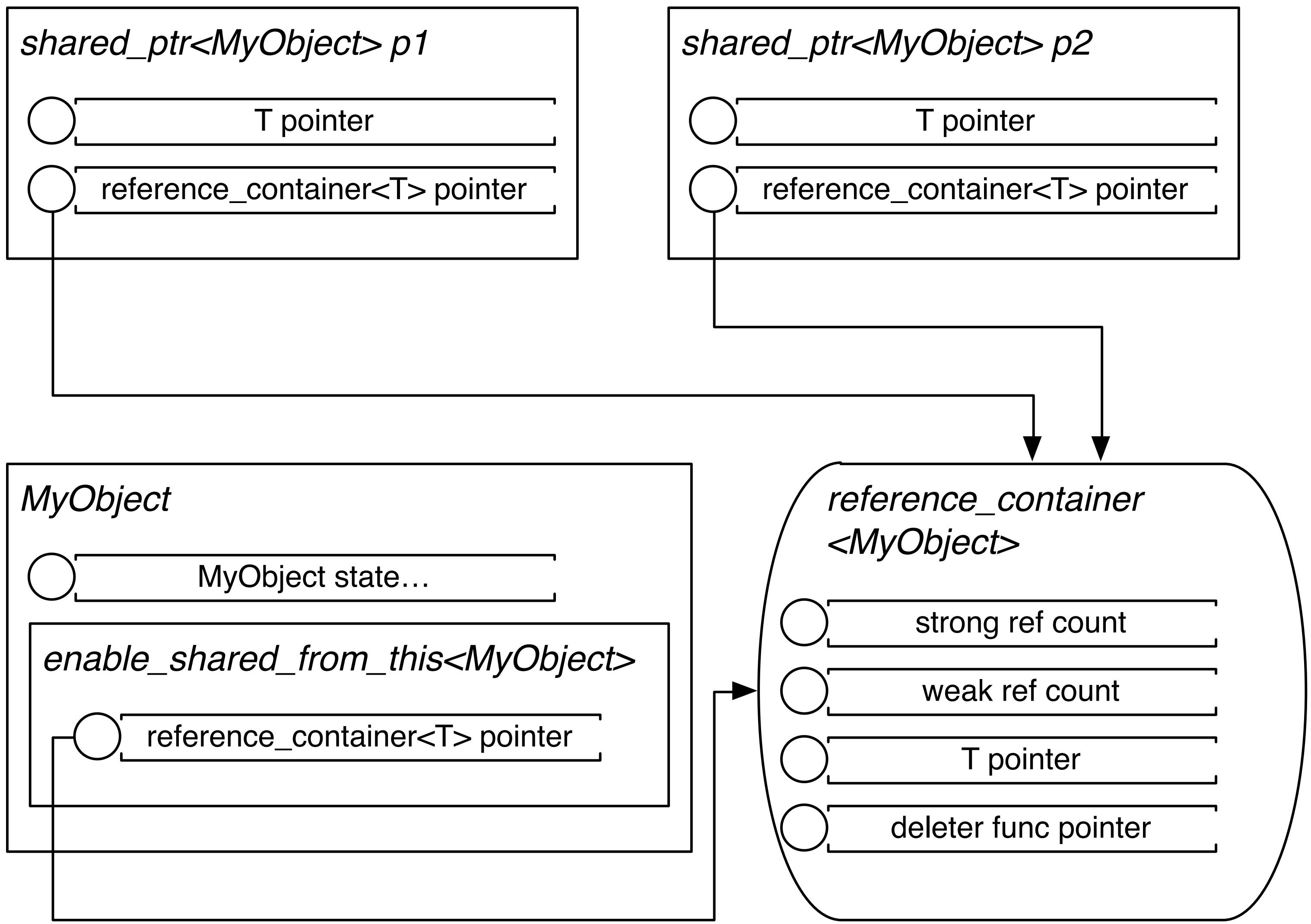
shared_ptr share-aware class and enable_shared_from_this

```
class BAD {
public:
    shared_ptr<BAD> getSP()
    { return shared_ptr<BAD>(this); }

    static shared_ptr<BAD> create()
    { return make_shared<BAD>(); }
};

shared_ptr<BAD> beBAD() {
    shared_ptr<BAD> p1 = BAD::create();
    BAD* raw = p1.get();
    shared_ptr<BAD> p2(raw->getSP());
    return p1;
}
```

shared_ptr share-aware class and enable_shared_from_this

```
class Good: public enable_shared_from_this<Good> {
public:
    shared_ptr<Good> getSP()
    { return shared_from_this(); }

    static shared_ptr<Good> create()
    { return make_shared<Good>(); }
};

shared_ptr<Good> doGood() {
    shared_ptr<Good> p1 = Good::create();
    Good* raw = p1.get();
    shared_ptr<Good> p2(raw->getSP());
    return p1;
}
```

Example enable_shared_from_this



shared_ptr custom deleter

```
shared_res* allocShared() {
    // Do some magic to allocate shared resource
    return malloc(100);
}

void releaseShared(shared_res* p) {
    // Do some magic to release shared resource
    free(p);
}

void testSharedResource() {
    shared_ptr<shared_res> sr(allocShared(), releaseShared);
    // Do something with sr...
    // safe and automatic cleanup
}
```

weak_ptr

- Helps break circular references
- Holds a non-owning “weak” reference
- Contained resource can only be used by requesting a shared_ptr from the weak_ptr
- A weak_ptr will not prevent resource from being destroyed (but does delay destruction of shared ref count block)

weak_ptr

```
void wpTwiddler() {
    boost::weak_ptr<C> w1;
    {
        boost::shared_ptr<C> p1 = boost::make_shared<C>();
        assert(p1.use_count() == 1);
        w1 = p1;
        assert(p1.use_count() == 1); // weak adds no ref
        assert(!w1.expired()); // w1 is usable?
        boost::shared_ptr<C> p2 = w1.lock();
        assert(p2); // not empty
        assert(p2.use_count() == 2);
    }
    assert(w1.expired() && w1.use_count() == 0);
}
```

What is a smart pointer?

- It's a container for a pointer
  - It holds your pointer and gives it back to you
  - It manages the lifetime of the held resource
  - It facilitates RAII
    - Constructs with reasonable defaults
    - Cleans up pointer at appropriate time
- (Ideally) properly conveys interface semantics
  - Shared, unique, etc.

What is a smart pointer?

- It’s a proxy for a pointer
  - It stands in for the pointer in most cases where you might use the raw pointer
  - In many cases it can transparently act as if it is the original pointer type

Various kinds of smart pointers

- “Generic” smart pointers:
  - auto_ptr
  - scoped_ptr
  - unique_ptr
  - shared_ptr
  - weak_ptr
  - _com_ptr_t
  - CComPtr

Various kinds of smart pointers

- Domain-specific smart pointers:
  - std::string
  - _bstr_t
  - _variant_t

The different kinds of reference counting

- Object-based reference counting
  - The count is stored in the object
  - Microsoft COM model
  - Sometimes referred to as "intrusive" (boost::intrusive_ptr)
  - Example: Microsoft COM
- Container-based reference counting
  - The count is stored in the smart pointer container
  - Example: shared_ptr

Object-based reference counting

- Advantages
  - Very simple smart pointer logic
  - Can easily mix with raw pointer usage
  - Object has greater visibility into its lifetime and cleanup
  - Lighter weight overall
- Disadvantages
  - Outside the COM world, less familiar
  - Outside the COM world, no ready-made solution
  - Object itself becomes slightly bigger and more complex

Container-based reference counting

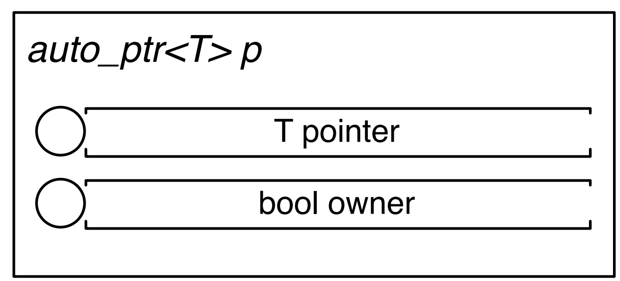
- Advantages
  - Can contain pointer to any objet; no added object knowledge or complexity
  - Very standard and widely used
- Disadvantages
  - Difficult to mix with raw pointer usage
  - Heavier weight overall because of complexity of good shared_ptr implementation

How it works: object-based reference counting

- Sorry this talk is too short, so here's what you get:
  - Stick an atomic count in an object
  - Provide methods to add and release references
  - Object deletes itself when reference goes to zero

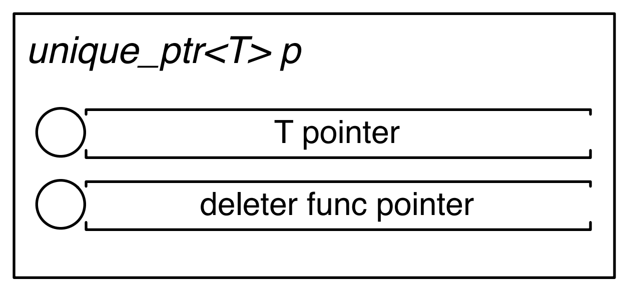
How it works: auto_ptr

- Contains raw pointer and an ownership flag
- Ownership is transferred to last auto_ptr that takes ownership of pointer
- Problematic and error-prone ownership semantics, so it's deprecated!



How it works: scoped_ptr and unique_ptr

- Contains raw pointer
- Does now allow transfer of ownership (except via move in unique_ptr), which simplifies logic
- unique_ptr optionally contains pointer to custom deleter func, which will be called instead of delete if desired



How it works: shared_ptr

- High level overview:
- Contains a pointer and a reference count that is shared among all instances
- Contains optional custom deleter
- Last releaser decrements reference to zero, and object is deleted

Interface Semantics or How to Pass Your Stuff Around

- Functions that receive resources controlled by shared_ptr can be written with several different interfaces:
  - `foo(shared_ptr<Widget>)`
  - `foo(const shared_ptr<Widget>&)`
  - `foo(shared_ptr<Widget>&&) [C++11]`
  - `foo(Widget&)`
  - `foo(const Widget&)`
  - `foo(Widget*)`
  - `foo(const Widget*)`
- The interface you choose says a lot about what you intend for both the caller and the callee

Interface Semantics or How to Pass Your Stuff Around

`foo(shared_ptr<Widget>)`

- Caller contract:
  - Caller is required to wrap object in shared_ptr, which implies that Widget object cannot be stack-based
- Callee contract:
  - Callee is allowed to take an ownership stake in object (i.e. copy shared_ptr)
- Other implications:
  - shared_ptr is copied during call, resulting in increment of atomic variable

Interface Semantics or How to Pass Your Stuff Around

`foo(const shared_ptr<Widget>&)`

- Caller contract:
  - Caller is required to wrap object in shared_ptr, which implies that Widget object cannot be stack-based
- Callee contract:
  - Callee is may take an ownership stake in object (i.e. copy shared_ptr), but is not required to do so
- Other implications:
  - shared_ptr reference is not copied during function call, resulting in better performance

Interface Semantics or How to Pass Your Stuff Around

`foo(shared_ptr<Widget>&&) [C++11]`

- Caller contract:
  - Caller is required to wrap object in shared_ptr, which implies that Widget object cannot be stack-based
  - Caller will receive object that has been emptied out by move operation
- Callee contract:
  - Callee must take ownership stake of object (i.e. move shared_ptr)
- Other implications:
  - shared_ptr reference is moved during function call from caller's copy to callee's copy

Interface Semantics or How to Pass Your Stuff Around

`foo(Widget&)` **and** `foo(const Widget&)`

- Caller contract:
  - Caller is not required to wrap object in shared_ptr, which allows object to be allocated either on stack or heap
  - Caller retains full ownership of object
  - Object is always valid (reference)
- Callee contract:
  - Callee cannot take ownership stake of object, because shared_ptr container is absent
- Other implications:
  - No churn of atomic reference count

Interface Semantics or How to Pass Your Stuff Around

`foo(Widget*)` and `foo(const Widget*)`

- Caller contract:
  - Caller is not required to wrap object in shared_ptr, which allows object to be allocated either on stack or heap
  - Caller retains full ownership of object
  - Object may be optional (i.e. NULL pointer)
- Callee contract:
  - Callee cannot take ownership stake of object, because shared_ptr container is absent
- Other implications:
  - No churn of atomic reference count
  - Prefer reference over pointer unless object must be optional

Interface Semantics Key Points

- Choose to pass by `Widget&` or `Widget*`, unless the callee is expected to take an ownership stake in the object
  – Passing by `Widget` type does not impose container or allocation constraints on caller
  – Choose to pass by `Widget&` over `Widget*`, unless the Widget object is optional (may be NULL)
- Choose to pass by `const shared_ptr<Widget>&` if the callee may take ownership in object
  – shared_ptr reference may be copied if needed, but will not make an unnecessary copy on function entry

Summary

- Use a smart pointer. It avoids the explicit, problematic manual delete
- Prefer unique_ptr (or boost::scoped_ptr if you don't have C++11 yet) because of its efficiency and simplicity if possible
- Use shared_ptr for any variables that need shared semantics
- Use make_shared when possible; it may be more efficient, and avoids explicit new
- Consider object-based reference counting for cases where it fits a paradigm better