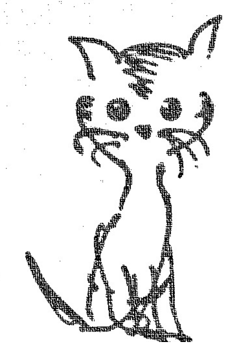
Introduction to C++ smart pointers

Jason Rassi
rassi@mongodb.com
github.com/jrassi



Smart pointer pattern

- Using pointer-like objects to make programs simple and leak-free

Resource leaks

```
f(g(), new Kitten);
```

Assumptions

- This is a beginner talk
- You're familiar with use of basic C++ language features
  - Classes, templates, pointers, references, exceptions, operator overloading
- You don't know what a smart pointer is, or are curious to learn more about them
- You like kittens

Roadmap

- Pointers and ownership
- What is a smart pointer?
- Survey
  - boost::scoped_ptr
  - std::auto_ptr
  - std::unique_ptr
  - std::shared_ptr, std::weak_ptr

What is a pointer?

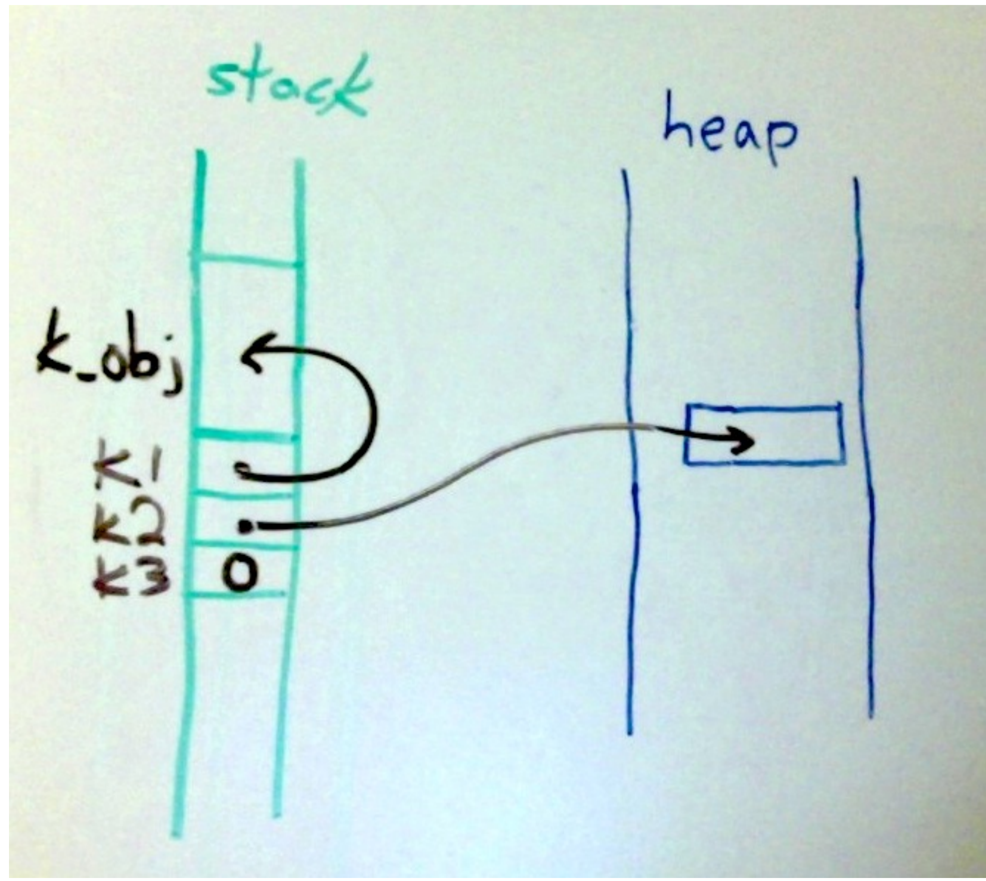
- Conceptually: a handle to an object
  - Implements indirection
  - Often can be re-bound to a different object
- Pointers ("bare pointers") in C/C++:
  - Associated with a specific type named in declaration
  - Generally are in one of two valid states
    - Point to an object
      - Value is the starting location of that object in the process' address space
    - Point to no objects
      - Value is null
  - Simple data type: no destructor

Kittens

- Properties of a Kitten
  - Lifetime is independent of lifetime of other objects in world
  - At any point in time, all Kittens must have [at least] one owner
    - Kitten with no owner is a leaked Kitten
    - Billions of Kittens with no owner => obvious system stability issues

Pointers in action

```
Kitten k_obj;
Kitten* k1 = &k_obj;
Kitten* k2 = new Kitten;
Kitten* k3 = nullptr;
```



What is ownership?

- Loose definition: responsibility for cleaning up after
- How do owners of these resources "clean up?"
  - Dynamically-allocated memory
  - File descriptors
  - Network connections
  - Locks

Ownership

- I found a pointer. Should I call delete on it?
  – Certainly not if the associated object has static or automatic lifetime
  – Certainly not if it's already been deleted
  – Certainly not if it's someone else's job to delete it
  – Certainly not if the associated object was allocated with a custom allocator

```
Kitten* k = a_mystery_function();
```

Ownership policies

- Can an owner give away the object?
  - Transferrable ownership
  - Non-transferrable ownership
- Can the object have multiple owners at once?
  - Shared ownership
  - Exclusive ownership

Exclusive, non-transferrable ownership

- Once owner, forever owner
  - Person acquires Kitten
  - Person never lets anyone else borrow or share Kitten
  - Kitten's lifetime must not outlast Person's lifetime

Exclusive, transferrable ownership

- One owner at a time, but owner can change
  - Person acquires Kitten
  - Person can use kitten-sitter for vacation:
    - Before vacation, person transfers ownership to kitten-sitter
    - After vacation, kitten-sitter transfers ownership back
  - Person can give Kitten up for adoption
  - Kitten's lifetime must not outlast its current owner (but can outlast previous owners)

Shareable ownership

- Set of owners changes over time
  - Person acquires Kitten
  - Ownership of Kitten shared with cohabitants, descendants
    - Owners added, removed over time
  - Kitten’s lifetime must not outlast that of its last remaining owner

Ownership with raw pointers

- Error-prone!
- Ownership transfer

```
find_new_owner(k);
k = nullptr;
// k not valid
```

- Ownership sharing

```
add_new_roommate(candidates[“alice”], k);
// k still valid
```

Pitfalls of bare pointers

- Suppose we're play-testing "t", a new Kitten toy
  - Kitten promises not to throw in play_with_toy()

```
{
    Kitten* k = kitten_factory();
    Rating r = k->play_with_toy(t);
    delete k;
    results.add_rating(r);
}
```

- Simple to write first pass, but simple to maintain?

Pitfalls of bare pointers

- What if the precondition on play_with_toy() changes?
- What if play_with_toy()'s exception specification changes?

```
{
     Kitten* k = kitten_factory();
     if (!k->interested_in_play()) {
          delete k;
          return;
     }
     Rating r;
     try {
          r = k->play_with_toy(t);
     }
     catch (CutenessException& e) {
          delete k;
          throw;
     }
     delete k;
     results.add_rating(r);
}
```

What happened?

- We're lazy
  - Putting "delete k" in the exact correct set of places is hard work
- What all those "delete k" statements are trying to express here:
  - We just want the Kitten object to be deleted before it goes out of scope

Roadmap

- ~~Pointers and ownership~~
- What is a smart pointer?
- Survey
  - boost::scoped_ptr
  - std::auto_ptr
  - std::unique_ptr
  - std::shared_ptr, std::weak_ptr

What is a “smart pointer?”

- Loose definition: object that behaves like a pointer, but somehow “smarter”
  - Major similarities to bare pointer:
    - Is bound to 0 or 1 objects at a time, often can be re-bound
    - Supports indirection: operator*, operator->
  - Major differences:
    - Has some “smart feature”

“Smart feature”

- Traditional smart pointers: resource management
  - Automatic deletion of the owned object
- Other “smart pointers”:
  - Iterators (think: *it, it->f(), it++)
  - Objects with pointer semantics that do something silly when dereferenced

“Smart feature”

- Traditional smart pointers: resource management
  - Automatic deletion of the owned object
- Other “smart pointers”:
  - Iterators (think: *it, it->f(), it++)
  - Objects with pointer semantics that do something silly when dereferenced

```
template<typename T>
class kittensay_ptr {
    T* ptr_;

public:
    explicit kittensay_ptr(T* t) : ptr_(t) {}

    T& operator*() {
        std::cout << ”Meow!\n";
        return *ptr_;
    }

    T* operator->() {
        std::cout << ”Meow!\n";
        return ptr_;
    }
};
```

kittensay_ptr

- In action:

```
int main() {
    Kitten k_obj;
    kittensay_ptr<Kitten> k(&k_obj);
    k->feed();
    return EXIT_SUCCESS;
}

$ ./a.out
Meow!
```

operator->()

- Unary operator
  - Even though it has a thing on the left and on the right
- Evaluating "f->method()" – pseudocode:
  - x := f
  - while x not a bare pointer:
    - x := x.operator->()
  - evaluate x->method()
- Temporary objects created in while loop

Resource Acquisition Is Initialization

- RAII is the pattern of:
  - Acquiring resources with a constructor
  - Releasing resources in a destructor
- Why?
  - Guarantees that the acquired resource is not leaked
  - Simpler code
- Smart pointer pattern is an application of the general RAII pattern

Smart pointers to the rescue

- Suppose we wrapped bare Kitten pointers in a MyKittenPtr object:
  - MyKittenPtr::MyKittenPtr(Kitten* k) : k_(k) {}
  - Kitten* MyKittenPtr::operator->() { return k_; }
  - MyKittenPtr::~MyKittenPtr() { delete k_; }

```
{
   MyKittenPtr k(kitten_factory());
   if (k->interested_in_play()) {
      Rating r = k->play_with_toy(t);
      results.add_rating(r);
   }
}
```

Smart pointers to the rescue

- Idea: templatize MyKittenPtr as MyPtr
  - MyPtr<T>::MyPtr(T* p) : p_(p) {}
  - T* MyPtr<T>::operator->() { return p_; }
  - MyPtr<T>::~MyPtr() { delete p_; }

```
{
    MyPtr<Kitten> k(kitten_factory());
    if (k->interested_in_play()) {
        Rating r = k->play_with_toy(t);
        results.add_rating(r);
    }
}
```

- MyPtr is now reusable

Standard smart pointers

- RAII class
- Stores a pointer to an object on the free store
  - Smart pointer owns the object
    - Hence responsible for deleting it
- Pointer semantics
- Self-documents the object's ownership policy

Roadmap

- ~~Pointers and ownership~~
- ~~What is a smart pointer?~~
- Survey
  - boost::scoped_ptr
  - std::auto_ptr
  - std::unique_ptr
  - std::shared_ptr, std::weak_ptr

boost::scoped_ptr

- Exclusive, non-transferrable ownership
  - Not copyable, not moveable
  - Once owner, forever owner
- Available in Boost, not C++ standard library
  - One of the first smart pointer classes in Boost
- How it gets its name:
  - Owned object guaranteed* to be deleted by the time scoped_ptr goes out of scope
    - *Unless swap() called – supporting swap() is arguably a design error

Common smart pointer methods

- void sp::reset(T* k = nullptr);
  - Replace my existing Kitten with a new Kitten
  - My former Kitten is deleted
- T* sp::get() const;
  - Expose to the caller a bare pointer to my Kitten, without changing ownership
    - For use with a legacy API only

```
template<typename T>
class scoped_ptr {
    T* px_;

    scoped_ptr(const scoped_ptr&);
    scoped_ptr& operator=(const scoped_ptr&);

public:
    explicit scoped_ptr( T* p = 0 ): px_( p ) {}

    ~scoped_ptr() { delete px_; }

    T& operator*() const { return *px_; }

    T* operator->() const { return px_; }

    T* get() const { return px_; }

    void swap(scoped_ptr& b) {
        T* tmp = b.px_;
        b.px_ = px_;
        px_ = tmp;
    }

    void reset(T* p = 0) { scoped_ptr<T>(p).swap(*this); }

};
```

boost::scoped_ptr

- Example

```
{
    boost::scoped_ptr<Kitten> a(new Kitten);
    a->feed();
    boost::scoped_ptr<Kitten> b;
    b = a;               // ERROR: will not compile.
    b.reset(new Kitten);
    b->feed();
    b.reset(new Kitten);
    b->feed();
}
```

boost::scoped_ptr

- One object only
  - Storing a pointer to a C-style array of objects: undefined behavior
  - If you must store an array, use boost::scoped_array
- No space overhead, almost no runtime overhead
  - sizeof(boost::scoped_ptr<T>) == sizeof(T*)
  - Inlined methods
    - Expands to nearly the same code as what you'd write with a bare pointer

std::auto_ptr

- Exclusive, transferrable ownership
- Introduced in C++98 as first standard-defined resource management smart pointer
- One object only
  - Incorrect to store a pointer to a C-style array in an auto_ptr
- Deprecated in C++11 in favor of std::unique_ptr

Common smart pointer methods

- T* sp::release();
  - Release ownership of my Kitten to the caller
  - Why not in boost::scoped_ptr?
    - Because boost::scoped_ptr provides non-transferrable ownership semantics

```
{
    std::auto_ptr<Kitten> a(new Kitten);
    std::auto_ptr<Kitten> b(a.release());
    // b now owns the Kitten.
    a->feed();                    // ERROR: null dereference.
    b->feed();                    // Kitten is fed.

    a.reset(b.release());
    // a owns the Kitten again.
    a->feed();                    // Kitten is fed.
    b->feed();                    // ERROR: null dereference.
}
```

std::auto_ptr

- What else is different from boost::scoped_ptr?
  - More ways to transfer: “copy” constructor and “copy” assignment operator
    - They take a non-const reference (!)
- What?

```
{
    std::auto_ptr<Kitten> a(new Kitten);
    std::auto_ptr<Kitten> b;
    b = a;                          // b now owns the Kitten.
    a->feed();                      // ERROR: null dereference.
    b->feed();                      // Kitten is fed.
}
```

Move versus copy

- Copy A to B: deep clone
  - Typically: member-wise deep copy
- Move A to B: rip the guts out of A and stuff them in B
  - Typically:
    - Member-wise shallow copy
    - Reset all of A’s members to default values

```
template<typename T>
class auto_ptr {
    T* px_;

public:
    explicit auto_ptr(T* p = 0) : px_(p) { }

    ~auto_ptr() { delete px_; }

    T& operator*() const { return *px_; }

    T* operator->() const { return px_; }

    T* get() const { return px_; }

...
```

```
...

   T* release() {
      T* tmp = px_;
      px_ = 0;
      return tmp;
   }

   void reset(T* p = 0) {
      if (p != px_) {
         delete px_;
         px_ = p;
      }
   }

   auto_ptr(auto_ptr& a) : px_(a.release()) { }

   auto_ptr& operator=(auto_ptr& a) {
      reset(a.release());
      return *this;
   }

};
```

std::auto_ptr

- You will be in hot water if you pass auto_ptr to a template function that executes “a = b;” and expects a copy
  - Compiler will make a valiant effort to prevent this by enforcing const-correctness
    - Only if developer didn’t forget to const-qualify…
- Can’t be put in standard containers

std::auto_ptr

- If:
  - You are able to use a compiler with C++11 support in your build system
- Then:
  - Do not use std::auto_ptr
  - Instead, use std::unique_ptr

std::unique_ptr

- Exclusive, transferrable ownership
- New in C++11
  - What std::auto_ptr always wanted to be, but the language didn't support

std::unique_ptr

- Can store a single pointer or a C-style array
  - Single pointer: unique_ptr<T>
  - Array: unique_ptr<T[]>
- Custom deleter: world of possibilities!
  - Single pointer: unique_ptr<T, D>
  - Array: unique_ptr<T[], D>
- So is unique_ptr's "move" any better than auto_ptr's?

Rvalue references lightning talk

- In C++11, rvalue reference (“T&&”) introduced
  - Similar to an lvalue reference (“T&”)
  - But different rules for binding to expressions:
    - Can be bound to a temporary
    - Can’t be bound to a “normal” lvalue without an explicit cast
- Gives rise to the following idiom
  - f(T&& k) { ... }
    - Caller of f is saying “Here, take k and do what you wish with it. I promise never to look at it again.”
    - “Move constructor”, “move assignment operator”
- Why relevant to this talk? Ownership transfer!

Rvalue references lightning talk

- std::move(x)
  - Poorly-named
  - Performs a cast
    - Casts argument to an rvalue reference
  - Does not perform a move
    - Generates zero code
- Move constructor and move assignment operator take a “T&&”
  - Hence the idiom “a = std::move(b);”
    - Overload resolution makes the compiler generate a call to “operator=(T&& t);”

std::unique_ptr

- Move: allowed
  - Move constructor:
    - unique_ptr(unique_ptr&& u);
  - Move assignment operator
    - unique_ptr& operator=(unique_ptr&& u);
- Copy: not allowed
  - No copy constructor:
    - unique_ptr(const unique_ptr&); // Not defined.
  - No copy assignment operator:
    - unique_ptr& operator=(const unique_ptr&); // Not defined.

```
/** Simplification: no custom deleter, no support for arrays. */
template<typename T>
class my_unique_ptr {
     T* px_;

public:
     explicit my_unique_ptr(T* p = 0) : px_(p) { }

     ~my_unique_ptr() { delete px_; }

     T& operator*() const { return *px_; }

     T* operator->() const { return px_; }

     T* get() const { return px_; }

     T* release() {
          T* tmp = px_;
          px_ = 0;
          return tmp;
     }

     void reset(T* p = 0) {
          if (p != px_) {
               delete px_;
               px_ = p;
          }
     }

...
```

```
...

public:
     my_unique_ptr(my_unique_ptr&& u)
          : px_(u.release()) { }

     my_unique_ptr& operator=(my_unique_ptr&& u) {
          reset(u.release());
          return *this;
     }

private:
     my_unique_ptr(const my_unique_ptr&);
     my_unique_ptr& operator=(const my_unique_ptr&);

};
```

std::unique_ptr

- What does assignment look like?

```
{
    std::unique_ptr<Kitten> a(new Kitten);
    std::unique_ptr<Kitten> b;
    b = a;              // ERROR: will not compile.
    b = std::move(a);   // b now owns the Kitten.
    a->feed();          // ERROR: null dereference.
    b->feed();          // Kitten is fed.
}
```

std::unique_ptr

- Good fit for a factory function

```
std::unique_ptr<Kitten> kitten_factory();
```

- Good fit for standard containers

```
std::vector<std::unique_ptr<Kitten>> v;
```

- Good fit for instance variable (if pointer semantics required)
  - No special handling needed for moveable classes
  - Special handling needed for copyable classes
- And most places where you would have a raw pointer to an object with dynamic lifetime
  - Except when shared ownership is absolutely necessary

std::unique_ptr custom deleter

- Pass a function object or function pointer as the second argument to the std::unique_ptr() ctor
- Allows storing of pointers to objects that need special freeing logic
  - Perhaps you're using a pool allocator
- Where is the deleter stored?
  - As additional private member data
  - In practice:
    - sizeof(std::unique_ptr<T, D>) > sizeof(T*)
    - sizeof(std::unique_ptr<T>) == sizeof(T*)

std::make_unique()

- Performs the allocation for you using new
- Accepted into C++14
- Never write a bare call to new again!

```
std::unique_ptr<Kitten> k(std::make_unique<Kitten>());
```

std::make_unique()

- What’s wrong with the bare “new” call?
- Consider:

```
void f(bool b, std::unique_ptr<Kitten> k);
```

- Is it possible for the Kitten to be leaked here?

```
f(g(), std::unique_ptr<Kitten>(new Kitten));
```

- What about here?

```
f(g(), std::make_unique<Kitten>());
```

std::shared_ptr

- Shared ownership
  - Thread-safe
- History
  - Started in Boost
    - boost::shared_ptr
  - Pulled into TR1
    - std::tr1::shared_ptr
  - Pulled into std namespace for C++11
    - std::shared_ptr

std::shared_ptr

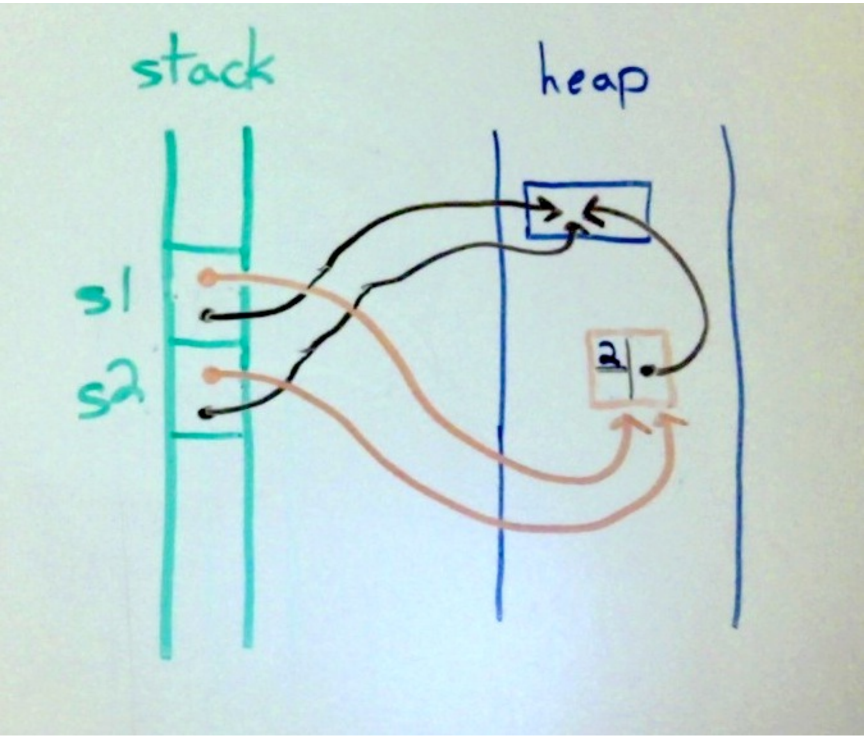
- Supports custom deleter
- Single object only
  - Use boost::shared_ptr (1.53 or newer) for arrays
    - boost::shared_ptr<T[]>, or
    - boost::shared_ptr<T[N]>
  - Older versions of boost: use boost::shared_array<T>

std::shared_ptr

- Reference-counting smart pointer
  - Not a garbage collector
- Reference count stored in dynamically-allocated *control block*
  - Control block allocated in “first owner” constructor, deallocated in “last owner” destructor
  - Reference count updated when a new owner is added or removed
    - Atomic increment/decrement instructions

std::shared_ptr control block

```
std::shared_ptr<Kitten> s1(new Kitten);
std::shared_ptr<Kitten> s2(s1);
```



std::shared_ptr

- When is associated object deleted?
  - shared_ptr destructor checks reference count
  - If no more owners, deletes object

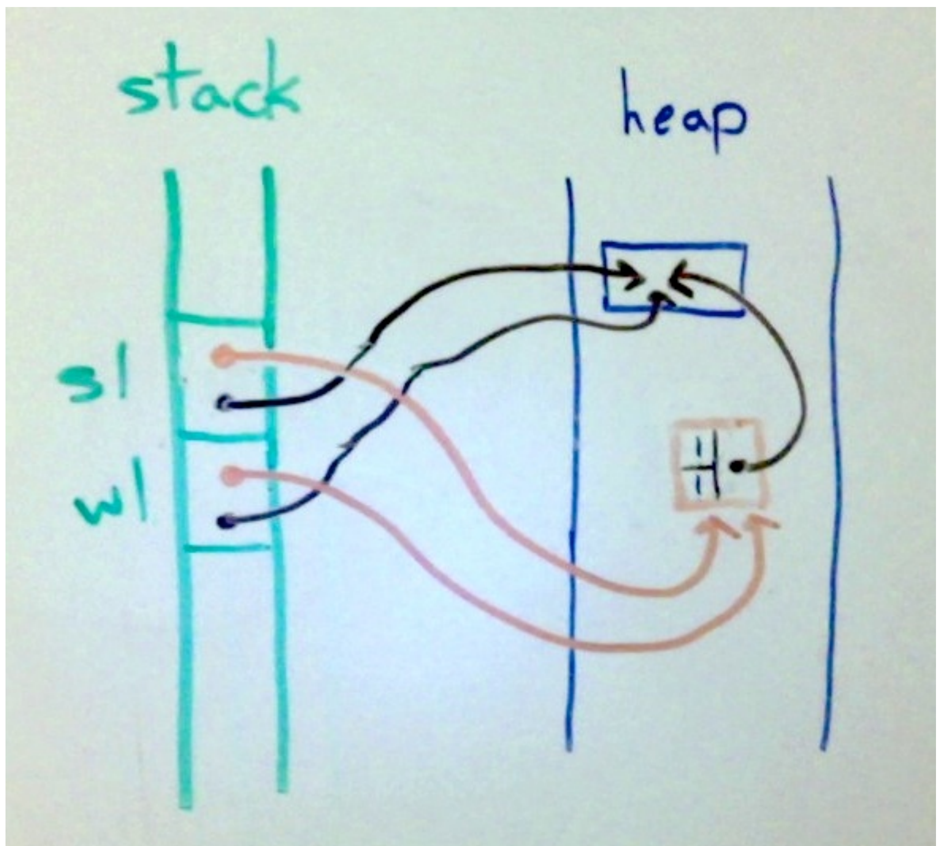
```
std::shared_ptr<Kitten> o1(new Kitten);
std::shared_ptr<Kitten> o2;
o2 = o1;              // Owners now: o1, o2.
o1 = nullptr;         // Owners now: o2.
o1 = std::move(o2); // Owners now: o1.
o1 = nullptr;         // No owners! Deletion now.
```

std::shared_ptr, std::weak_ptr

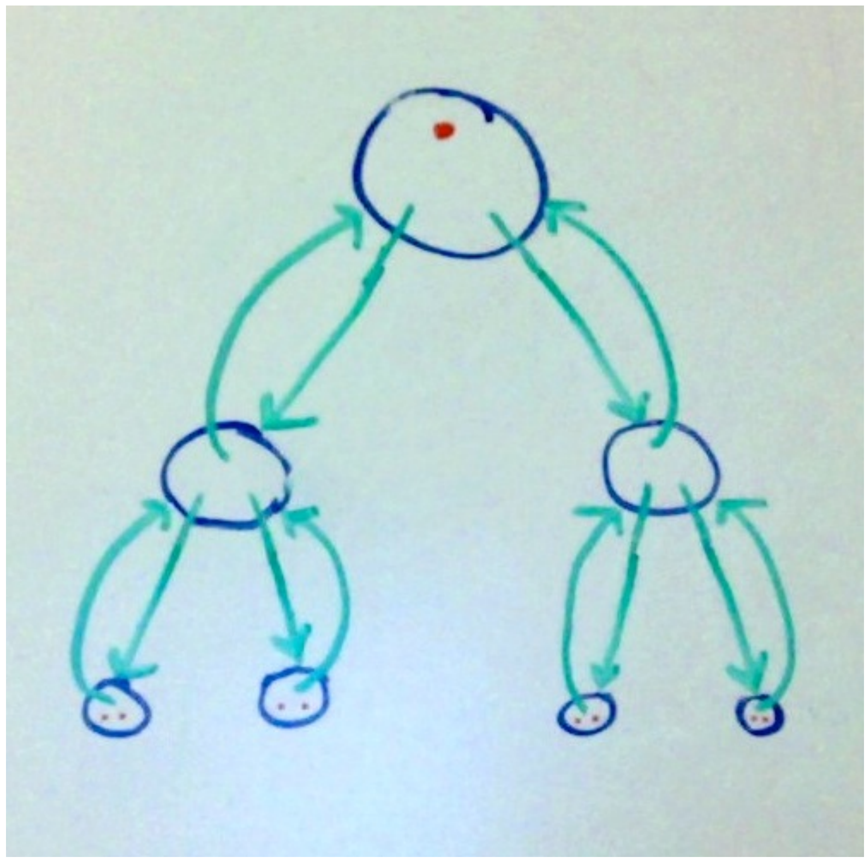
- shared_ptr to object: “strong owner” of object
- weak_ptr to same object: “weak owner”
  - Think “observer from afar”
  - Can’t access object directly: no operator->() etc.
- Control block contains “strong reference count” and “weak reference count”
  - Object deleted when strong count reaches zero
    - Even if weak owners still exist
  - Control block deleted when strong count and weak count reach zero
- Use weak_ptr for:
  - Statistics tracking
  - Breaking cycles of strong owners

Control block, now with std::weak_ptr

```
std::shared_ptr<Kitten> s1(new Kitten);
std::weak_ptr<Kitten> w1(s1);
```

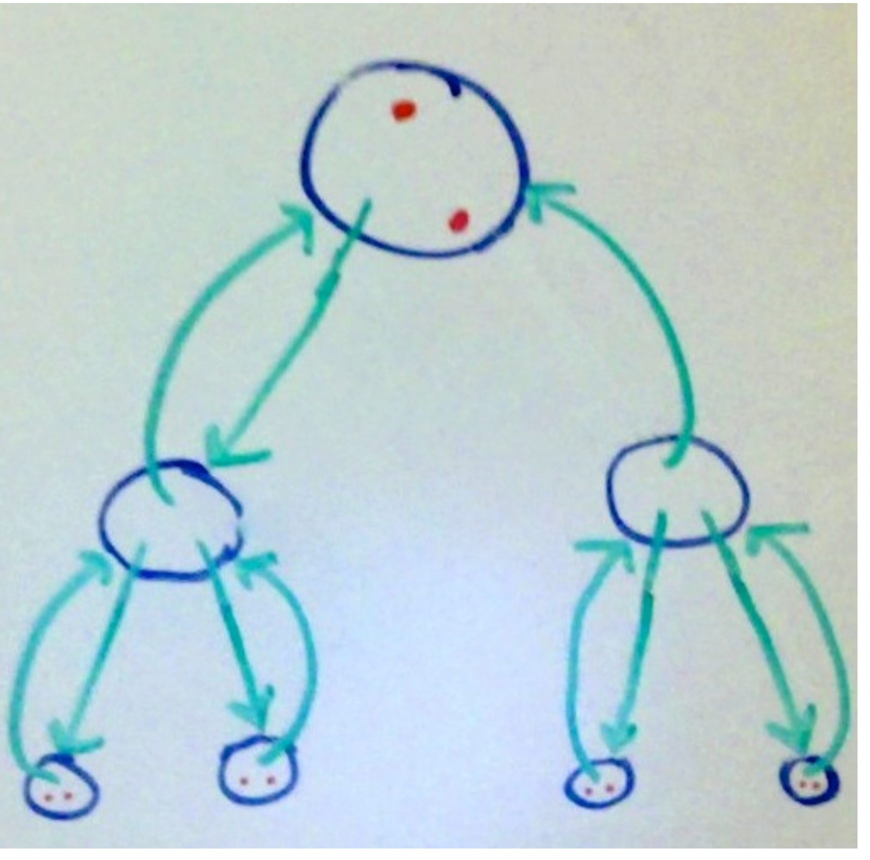


Ownership cycle: tree with shared_ptr back references

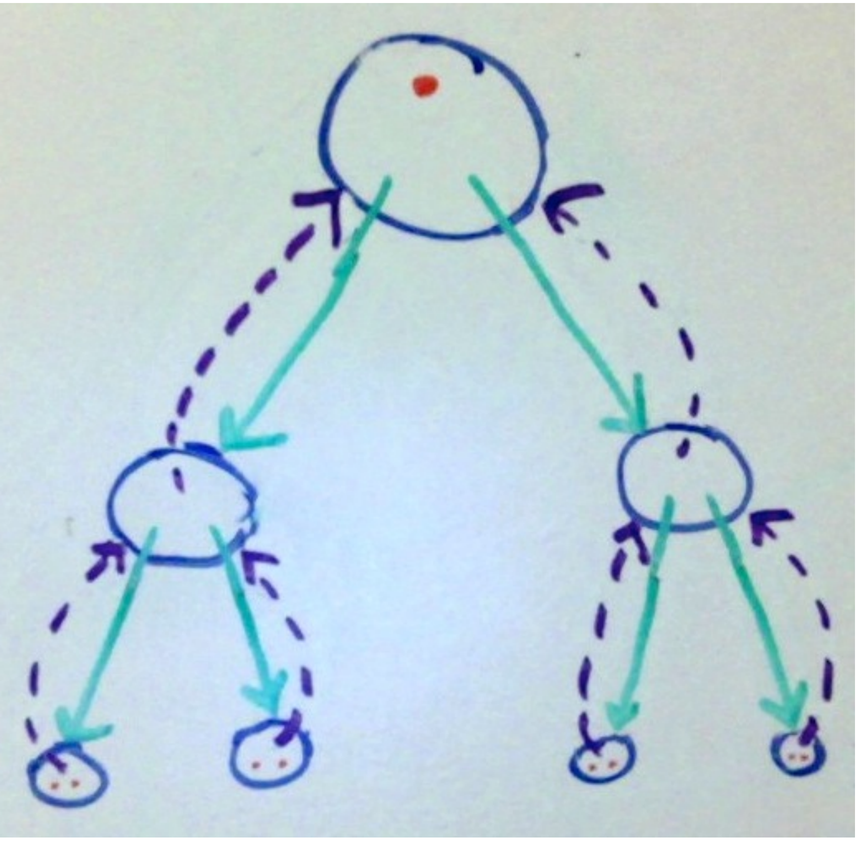


Ownership cycle: tree with shared_ptr back references

```
this.right_child_.reset(); // Sub-tree is leaked.
```

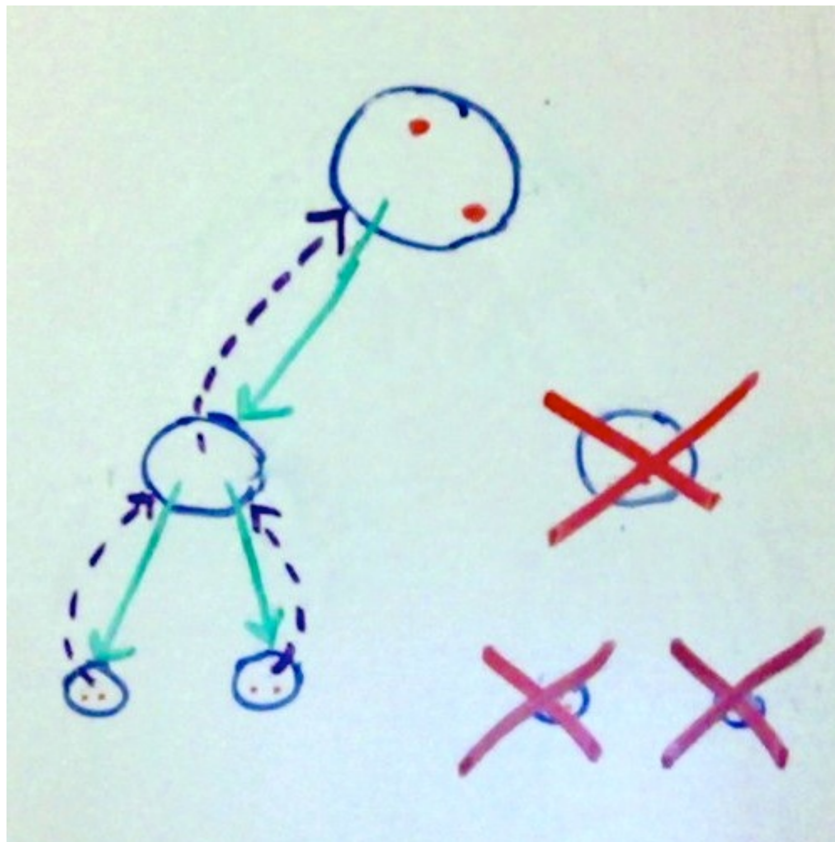


Breaking the cycle: tree with weak_ptr back references



Breaking the cycle: tree with weak_ptr back references

```
this.right_child_.reset(); // Sub-tree is deleted.
```



std::weak_ptr

- lock()
  - shared_ptr<T> weak_ptr<T>::lock() const;
  - Weak owners use this to request strong ownership
  - If object expired, returns empty shared_ptr<T>()
  - Otherwise, returns shared_ptr<T>(*this)

std::shared_ptr, std::weak_ptr

- In action:

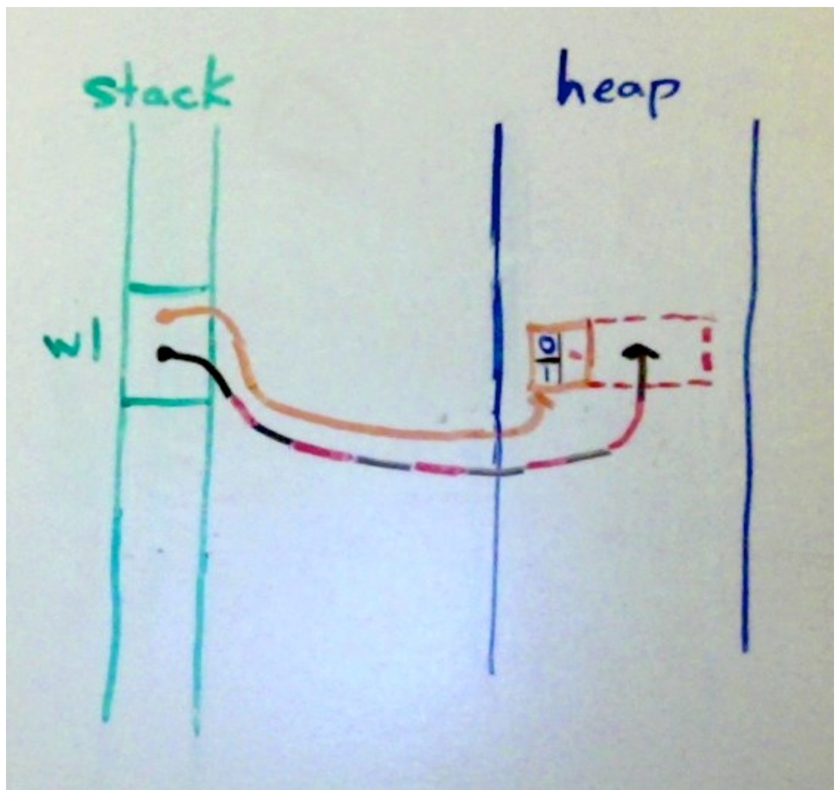
```
std::shared_ptr<Kitten> s1(new Kitten);
std::shared_ptr<Kitten> s2;
std::weak_ptr<Kitten> w;
w = s1;              // Owners now: s1, w.
s2 = w.lock();       // Owners now: s1, s2, w.
s1 = s2 = nullptr; // Owners now: w.  No strong owners!
                     // Kitten deletion now.
s2 = w.lock();       // Owners now: w.  s2 does not
                     // change.
w = nullptr;         // No owners at all!  Control block
                     // deallocation.
```

std::make_shared()

- std::make_unique() for shared pointers
  - Unlike std::make_unique(), this is actually in C++11
- std::make_shared() allocates the control block and the managed object in a single allocation
  - Pro: one allocation instead of two
  - Con: memory associated with object won't be freed until all owners (strong and weak) are gone

Control block, now with std::make_shared()

```
std::shared_ptr<Kitten> s1(std::make_shared<Kitten>());
std::weak_ptr<Kitten> w1(s1);
s1 = nullptr; // Kitten destroyed, memory still alloc.
```



Be wary of shared ownership

- “Do not design your code to use shared ownership without a very good reason”
  - “One such reason is to avoid expensive copy operations, but you should only do this if the performance benefits are significant, and the underlying object is immutable (i.e. shared_ptr<const Foo>)”
- “If you do use shared ownership, prefer to use shared_ptr”

Excerpted from the Google C++ Style Guide <http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml>

Review

	Ownership	Copyable? Moveable?	Availability
scoped_ptr	Exclusive	--	Boost
auto_ptr	Exclusive	Moveable… poorly	C++98
unique_ptr	Exclusive	Moveable	C++11
shared_ptr, weak_ptr	Shared	Copyable Moveable	Boost, TR1, C++11

boost/smart_ptr/

The smart pointer library provides six smart pointer class templates:

scoped_ptr	<boost/scoped_ptr.hpp>	Simple sole ownership of single objects. Noncopyable.
scoped_array	<boost/scoped_array.hpp>	Simple sole ownership of arrays. Noncopyable.
shared_ptr	<boost/shared_ptr.hpp>	Object ownership shared among multiple pointers.
shared_array	<boost/shared_array.hpp>	Array ownership shared among multiple pointers.
weak_ptr	<boost/weak_ptr.hpp>	Non-owning observers of an object owned by **shared_ptr**.
intrusive_ptr	<boost/intrusive_ptr.hpp>	Shared ownership of objects with an embedded reference count.

Further reading

- Meyers, Scott. *Effective C++, Third Edition*.
- Stroustrup, Bjarne. *The C++ Programming Language, Fourth Edition*.
- Sutter, Herb. *More Exceptional C++.*

Special thanks

- Andrew Morrow
- Charlie Page
- Greg Steinbruner

Resource leaks no more

```
f(g(),
  std::make_unique<Kitten>());
```