Smart Pointers

ITP 435 – Spring 2016
Week 4, Lecture 2
Lecturer: Sanjay Madhav

USC Viterbi
School of Engineering

University of Southern California

Memory Leaks 101

- Forgetting to deallocate memory is a very common mistake in C/C++ code

```
int main()
{
    Square* mySquare = new Square();

    // OOPS!
    return 0;
}
```

Memory Leaks 102

- Sometimes the leaks can be caused by exceptions:

```
bool badStuff = false;
Square* mySquare = new Square();

if (badStuff)
{
   throw std::exception();
}

// Deallocate, but what happens if there's a throw?
delete mySquare;
```

Memory Leaks 201

- Worse is when there's confusion between who should delete what:

```
void doStuff(Shape* shape) {
   // Do stuff...
   // Done with shape, so delete it!
   delete shape;
}

int main() {
   Square* mySquare = new Square();
   doStuff(mySquare);
   delete mySquare;

   return 0;
}
```

Uh-oh…

Smart Pointer

- A ***smart pointer*** is an object that encapsulates dynamically allocated data

- There are several variations of smart pointers, some providing more general usage than others

Scoped Pointer

- We'll define a ***scoped pointer*** as a smart pointer that encapsulates dynamic data for the single scope in which it's used

- This will solve the basic Memory Leaks 101/102 problems:
  - Forgetting to deallocate
  - Exception bypassing a delete statement

ScopedPtr – Minimal Declaration


```
template <typename T>
class ScopedPtr {
public:
   // Construct based on pointer to dynamic object
   explicit ScopedPtr(T* obj);

   // Destructor (clean up memory)
   ~ScopedPtr();

   // Overload dereferencing * and ->
   T& operator*();
   T* operator->();
private:
   // Disallow assignment/copy
   ScopedPtr(const ScopedPtr& other);
   ScopedPtr& operator=(const ScopedPtr& other);

   // Track the dynamically allocated object
   T* mObj;
};
```

ScopedPtr – Constructor/Destructor


```
// Construct based on pointer to dynamic object
template <typename T>
ScopedPtr<T>::ScopedPtr(T* obj)
: mObj(obj)
{
}

// Destructor (clean up memory)
template <typename T>
ScopedPtr<T>::~ScopedPtr()
{
   // Delete dynamically allocated object
   delete mObj;
}
```

ScopedPtr – Operators


```
// Overloaded dereferencing
template <typename T>
T& ScopedPtr<T>::operator*()
{
   return *mObj;
}

template <typename T>
T* ScopedPtr<T>::operator->()
{
   return mObj;
}
```

ScopedPtr in Action


```
int main() {
   // Construct a scoped pointer to a newly-allocated object
   ScopedPtr<Square> mySquare(new Square());

   // Can call functions just like a regular pointer!
   mySquare->Draw();

   // No delete necessary
   return 0;
}
```

USC Viterbi School of Engineering

University of Southern California

When mySquare is destructed, the underlying dynamic data will automatically be deleted. Keep in mind the destructor will automatically be called in the event of an exception

Reference Counting Pointer

- ***Reference counting*** is a method in which the number of references to a dynamically allocated object is tracked
- When the references hit zero, the object will be automatically deallocated
- **Important!!!** – This is different from garbage collection (in a language such as Java) because there is a well-defined and consistent point where it will deallocate

This should fix the memory leaks 201 problem

Control Block

- A reference counting pointer needs a ***control block*** – another dynamically allocated object which tracks the number of references

- All instances of `RefCountPtr` that point to the same object will also point to the same control block

RefCountPtr – Minimal Declaration


```
// Declare the control block
struct ControlBlock {
   unsigned mRefCount;
};

template <typename T>
class RefCountPtr {
public:
   // Construct based on pointer to dynamic object
   explicit RefCountPtr(T* obj);
   // Allow copy constructor
   RefCountPtr(const RefCountPtr& other);
   // Destructor (reduce ref count)
   ~RefCountPtr();
   // Overload dereferencing * and ->
   T& operator*();
   T* operator->();
private:
   // Disallow assignment (for simplicity)
   RefCountPtr& operator=(const RefCountPtr& other);
   // Pointer to dynamically allocated object
   T* mObj;
   // Pointer to control block
   ControlBlock* mBlock;
};
```

RefCountPtr – Basic Constructor


```
// Construct based on pointer to dynamic object
template <typename T>
RefCountPtr<T>::RefCountPtr(T* obj)
   : mObj(obj)
{
   // Dynamically allocate a new control block
   mBlock = new ControlBlock;
   // Initially, one reference (self)
   mBlock->mRefCount = 1;
}
```

RefCountPtr – Copy Constructor


```
// Copy constructor
template <typename T>
RefCountPtr<T>::RefCountPtr(const RefCountPtr<T>& other)
{
    // Grab object and control block from other
    mObj = other.mObj;
    mBlock = other.mBlock;

    // Increment ref count
    mBlock->mRefCount += 1;
}
```

RefCountPtr – Destructor


```
// Destructor (reduce ref count)
template <typename T>
RefCountPtr<T>::~RefCountPtr()
{
   // Decrement ref count
   mBlock->mRefCount -= 1;

   // If there are zero references, delete the object
   // and the control block
   if (mBlock->mRefCount == 0) {
      delete mObj;
      delete mBlock;
   }
}
```

RefCountPtr in Action


```
int main() {
   // Construct a RefCountPtr to a newly-allocated object
   RefCountPtr<Square> mySquare(new Square());

   mySquare->Draw();

   {
      RefCountPtr<Square> mySquareTwo(mySquare);

      // This will call Draw on the same underlying square
      mySquareTwo->Draw();
   }

   return 0;
}
```

RefCountPtr in Action


```
int main() {
   // Construct a RefCountPtr to a newly-allocated object
   RefCountPtr<Square> mySquare(new Square());

   mySquare->Draw();

   {
      RefCountPtr<Square> mySquareTwo(mySquare);

      // This will call Draw on the same underlying square
      mySquareTwo->Draw();
   }

   return 0;
}
```

Construct, set ref count to 1

RefCountPtr in Action


```
int main() {
   // Construct a RefCountPtr to a newly-allocated object
   RefCountPtr<Square> mySquare(new Square());

   mySquare->Draw();

   {
      RefCountPtr<Square> mySquareTwo(mySquare);

      // This will call Draw on the same underlying square
      mySquareTwo->Draw();
   }

   return 0;
}
```

Construct as copy, increment ref count (now 2)

RefCountPtr in Action


```
int main() {
   // Construct a RefCountPtr to a newly-allocated object
   RefCountPtr<Square> mySquare(new Square());

   mySquare->Draw();

   {
      RefCountPtr<Square> mySquareTwo(mySquare);

      // This will call Draw on the same underlying square
      mySquareTwo->Draw();
   }

   return 0;
}
```

mySquareTwo destructed, decrement ref count (now 1)

RefCountPtr in Action


```
int main() {
   // Construct a RefCountPtr to a newly-allocated object
   RefCountPtr<Square> mySquare(new Square());

   mySquare->Draw();

   {
      RefCountPtr<Square> mySquareTwo(mySquare);

      // This will call Draw on the same underlying square
      mySquareTwo->Draw();
   }

   return 0;
}
```

mySquare destructed, decrement ref count (now 0), so delete underlying object

Using RefCountPtr w/ Functions


```
// Smart pointers should almost always be passed by value
void doStuff(RefCountPtr<Shape> shape) {
    shape->Draw();
}

int main() {
    // Construct a scoped pointer to a newly-allocated object
    RefCountPtr<Shape> myShape(new Square());

    doStuff(myShape);

    return 0;
}
```

Problems w/ RefCountPtr

- **Q:** What if class A has a RefCountPtr to class B, and class B has a RefCountPtr to class A?
- **A:** Circular references means neither will ever be deleted
- **Q:** What if you want to have an "observer" that can observe the RefCountPtr but not affect the number of references?
- **A:** It's currently not possible

Weak Pointer

- A ***weak pointer*** is a pointer that keeps a weak reference to a reference counter pointer

- A ***weak reference*** does not affect the lifetime of the object, because it uses a separate count (this is in contrast to the references we had before, which were ***strong references***)

Adding Weak References

- First, add a weak reference count to `ControlBlock`:

```
// Declare the control block
struct ControlBlock {
   unsigned mRefCount;
   unsigned mWeakRefCount;
};
```

- Also make WeakRefPtr a friend of RefCountPtr

Adding Weak References, Cont'd

- Update constructor for RefCountPtr:

```
// Construct based on pointer to dynamic object
template <typename T>
RefCountPtr<T>::RefCountPtr(T* obj)
    : mObj(obj)
{
    // Dynamically allocate a new control block
    mBlock = new ControlBlock;
    // Initially, one reference (self)
    mBlock->mRefCount = 1;
    // No weak references to start
    mBlock->mWeakRefCount = 0;
}
```

Adding Weak References, Cont'd

- The destructor should now only delete the control block if both regular and weak references are 0:

```
// Destructor (reduce ref count)
template <typename T>
RefCountPtr<T>::~RefCountPtr()
{
   // Decrement ref count
   mBlock->mRefCount -= 1;

   // If there are zero references, delete the object
   // and the control block
   if (mBlock->mRefCount == 0) {
      delete mObj;
      if (mBlock->mWeakRefCount == 0) {
         delete mBlock;
      }
   }
}
```

Declaration of WeakRefPtr


```
template <typename T>
class WeakRefPtr {
public:
   // Constructor accepts RefCountPtr
   explicit WeakRefPtr(RefCountPtr<T>& refPtr);
   // Allow copy constructor
   WeakRefPtr(const WeakRefPtr& other);
   // Destructor
   ~WeakRefPtr();
   // Overload dereferencing * and ->
   T& operator*();
   T* operator->();
   // Determines whether or not the object is alive
   bool isAlive();
private:
   // Disallow assignment (for simplicity)
   WeakRefPtr& operator=(const WeakRefPtr& other);
   // Pointer to dynamically allocated object
   T* mObj;
   // Pointer to control block
   ControlBlock* mBlock;
};
```

WeakRefPtr – Constructor


```
// Constructor accepts RefCountPtr
template <typename T>
WeakRefPtr<T>::WeakRefPtr(RefCountPtr<T>& refPtr)
{
    // Copy object/block from RefCountPtr
    mObj = refPtr.mObj;
    mBlock = refPtr.mBlock;

    // Increment weak reference count
    mBlock->mWeakRefCount += 1;
}
```

WeakRefPtr – Copy Constructor


```
// Copy constructor
template <typename T>
WeakRefPtr<T>::WeakRefPtr(const WeakRefPtr<T>& other)
{
    mObj = other.mObj;
    mBlock = other.mBlock;

    // Increment weak reference count
    mBlock->mWeakRefCount += 1;
}
```

WeakRefPtr – Destructor


```
// Destructor
template <typename T>
WeakRefPtr<T>::~WeakRefPtr()
{
   // Decrement weak reference count
   mBlock->mWeakRefCount -= 1;

   // If both strong and weak references are 0,
   // delete control block
   if (mBlock->mRefCount == 0 &&
       mBlock->mWeakRefCount == 0) {
      delete mBlock;
   }
}
```

WeakRefPtr – isAlive


```
template <typename T>
bool WeakRefPtr<T>::isAlive()
{
   // Only alive if strong ref count is not 0
   // (Because if it hits 0, object is destroyed)
   return (mBlock->mRefCount != 0);
}
```

WeakRefPtr – Dereferencing


```
template <typename T>
T& WeakRefPtr<T>::operator*()
{
   if (!isAlive()) {
      throw std::exception();
   }
   return *mObj;
}

template <typename T>
T* WeakRefPtr<T>::operator->()
{
   if (!isAlive()) {
      throw std::exception();
   }
   return mObj;
}
```

WeakRefPtr in Action


```
WeakRefPtr<Shape> makeShapeWeak() {
   // Construct a RefCountPtr
   RefCountPtr<Shape> myShape(new Square());

   WeakRefPtr<Shape> weakShape(myShape);

   weakShape->Draw();

   // Return a WeakRefPtr to the shape
   return weakShape;
}

int main() {
   WeakRefPtr<Shape> weakPtr(makeShapeWeak());

   // Try to access weak reference here
   weakPtr->Draw();

   return 0;
}
```


The weakPtr dereference inside main will throw an exception, because the strong ref count hit 0

A Question…

- So if you want to use smart pointers, do you have to declare your own?

- In C++98, the answer was yes ☹ (more or less)

- But in C++11 the answer is ***no!***

Smart Pointers in C++ 11

`std::unique_ptr` (similar to `ScopedPtr`)

- A pointer that allows only a single reference at a time
- When it's out of scope, delete the dynamically allocated object
- Should create with `std::make_unique`

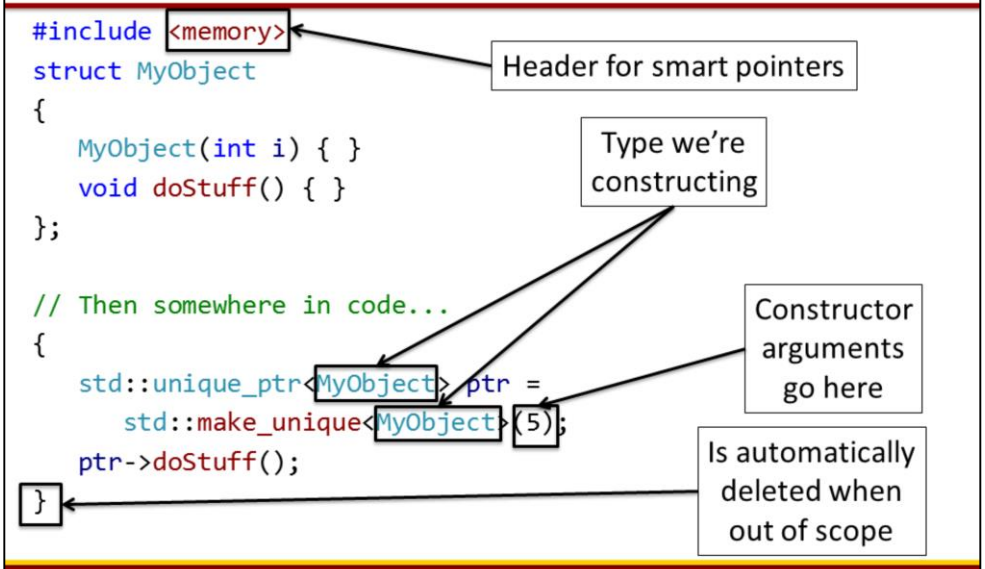
`std::shared_ptr` (similar to `RefCountPtr`)

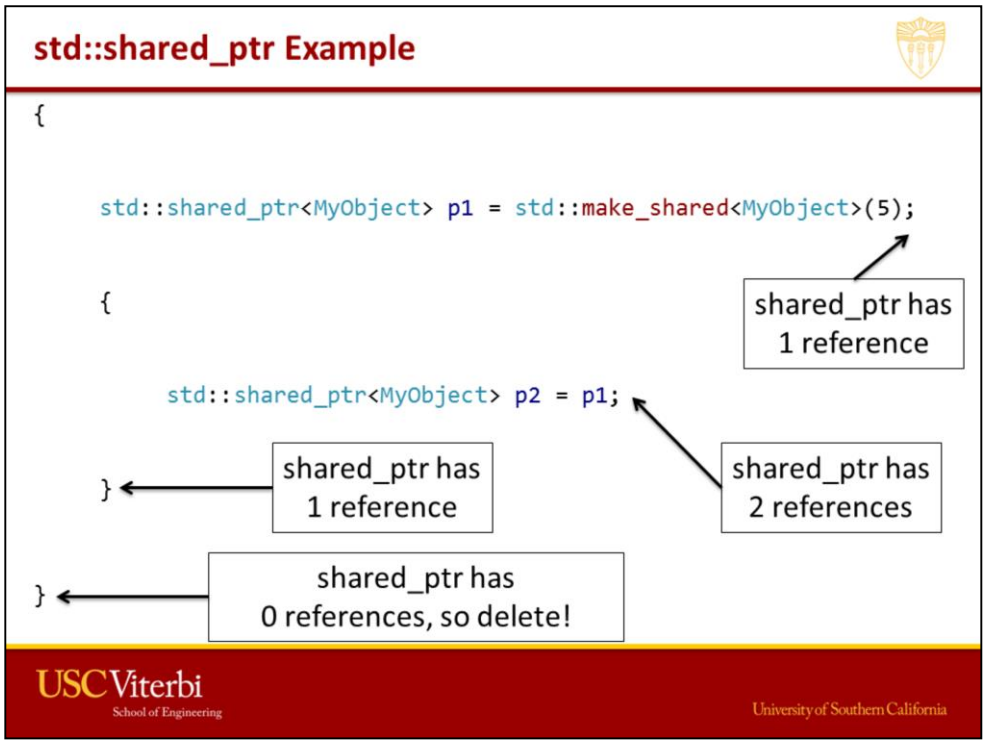
- Allows multiple references at once
- When it goes out of scope, decrement the reference
- If references hit 0, delete the dynamically allocated object
- Should create with `std::make_shared`

`std::weak_ptr` (similar to `WeakPtr`)

- Allows for weak references to shared pointers

std::unique_ptr Example





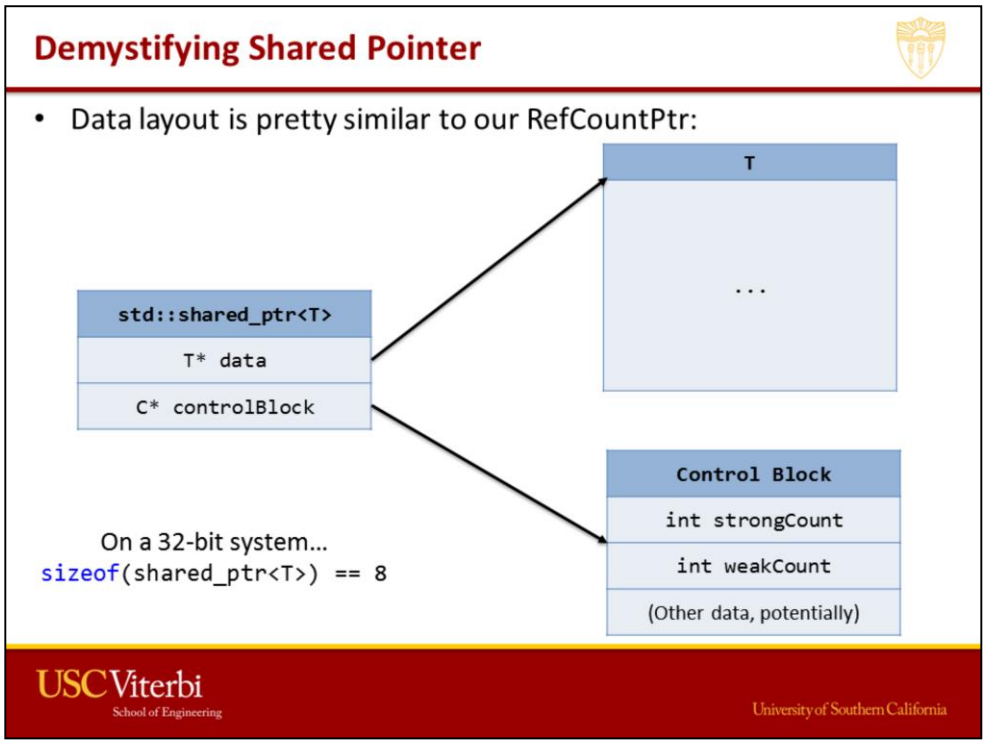
So for shared_ptrs, we use make_shared

A more complex shared_ptr example


```
struct A
{ };
struct B
{
B(std::shared_ptr<A> ptr)
   : mPtr(ptr)
   { }
private:
   std::shared_ptr<A> mPtr;
};

// Then...
std::shared_ptr<B> ptrB;
{
   std::shared_ptr<A> ptrA = std::make_shared<A>();
   ptrB = std::make_shared<B>(ptrA);
}
```

What happens here?



In most implementations, a smart pointer is sizeof(void*) x 2.

make_shared

- Technically, you don't ***have*** to use make_shared (or make_unique).

- Eg:

```
std::shared_ptr<MyObject> ptr(new MyObject(5));
```

- Problems:
  - This requires ***two*** heap allocations (one for MyObject and one for the shared_ptr)
  - What if there is an exception?

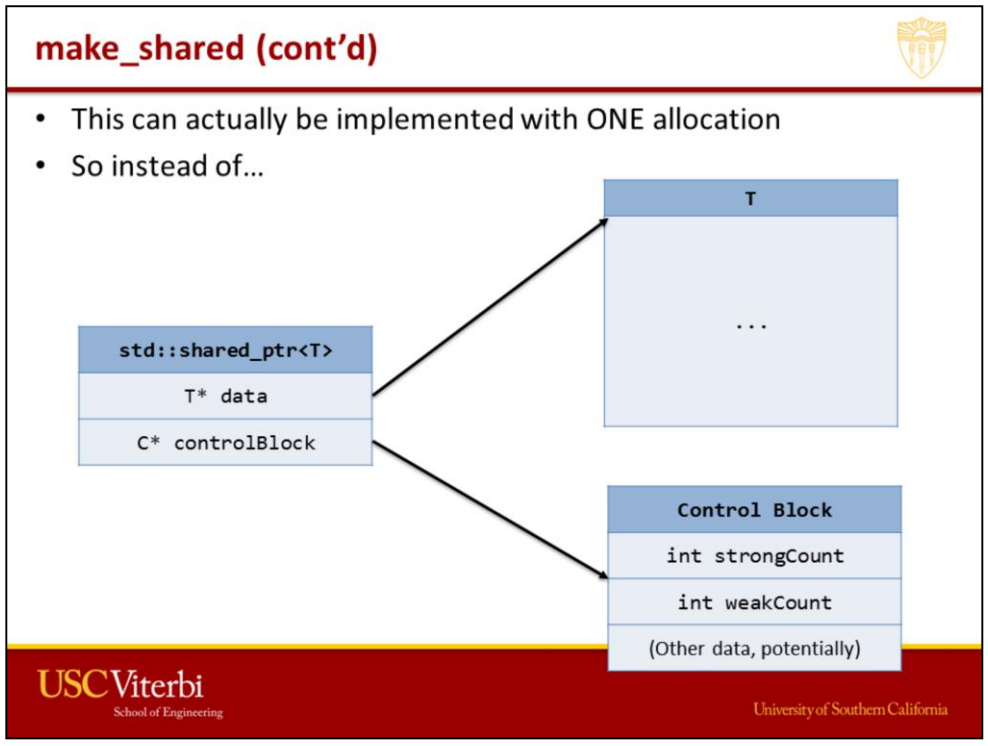
Note: You can't use make_shared in instances where the constructor is protected/private (looking at you Gdiplus::Pen!!)

Other advantage of make_shared/unique

- If you initialize a smart pointer std::make_shared/unique, you can declare your pointer as an auto and its type will be deduced correctly:

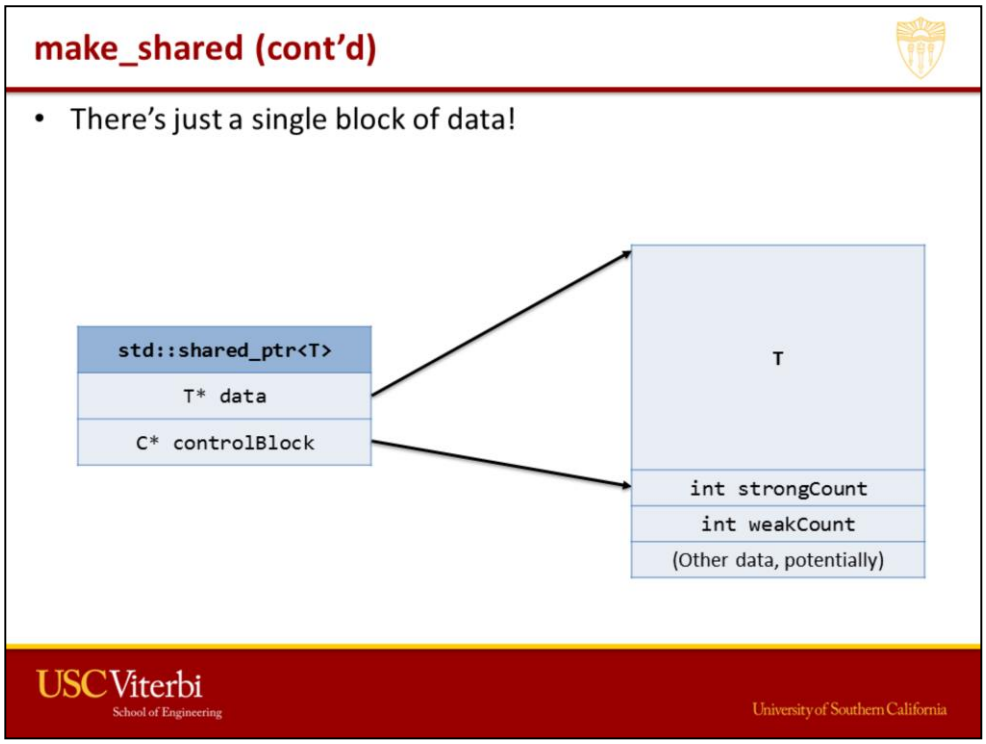
```
// auto will automatically be a std::shared_ptr<A>
auto ptrA = std::make_shared<A>();
```

make_shared (cont'd)



- This can actually be implemented with ONE allocation
- So instead of…

make_shared (cont'd)



- There's just a single block of data!

std::weak_ptr in Action


```
std::shared_ptr<Shape> myShape = std::make_shared<Square>();

{
   // Make a weak ptr to this
   std::weak_ptr<Shape> myWeakPtr(myShape);

   // expired is roughly the equivalent of isAlive
   if (!myWeakPtr.expired()) {
      // lock will create a shared_ptr that references the object
      // (this can be used to temporarily "acquire" the object)
      std::shared_ptr<Shape> ptrB = myWeakPtr.lock();
   }
}
```

std::weak_ptr in Action


```
std::shared_ptr<Shape> myShape = std::make_shared<Square>();

{
   // Make a weak ptr to this
   std::weak_ptr<Shape> myWeakPtr(myShape);

   // expired is roughly the equivalent of isAlive
   if (!myWeakPtr.expired()) {
      // lock will create a shared_ptr that references the object
      // (this can be used to temporarily "acquire" the object)
      std::shared_ptr<Shape> ptrB = myWeakPtr.lock();
   }
}
```

Strong Count = 1
Weak Count = 0

std::weak_ptr in Action


```
std::shared_ptr<Shape> myShape = std::make_shared<Square>();

{
   // Make a weak ptr to this
   std::weak_ptr<Shape> myWeakPtr(myShape);

   // expired is roughly the equivalent of isAlive
   if (!myWeakPtr.expired()) {
      // lock will create a shared_ptr that references the object
      // (this can be used to temporarily "acquire" the object)
      std::shared_ptr<Shape> ptrB = myWeakPtr.lock();
   }
}
```

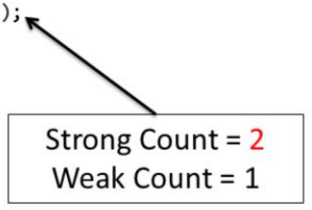
Strong Count = 1
Weak Count = 1

std::weak_ptr in Action


```
std::shared_ptr<Shape> myShape = std::make_shared<Square>();

{
   // Make a weak ptr to this
   std::weak_ptr<Shape> myWeakPtr(myShape);

   // expired is roughly the equivalent of isAlive
   if (!myWeakPtr.expired()) {
      // lock will create a shared_ptr that references the object
      // (this can be used to temporarily "acquire" the object)
      std::shared_ptr<Shape> ptrB = myWeakPtr.lock();
   }
}
```



std::weak_ptr in Action


```
std::shared_ptr<Shape> myShape = std::make_shared<Square>();

{
   // Make a weak ptr to this
   std::weak_ptr<Shape> myWeakPtr(myShape);

   // expired is roughly the equivalent of isAlive
   if (!myWeakPtr.expired()) {
      // lock will create a shared_ptr that references the object
      // (this can be used to temporarily "acquire" the object)
      std::shared_ptr<Shape> ptrB = myWeakPtr.lock();
   }
}
```

Strong Count = 1
Weak Count = 1

std::weak_ptr in Action


```
std::shared_ptr<Shape> myShape = std::make_shared<Square>();

{
   // Make a weak ptr to this
   std::weak_ptr<Shape> myWeakPtr(myShape);

   // expired is roughly the equivalent of isAlive
   if (!myWeakPtr.expired()) {
      // lock will create a shared_ptr that references the object
      // (this can be used to temporarily "acquire" the object)
      std::shared_ptr<Shape> ptrB = myWeakPtr.lock();
   }
}
```

Strong Count = 1
Weak Count = 0

One Complexity – Custom Deleters

- By default, shared_ptrs just use the basic delete

- What if you want a shared_ptr to a dynamically allocated array?

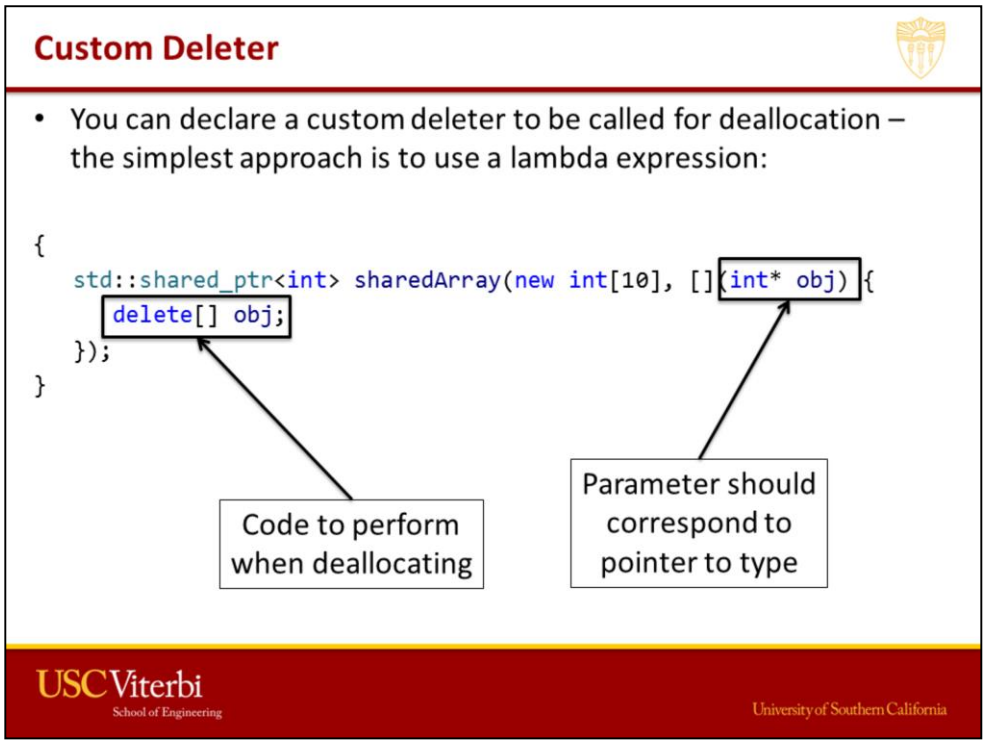
```
{
   std::shared_ptr<int> sharedArray(new int[10]);
}
```

USC Viterbi School of Engineering

University of Southern California

This is a problem, because it'll only call delete and not delete[]

Custom Deleter



- You can declare a custom deleter to be called for deallocation – the simplest approach is to use a lambda expression:

```
{
   std::shared_ptr<int> sharedArray(new int[10], [](int* obj) {
      delete[] obj;
   });
}
```

Code to perform when deallocating

Parameter should correspond to pointer to type

This won't leak memory anymore

Custom Deleters, Cont'd

- When you use a custom deleter, you ***can't*** use make_shared
- In the prior example, as opposed to using a shared_ptr to an array, it may be better to just use an STL data structure
- However, custom deleters can be useful in the instance where there's some very specific deinitialization you must perform

When to Use Smart Pointers?

- Unless the code is ***really*** low level and you ***really*** need the highest possible performance you should probably use smart pointers!

- The additional cost of using a smart pointer, especially if you stick to make_unique and make_shared, is not terribly high

Which Smart Pointer?

- If there's only one owner, ever, use std::unique_ptr
- If there's multiple possible owners, use std::shared_ptr
- Usage of std::weak_ptr is more rare

- For more information on when to pick which one, you can read the following in *Effective Modern C++*:
  - Item 18: Use std::unique_ptr for exclusive-ownership resource management
  - Item 19: Use std::shared_ptr for shared-ownership resource management
  - Item 20: Use std::weak_ptr for std::shared_ptr-like pointers that can dangle