Smart Pointers

ITP 435
Week 4, Lecture 1

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

Smart Pointers in C++ 11

`std::unique_ptr`

– A pointer that allows only a single reference at a time

`std::shared_ptr`

– Allows multiple references at once

`std::weak_ptr`

– Allows for weak references to shared pointers

- *Although these are built-in STL classes, we will look at how they are implemented under-the-hood both to demystify them and to understand their uses and limitations!*

The “Rule of zero”

There’s also the “rule of zero” which says that in modern C++, you shouldn’t have to overload any of the five member functions!

This can only be the case if you avoid using `new` altogether and instead use:

- STL collections
- Smart pointers

We’ll get here eventually

Unique Pointer

- A ***unique pointer*** is a pointer that uniquely controls the lifetime of an object
- When the unique pointer goes out of scope, the object is deleted
- This solves the basic Memory Leaks 101/102 problems:
  - Forgetting to deallocate
  - Exception bypassing a delete statement

UniquePtr – Minimal Declaration

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

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

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

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

UniquePtr – Constructor/Destructor

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

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

UniquePtr – Operators

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

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

UniquePtr in Action

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

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

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

Shared Pointer

- A ***shared pointer*** is used when an object has shared ownership between multiple pointers
- Uses ***reference counting*** to track the number of references to the 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

Control Block

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

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

SharedPtr – Minimal Declaration

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

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

SharedPtr – Basic Constructor

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

SharedPtr – Copy Constructor

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

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

SharedPtr – Destructor

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

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

SharedPtr in Action

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

    mySquare->Draw();

    {
        SharedPtr<Square> mySquareTwo(mySquare);

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

    return 0;
}
```

SharedPtr in Action

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

    mySquare->Draw();

    {
        SharedPtr<Square> mySquareTwo(mySquare);

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

    return 0;
}
```

Construct, set ref count to 1

SharedPtr in Action

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

    mySquare->Draw();

    {
        SharedPtr<Square> mySquareTwo(mySquare);

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

    return 0;
}
```

Construct as copy, increment ref count (now 2)

SharedPtr in Action

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

    mySquare->Draw();

    {
        SharedPtr<Square> mySquareTwo(mySquare);

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

    return 0;
}
```

mySquareTwo destructed, decrement ref count (now 1)

SharedPtr in Action

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

   mySquare->Draw();

   {
      SharedPtr<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 SharedPtr w/ Functions

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

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

   doStuff(myShape);

   return 0;
}
```

Problems w/ SharedPtr

- **Q:** What if class A has a SharedPtr to class B, and class B has a SharedPtr 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 SharedPtr 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 shared 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 mShared;
   unsigned mWeakShared;
};
```

- Also make WeakPtr a friend of SharedPtr

Adding Weak References, Cont'd

- Update constructor for SharedPtr:

```
// Construct based on pointer to dynamic object
template <typename T>
SharedPtr<T>::SharedPtr(T* obj)
    : mObj(obj)
{
    // Dynamically allocate a new control block
    mBlock = new ControlBlock;
    // Initially, one reference (self)
    mBlock->mShared = 1;
    // No weak references to start
    mBlock->mWeakShared = 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>
SharedPtr<T>::~SharedPtr()
{
    // Decrement ref count
    mBlock->mShared -= 1;

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

Declaration of WeakPtr

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

WeakPtr – Constructor

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

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

WeakPtr – Copy Constructor

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

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

WeakPtr – Destructor

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

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

WeakPtr – isAlive

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

WeakPtr – Dereferencing

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

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

WeakPtr in Action

```
WeakPtr<Shape> makeShapeWeak() {
   // Construct a SharedPtr
   SharedPtr<Shape> myShape(new Square());

   WeakPtr<Shape> weakShape(myShape);

   weakShape->Draw();

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

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

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

   return 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 `UniquePtr`)

- 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 `SharedPtr`)

- 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

```
#include <memory>
struct MyObject
{
   MyObject(int i) { }
   void doStuff() { }
};

// Then somewhere in code...
{
   std::unique_ptr<MyObject> ptr =
      std::make_unique<MyObject>(5);
   ptr->doStuff();
}
```

Header for smart pointers

Type we're constructing

Constructor arguments go here

Is automatically deleted when out of scope

std::shared_ptr Example

```
{
    std::shared_ptr<MyObject> p1 = std::make_shared<MyObject>(5);
    {
        std::shared_ptr<MyObject> p2 = p1;
    }
}
```

shared_ptr has 1 reference

shared_ptr has 2 references

shared_ptr has 1 reference

shared_ptr has 0 references, so delete!

A more complex shared_ptr example

```cpp
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);
}
```

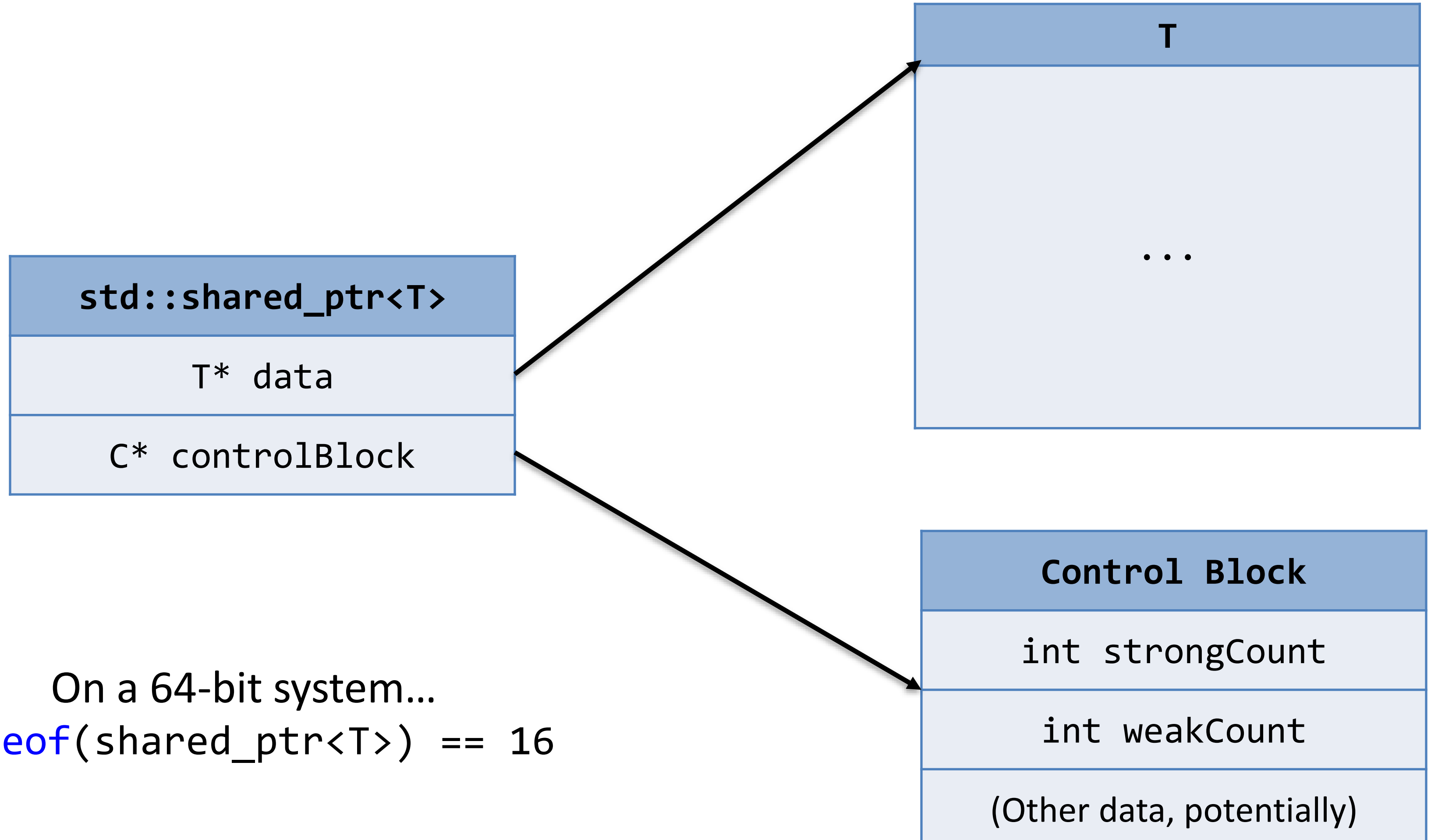
What happens here?

In-class activity

- Let’s review

Demystifying Shared Pointer

- Data layout is pretty similar to our SharedPtr:



On a 64-bit system…

```
sizeof(shared_ptr<T>) == 16
```

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?

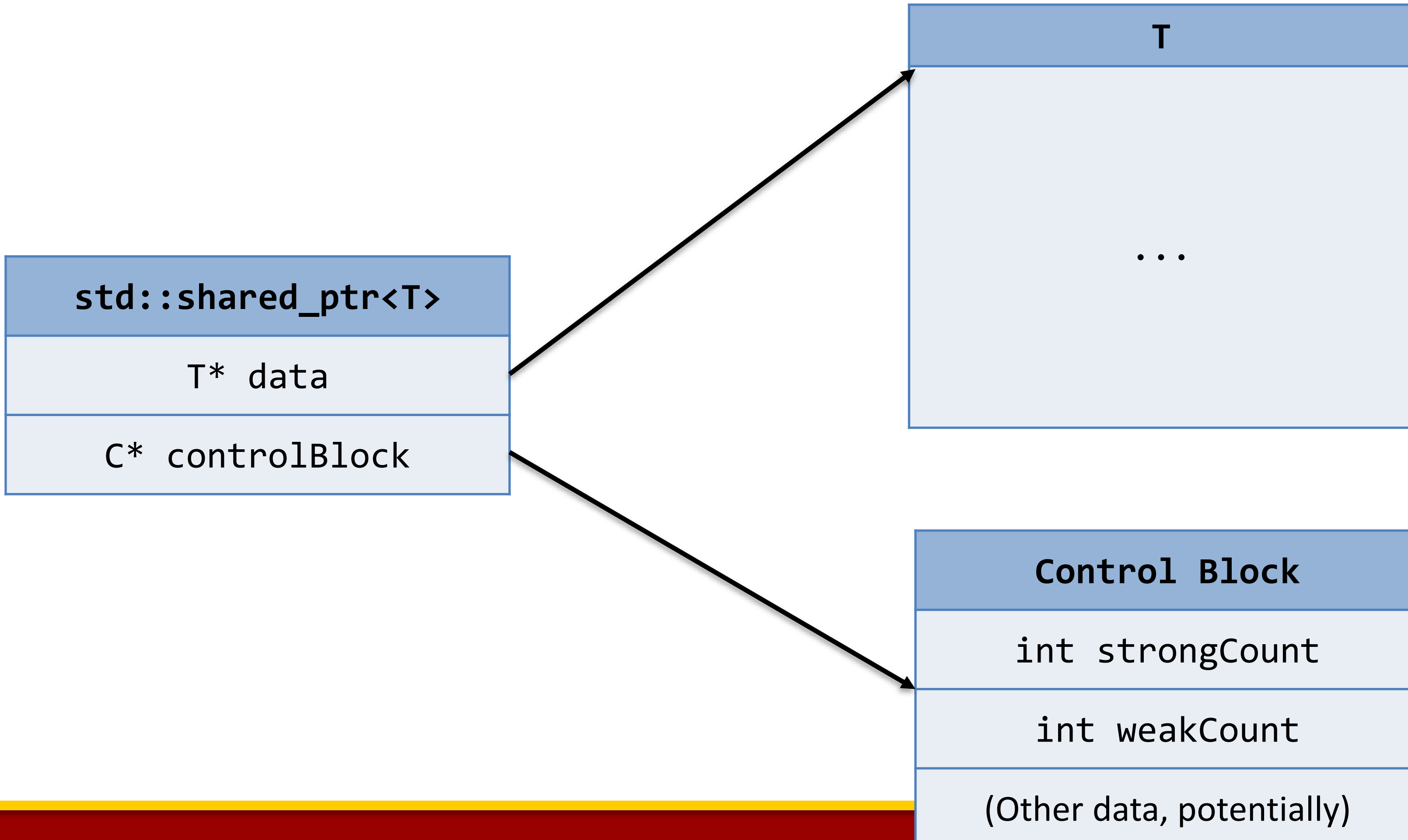
Other advantage of make_shared/unique

- 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>();
```

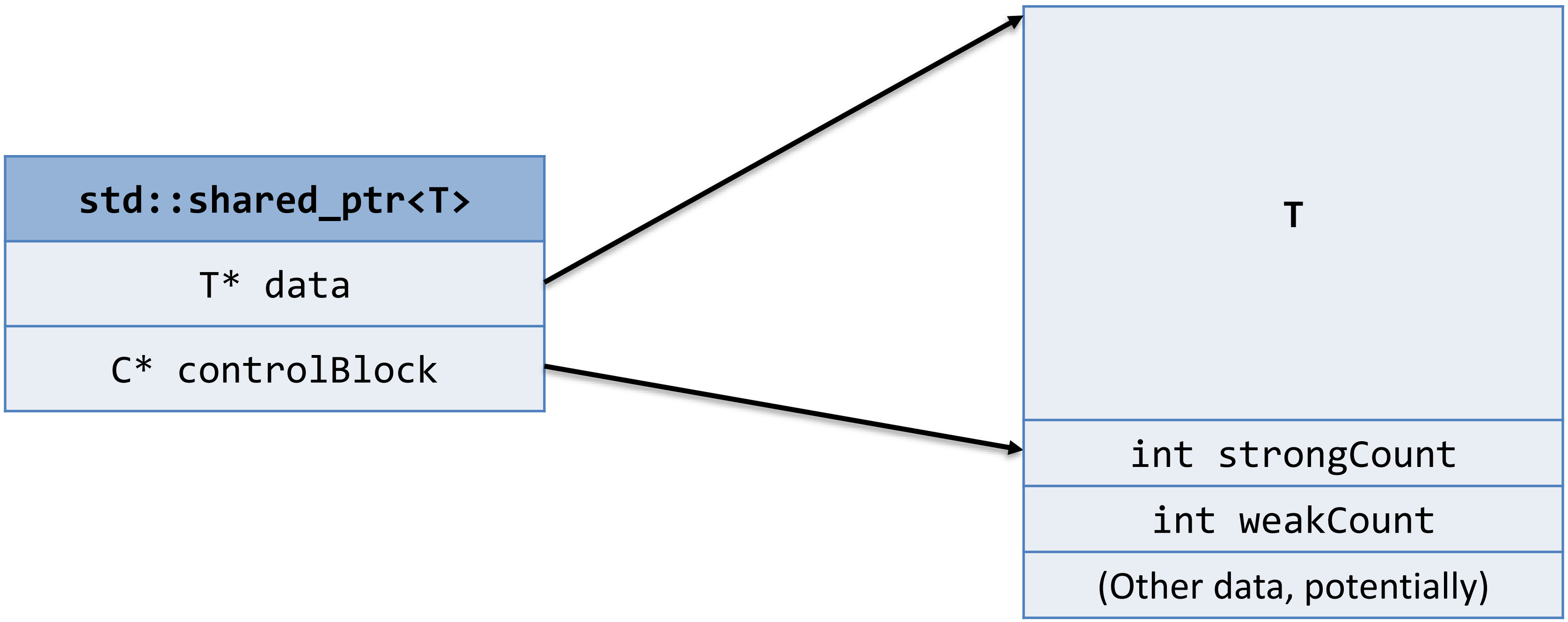
make_shared (cont'd)

- 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();
   }
}
```

Strong Count = 2
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 = 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]);
}
```

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

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

Unique Array?

- Unique pointer has a templated version that takes in an array, which you can use like this:

```
std::unique_ptr<int[]> uniqueArray(new int[10]);
```

Performance Cost of Smart Pointers?

- Unique pointers have negligible overhead
- Shared pointers have overhead because of thread safety guarantees

When to Use Smart Pointers?

- Unless the code is ***really*** low level and you ***really*** need the highest possible performance unique pointers are good to use!

- Shared pointers are not as free to use, but shared ownership is also not needed that often

Which Smart Pointer?

- Most of the time, 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