shared_ptr

Or: How I Learned To Stop Worrying
And Love Resource Management

Stephan T. Lavavej
Visual C++ Libraries Developer
What Does TR1 Contain?

- Boost Components
  - `shared_ptr` (and `weak_ptr`)
  - `mem_fn()`, `bind()`, and `function`
  - `regex`
  - `<random>`
  - "Containers": `tuple`, `array`, `unordered_set` (etc.)
  - `<type_traits>`
  - `reference_wrapper`

- C99 Compatibility (<`cstddef`>, etc.)

- Special Math Functions (`riemann_zeta()`, etc.)
What Are The Experts Saying?

  - "shared_ptr may be the most widely useful component in TR1."

- **C++ Coding Standards** (2005) by Herb Sutter and Andrei Alexandrescu:
  - "Store only values and smart pointers in containers. To this we add: If you use [Boost] and [C++TR104] for nothing else, use them for shared_ptr."
What Is shared_ptr?

- A templated...
- non-intrusive...
- deterministically reference-counted...
- smart pointer...
- (to a single object)...
- that works with polymorphic types...
- incomplete types...
- and STL containers (sequence and associative)!
Generalizing over types without forgetting types

shared_ptr<T>
  "shared pointer to T"

shared_ptr<const T>
  "shared pointer to const T"

const shared_ptr<const T>
  "const shared pointer to const T"

"Look, Mom! No backwards reading!"
You can instantiate `shared_ptr<T>` without modifying the definition of `T` (That is, the reference count is not embedded).

Huge usability benefit for minimal perf cost.

Works with built-in types: `shared_ptr<int>`

You can begin using `shared_ptr` in your codebase without having to modify your existing types.

You can stop using `shared_ptr` for a type without having to rip machinery out of it.

A type can be sometimes held by `shared_ptr` and sometimes contained by another type.
Deterministic
- shared_ptr s collectively share ownership of an object
- When the last shared_ptr dies, the object dies...
- Immediately!

Reference-Counted
- Directed acyclic graphs of shared_ptr s to objects containing shared_ptr s to other objects... are OKAY
- Cycles of shared_ptr s are LEAKTROCITY
- Someone else has to ultimately own you
- You can't own yourself!
shared_ptr Is A Smart Pointer

- **Smart**
  - Unlike auto_ptr, which was a stupid smart pointer
  - Sane copy constructor and copy assignment operator
  - Behaves like an ordinary value type
  - Pass and return by value and by reference as usual
  - Plays nice with const
- **Pointer**
  - Overloads operator*() and operator->()
  - Conversion function to _unspecified-bool-type_
  - if (sp) will compile, sp * 5 will not compile
- **No jagged metal edges!**
A single object, not an array!
- If you new up an array and hand it to a shared_ptr:
  - It will compile
  - It will trigger **UNDEFINED BEHAVIOR**
  - Which might mean **LEAKTROCITY** or **CRASHTROCITY**

If you need...
- a container: vector
- a shared container: shared_ptr<vector<T> >
- less overhead: shared_array (in Boost, but not TR1)
- or perhaps: shared_ptr<array<T, N> > (in TR1)

Custom deleters are insufficient (no op[ ])
shared_ptr Works With Polymorphic Types

- shared_ptr<Derived> is convertible to shared_ptr<Base>
- Works fine, doesn't screw up the reference count
- Need to convert back?
  - static_pointer_cast<Derived>(spBase)
  - dynamic_pointer_cast<Derived>(spBase)
- While we're at it...
  - const_pointer_cast<T>(spConstT)
- None of these throw exceptions!
- There is no reinterpret_pointer_cast
- Note: shared_ptr itself is not polymorphic
struct X;

void fxn(const shared_ptr<X>& p);

However, X must be complete by the time that you instantiate certain member functions of shared_ptr<X>, such as its constructor from X.*

Reason: If the constructor fails (e.g. to allocate memory for a reference count), it must delete the X before throwing, and deletion requires complete types in general.
**shared_ptr Works With Containers**

- **auto_ptr** is inherently an enemy of the STL
  - The STL loves ordinary value types
  - auto_ptr does not behave like an ordinary value type
  - Whoever wins, we lose

- **shared_ptr** is the STL's best friend
  - shared_ptr behaves like an ordinary value type
  - In fact, shared_ptr wraps non-values like noncopyable and polymorphic types in value's clothing
  - `vector<shared_ptr<Socket>>`
  - `vector<shared_ptr<Base>>`
  - Comes with `operator<()` for use in sets and maps
Policy Customizable
- Loki smart pointers are extremely customizable
- Ownership: refcount, remlink, destructive, etc.
- Implicit conversion to raw pointer: allow, disallow
- And so forth
- Policies are encoded in the smart pointer's type, preventing interoperability (sometimes, but not always, solvable with ninja template heroics)
- shared_ptr chooses good policies and bakes them in
- Deleters and allocators are customizable, as they don't affect the type
Containers of \texttt{shared\_ptr}

- \texttt{vector\<\texttt{shared\_ptr\<\texttt{NoncopyableResource}\>\>\>}
- \texttt{vector\<\texttt{shared\_ptr\<\texttt{PolymorphicBase}\>\>\>}
- Any other STL/TR1 containers, especially caches:
  - \texttt{map\<\texttt{Key, shared\_ptr\<\texttt{NoncopyableResource}\>\>}\>}

Passing around copyable but "heavy" objects efficiently (a simple version of move semantics)

Exception safety, superseding \texttt{auto\_ptr}

- Holding dynamically allocated objects at local scope
- Holding multiple dynamically allocated objects as members (what does this mean? See next slide...)

\texttt{shared\_ptr Use Cases (1/3)}
 Behold **LEAKTROCITY**:

```cpp
Foo::Foo() : m_p(0), m_q(0) {
    m_p = new X;
    m_q = new Y;
}
Foo::~Foo() {
    delete m_p;
    delete m_q;
}
```

**shared_ptr** **FIXES** the leak:

```cpp
Foo::Foo() : m_sp(new X), m_sq(new Y) {
} // Implicitly defined dtor is OK for these members
```
**shared_ptr Use Cases (3/3)**

- **Guidelines:**
  - All occurrences of `new[]/delete[]` should already have been replaced with `vector`
  - All occurrences of `new` should immediately be given to a named `shared_ptr`
  - All occurrences of `delete` should vanish

- **Exceptions:**
  - When implementing custom data structures like trees that can't be composed from the STL and TR1
  - When performance is absolutely critical

- Manual resource management is extremely difficult to do safely; consider it to be a last resort
Basic shared_ptr Use

shared_ptr<string> sp(new string("meow"));
cout << *sp << endl;
cout << sp->size() << endl;

- Prints:
  meow
  4

- Each new object is immediately given to a shared_ptr

- Each delete statement vanishes from the source
shared_ptr<string> sp = new string("meow");

- Compiler error (after substitution):
  error C2440: 'initializing' : cannot convert from
  'std::string *' to 'std::tr1::shared_ptr<std::string>'
  Constructor for class 'std::tr1::shared_ptr<std::string>'
  is declared 'explicit'

- Direct-initialization can use an explicit ctor

- Copy-initialization performs conversion: explicit ctors are unavailable

- shared_ptr acquires ownership explicitly
shared_ptr<int> a;
shared_ptr<int> b(new int(137));
cout << (a ? "a" : "X") << endl;
if (b) {
    cout << "b" << endl;
} else {
    cout << "Y" << endl;
}

- Prints:
  X
  b

- Also: if (!sp), if (sp && blah), if (sp || blah)
Using Other Smart Pointers In Conditionals

- auto_ptr: Not directly testable. Instead, you must test if (ap.get())
  - auto_ptr is deprecated in C++0x!
- unique_ptr: Has a conversion function to unspecified-bool-type just like shared_ptr
  - unique_ptr is the C++0x replacement for auto_ptr (not part of TR1)
- weak_ptr: Not directly testable. Instead, you must test if (!wp.expired())
  - weak_ptr usage is covered later in this presentation
```cpp
shared_ptr<int> foo(int n) {
    shared_ptr<int> r(new int(n));
    *r += 5;
    return r;
}

int main() {
    shared_ptr<int> p = foo(3);
    cout << *p << endl;
}
```

- Prints:
  8
shared_ptr<int> a(new int(1));
shared_ptr<int> b = a;
*a += 6;
cout << *a << "", " " << *b << endl;
a.reset();
cout << "a: " << (a ? "owns" : "empty") << endl;
cout << "b: " << (b ? "owns" : "empty") << endl;
cout << *b << endl;

- Prints:
  7, 7
  a: empty
  b: owns
  7
shared_ptr<int> frob(new int(100));
shared_ptr<const int> look = frob;
cout << *look << endl;
*frob /= 2;
cout << *look << endl;
// *look /= 2;

- **Prints:**
  100
  50

- **Uncomment the last line to get this compiler error:**
  error C3892: 'look' : you cannot assign to a variable that is const
cats.txt:
Abyssinian
Balinese
Chesire
Devon Rex

dogs.txt:
Alsatian
Beagle
Collie

people.txt:
Alan
Bjarne
Charles
Donald
Edsger
```cpp
queue<shared_ptr<ifstream>> q;

for (string s; getline(cin, s); ) {
    shared_ptr<ifstream> p(new ifstream(s.c_str()));
    q.push(p);
}

while (!q.empty()) {
    string s;

    if (getline(*q.front(), s)) {
        cout << s << endl;
        q.push(q.front());
    }

    q.pop();
}
```
cats.txt
dogs.txt
people.txt
^Z
Abyssinian
Alsatian
Alan
Balinese
Beagle
Bjarne
Chesire
Collie
Charles
Devon Rex
Donald
Edsger
class Animal {
public:
    explicit Animal(const string& name) : m_name(name) { }
    string noise() const {
        return m_name + " says " + noise_impl();
    }
    virtual ~Animal() { }
private:
    Animal(const Animal&);
    Animal& operator=(const Animal&);
    virtual string noise_impl() const = 0;
    string m_name;
};
class Cat : public Animal {
public: explicit Cat(const string& name) : Animal(name) { }
private: virtual string noise_impl() const { return "meow"; }
};

class Dog : public Animal {
public: explicit Dog(const string& name) : Animal(name) { }
private: virtual string noise_impl() const { return "woof"; }
};

class Pig : public Animal {
public: explicit Pig(const string& name) : Animal(name) { }
private: virtual string noise_impl() const { return "oink"; }
};
vector<shared_ptr<Animal> > v;

shared_ptr<Cat> c(new Cat("Garfield"));
shared_ptr<Dog> d(new Dog("Odie"));
shared_ptr<Pig> p(new Pig("Orson"));

v.push_back(c);
v.push_back(d);
v.push_back(p);

transform(v.begin(), v.end(),
    ostream_iterator<string>(cout, "\n"),
    mem_fn(&Animal::noise));
Garfield says meow
Odie says woof
Orson says oink
shared_ptr<Cat> p(new Cat("Peppermint"));
shared_ptr<Cat> c;
shared_ptr<Animal> a;
c = p;
a = p;
cout << c->noise() << endl;
cout << a->noise() << endl;

- Prints:
  Peppermint says meow
  Peppermint says meow
shared_ptr<Cat> p(new Cat("Peppermint"));
shared_ptr<Animal> a = p;
cout << (p == a ? "same" : "different") << endl;

- Prints:
  same
shared_ptr<Animal> a(new Cat("Bucky"));
cout << a->noise() << endl;
a.reset(new Dog("Satchel"));
cout << a->noise() << endl;

- Prints:
  Bucky says meow
  Satchel says woof
vector<T> reallocation conceptually involves copying Ts into the new memory block and destroying Ts from the old memory block

- Expensive when T is an STL container, etc.

VC8 detected when T was an STL container, and swapped from the old into the new memory block

- STL containers have O(1) nofail swaps

VC9 TR1 extends this to all TR1 types with swap() 

- All sane implementations of shared_ptr<T>::swap() never modify the reference counts
Swapping shared_ptr (1/2)

- shared_ptr has both member and free swap()
  - Just like STL containers
- swap() is intended to be implemented efficiently
  - In VC9 TR1, it is implemented efficiently
  - "Efficient" means not modifying the refcounts
- This is GOOD:

```cpp
shared_ptr<string> a(new string("meow")); // meow: 1
shared_ptr<string> b(new string("purr")); // purr: 1
a.swap(b); // meow: 1, purr: 1
swap(a, b); // meow: 1, purr: 1
```
Behold SLOWTROCYT:

```cpp
shared_ptr<string> a(new string("meow")); // meow: 1
shared_ptr<string> b(new string("purr")); // purr: 1
{
    shared_ptr<string> t(a); // ++meow: 2
    a = b; // --meow: 1, ++purr: 2
    b = t; // ++meow: 2, --purr: 1
} // --meow: 1
```

This unnecessarily modifies the refcounts 6 times
- Even worse, this dereferences pointers 6 times
- Even worse, this uses interlocked operations 6 times

Solution: Just use `swap()`
Getting T * From shared_ptr<T>

- Correct:
  shared_ptr<int> owning(new int(47));
  int * raw = owning.get();

- Incorrect:
  shared_ptr<int> owning(new int(47));
  int * raw = owning;

- Compiler error (after substitution):
  error C2440: 'initializing' : cannot convert from 'std::tr1::shared_ptr<int>' to 'int *'

  No user-defined-conversion operator available that can perform this conversion, or the operator cannot be called
Which statements contain **LEAKTROCITY**?

f1(shared_ptr<Foo>(new Foo(args)));
f2(shared_ptr<Foo>(new Foo(args)), g());
f3(shared_ptr<Foo>(new Foo(args)),
    shared_ptr<Bar>(new Bar(args)));

**Solution: Give each shared_ptr a name**

shared_ptr<Foo> foo(new Foo(args));
shared_ptr<Bar> bar(new Bar(args));
f1(foo);
f2(foo, g());
f3(foo, bar);
void foo() {
    shared_ptr<int> sp(new int(1729));
    int * raw = sp.get();
    delete raw;
}

- Result: **DOUBLE DELETION**
- Unlike auto_ptr, shared_ptr has no release() member function
- get() returns a non-owning raw pointer
struct Ansible {
   shared_ptr<Ansible> get_shared() {
      shared_ptr<Ansible> ret(this);
      return ret;
   }
};

int main() {
   shared_ptr<Ansible> a(new Ansible);
   Ansible& r = *a;
   shared_ptr<Ansible> b = r.get_shared();
}

- Result: **DOUBLE DELETION**
struct Ansible
  : public enable_shared_from_this<Ansible> {};

int main() {
    shared_ptr<Ansible> a(new Ansible);
    Ansible& r = *a;
    shared_ptr<Ansible> b = r.shared_from_this();
}

- a and b share ownership, as if:
  shared_ptr<Ansible> b = a;
shared_ptr<int> a(new int(2161));
shared_ptr<const int> b(a);
shared_ptr<int> c(const_cast<int*>(b.get()));

- **Result:** DOUBLE DELETION

- **Solution:** Use `const_pointer_cast`

  shared_ptr<int> c(const_pointer_cast<int>(b));

- `static_pointer_cast`, `dynamic_pointer_cast`, and `const_pointer_cast` exist for correctness, not convenience.
shared_ptr's Little Helper: weak_ptr

void observe(const weak_ptr<int>& wp) {
    shared_ptr<int> t = wp.lock();
    cout << (t ? *t : 2010) << endl;
}

weak_ptr<int> wp;
{
    shared_ptr<int> sp(new int(1969));
    wp = sp;
    observe(wp);
}
observe(wp);

Prints:
1969
1969
2010
2010
**shared_ptr Thread Safety**

- **Read:** Any operation that can be performed to a const `shared_ptr` (copying, dereferencing, etc.)
- **Write:** Any operation that cannot be performed to a const `shared_ptr` (assigning, resetting, swapping, etc.)
- Destruction counts as a write
- Multiple threads can simultaneously read a single `shared_ptr` object
- Multiple threads can simultaneously read/write different `shared_ptr` objects
  - **Even when the objects are copies that share ownership**
- Anything else triggers **UNDEFINED BEHAVIOR**
- Both VC9 TR1 and Boost provide these guarantees
shared_ptr Deleters

- **shared_ptr's** `ctor` and `reset()` can take an additional "deleter" argument
- A deleter is a functor that will be called with the stored raw pointer to release the owned object
- Simplest example: `free()`
- The deleter's actual type is forgotten
  - As if through inheritance
- The deleter stays with the owned object
  - NOT with the `shared_ptr`
Allocator support is a C++0x feature (not in TR1)
- Implemented by VC9 TR1 and Boost 1.35

`shared_ptr<T>` gains a three-arg ctor and `reset()`
- Taking `(T *, Deleter, Allocator)`

The third argument:
- Must be an STL allocator (20.1.5 lists the requirements)
- Will be rebound (you can pass `YourAlloc<int>`) 
- Will be used to allocate/deallocate the reference count

The allocator's actual type is forgotten
- As if through inheritance

The allocator stays with the owned object
- **NOT** with the `shared_ptr`
shared_ptr and weak_ptr contain two raw pointers:
- Pointer to owned object (used for dereferencing)
- Pointer to _Ref_count_base

_Ref_count_base contains:
- Pointer to owned object (used for deleting)
- 32-bit strong refcount (# of shared_ptrs)
- 32-bit weak refcount (# of weak_ptrs + 1 for all shared_ptrs)

When the strong refcount falls to zero:
- _Ref_count deletes the owned object
- _Ref_count_d uses its stored deleter to nuke the owned object
- Both decrement the weak refcount

When the weak refcount falls to zero:
- _Ref_count deletes itself
- _Ref_count_d uses its stored allocator to nuke itself

Takeaways:
- shared_ptr is reasonably small
- Dereferencing a shared_ptr involves ZERO OVERHEAD
Powered by variadic templates and rvalue references:

```cpp
template <class T, class... Args>
    shared_ptr<T> make_shared(Args&&... args);
```

Convenient!

```cpp
shared_ptr<LongTypeName> p(new LongTypeName(stuff));
// Becomes:
auto p(make_shared<LongTypeName>(stuff));
```

Safe!
- Fixes the classic pitfall of shared_ptr temporaries

FAST! Say goodbye to intrusive refcounting!
- Stores the object and its refcount in the same memory block
Destructors encapsulate resource release

Destructors are resource agnostic
- Memory, files, sockets, locks, textures, etc.

Destructors are executed deterministically

STL containers enabled "one owning many"

shared_ptr enables "many owning one"

<table>
<thead>
<tr>
<th>Object Category</th>
<th>Owned By Their</th>
<th>Destroyed When</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic</td>
<td>Block</td>
<td>Control Leaves Block</td>
</tr>
<tr>
<td>Data Members</td>
<td>Parent</td>
<td>Parent Dies</td>
</tr>
<tr>
<td>Elements</td>
<td>Container</td>
<td>Container Dies</td>
</tr>
<tr>
<td>Dynamically Allocated</td>
<td>shared_ptrs</td>
<td>All shared_ptrs Die</td>
</tr>
</tbody>
</table>
Questions?

- For more information, see:
  - The TR1 draft: tinyurl.com/361wqe
  - The C++ Standard Library Extensions: A Tutorial And Reference by Pete Becker: tinyurl.com/27jv8n
  - Improving shared_ptr For C++0x, Revision 2: tinyurl.com/2dlw3v
    - Allocator Support, Aliasing Support, Object Creation, and Move Support were voted into the C++0x Working Paper
  - Improving shared_ptr For C++0x, Revision 1: tinyurl.com/36cty7
    - Atomic Access and Cycle Collection are still planned

Version 1.2 - March 17, 2008