Memory Management / C++ Smart Pointers

CSE 333 Winter 2021

Instructor: John Zahorjan

Teaching Assistants:

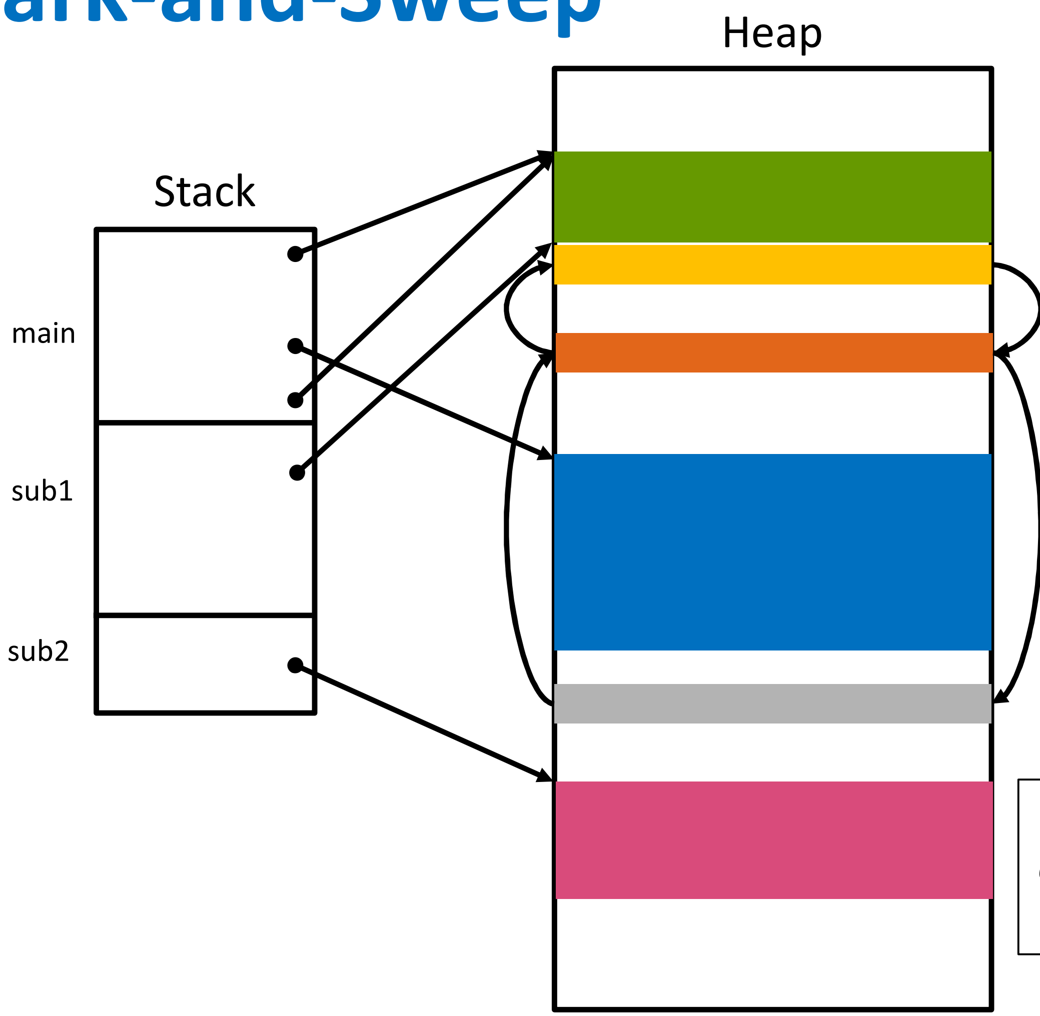
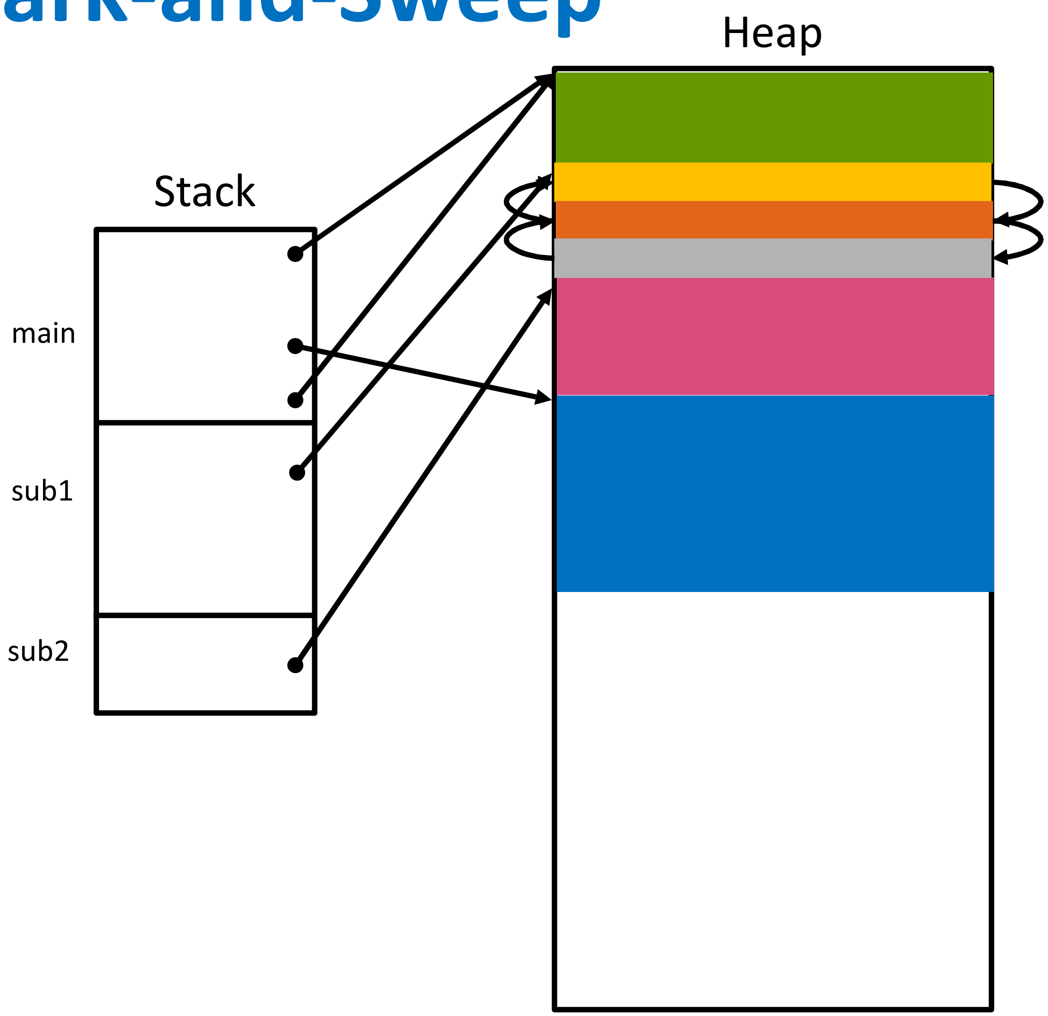
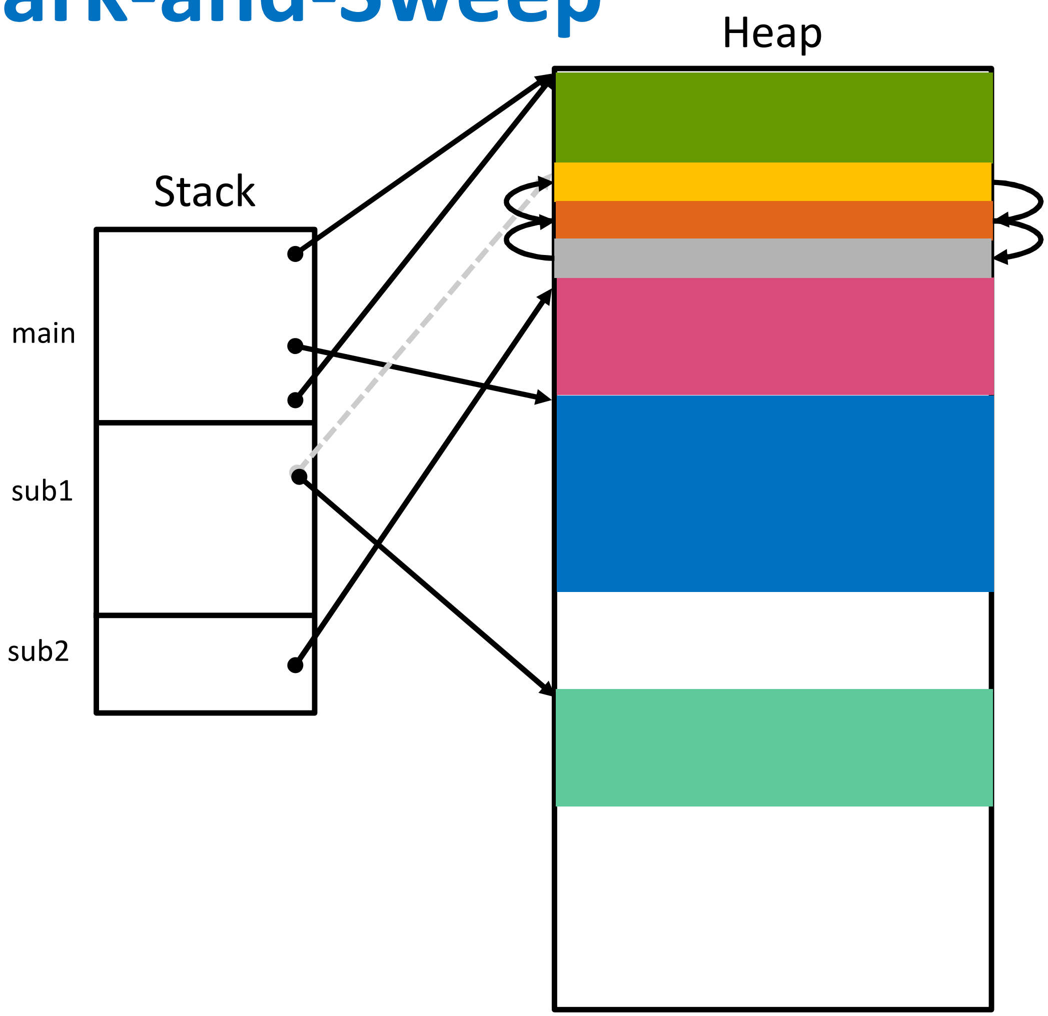
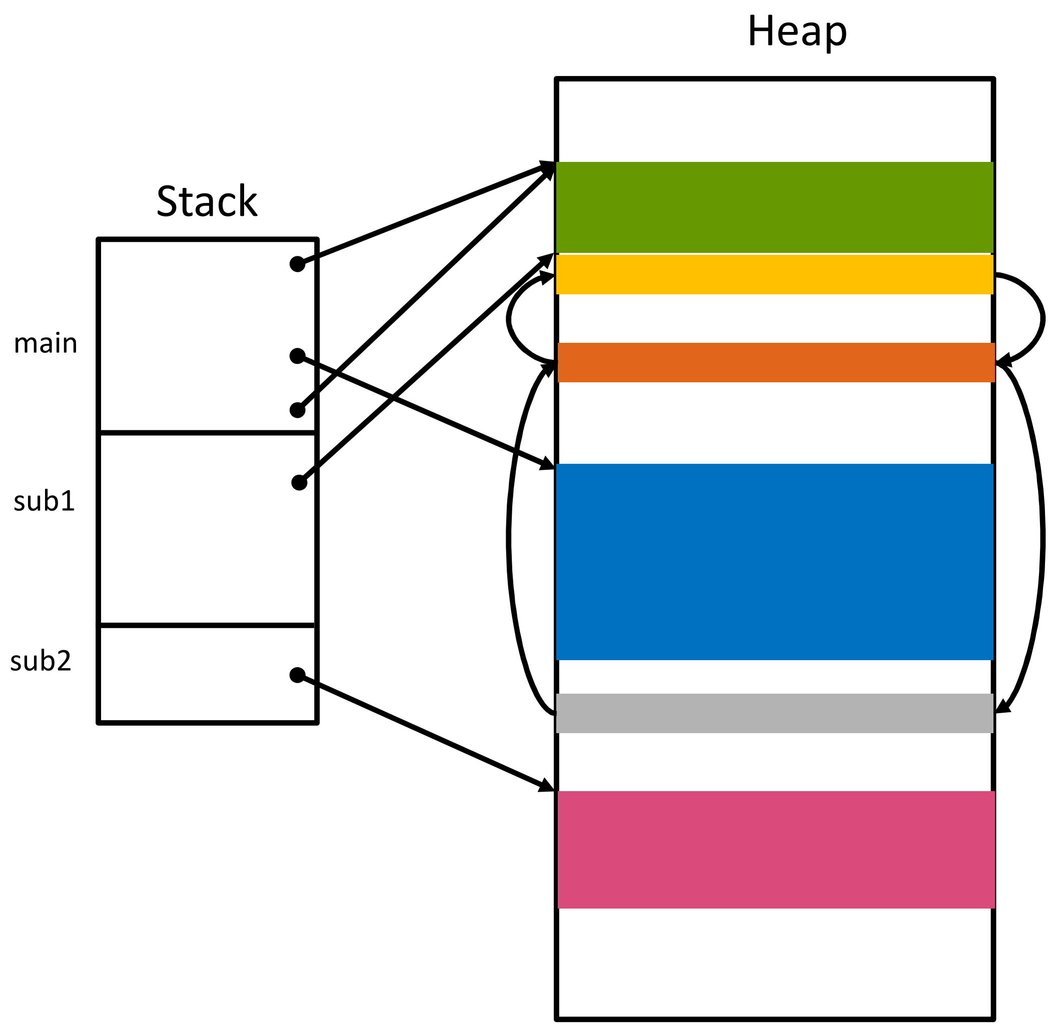
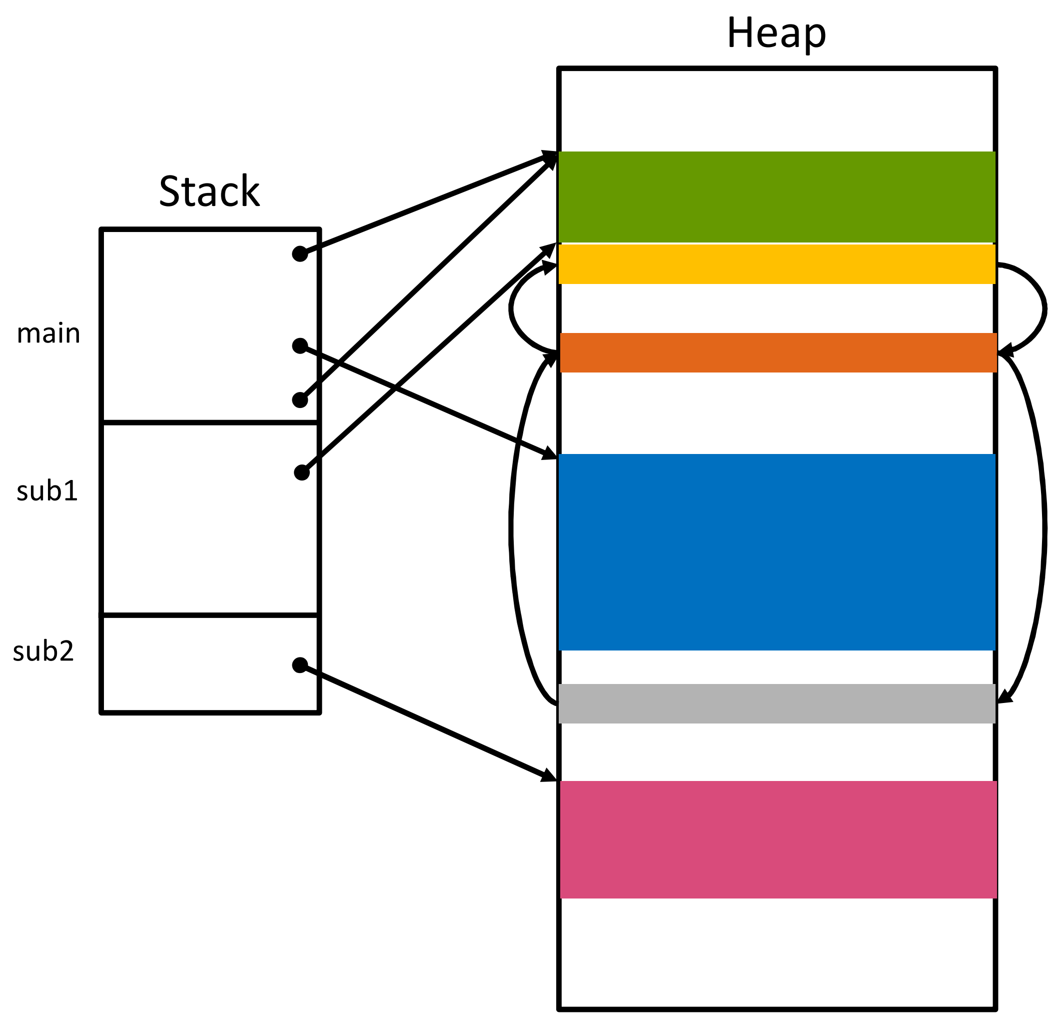
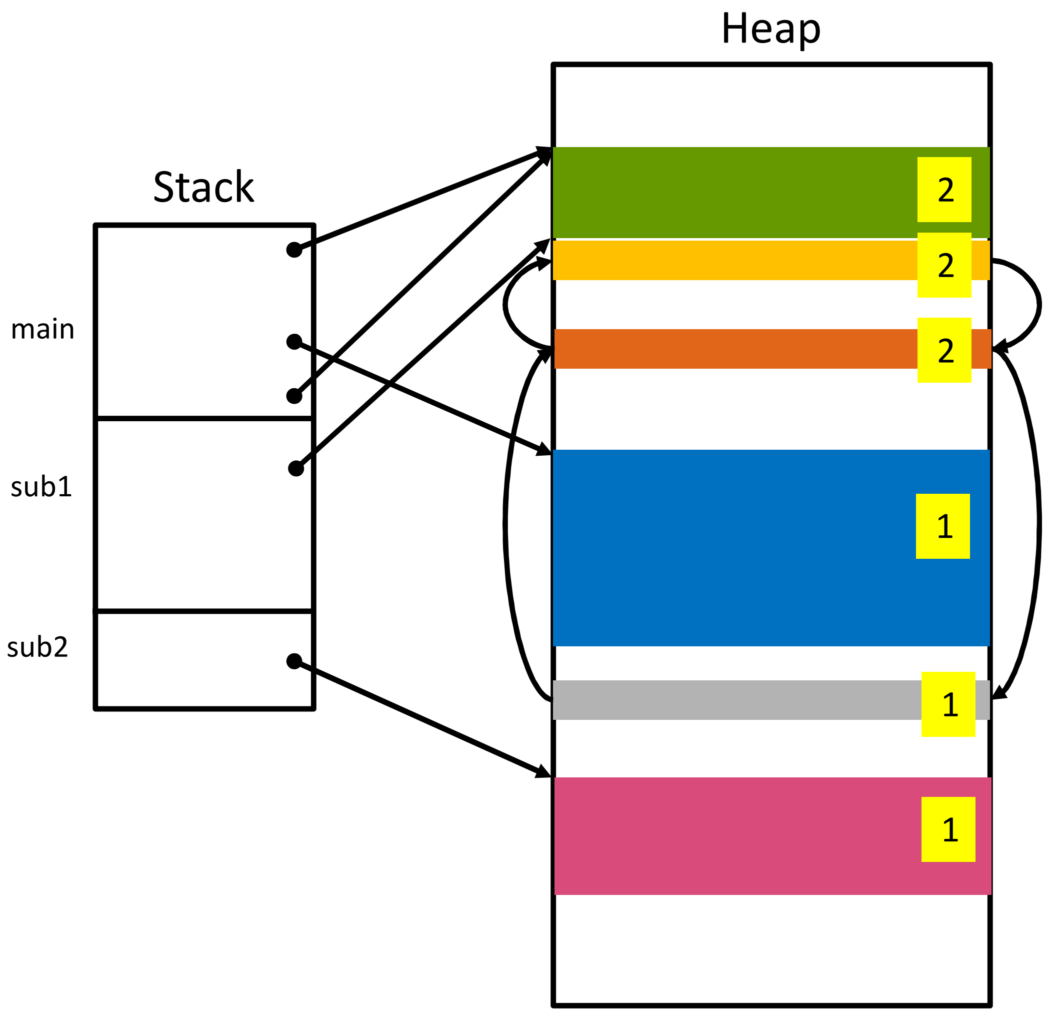
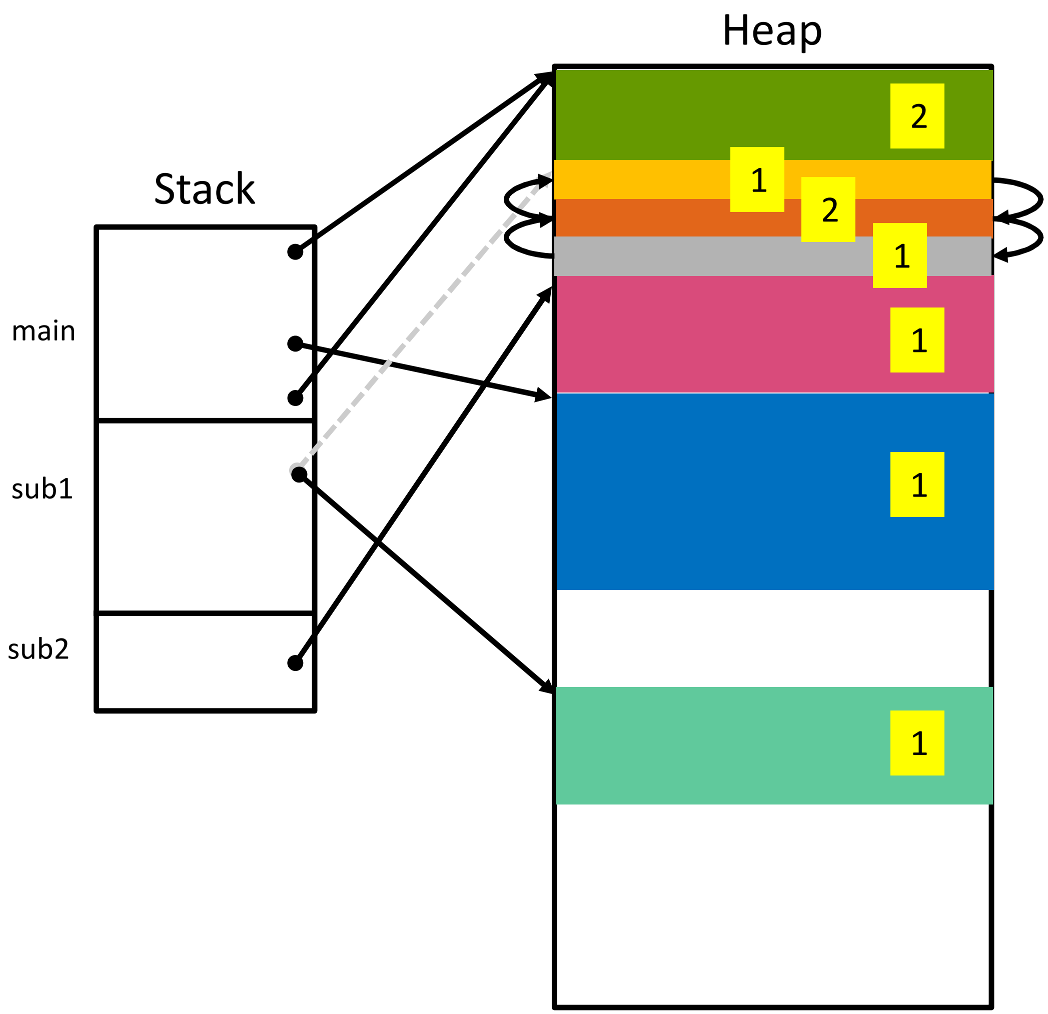
Matthew Arnold	Nonthakit Chaiwong	Jacob Cohen
Elizabeth Haker	Henry Hung	Chase Lee
Leo Liao	Tim Mandzyuk	Benjamin Shmidt
Guramrit Singh		

Lecture Outline

- Overview of Java Garbage Collection
  - Why doesn’t C++ do that?
- An Alternative: Reference counting
- Dynamically Allocated Memory Issues
- *ad hoc* RAII Memory Allocation in C++
- C++ Standard Library Support
  - `std::unique_ptr`
  - `std::shared_ptr`
  - `std::weak_ptr`

Garbage

- Dynamically allocated memory must eventually be deleted, or else you can run out
  - Even before you run out, you can run slower and slower…
- Memory must not be deleted before it becomes "garbage"
  - Garbage is memory that can never be accessed again

- ```
  pMyObj = new Obj("one");
  pMyObj = new Obj("two");
  ```
  The memory allocated in the first statement is garbage after the second, because it cannot be referenced
  ```

Automatic Garbage Collection

- Use of managed memory (e.g., malloc()/free()) is the source of many bugs and a lot of programming pain

- A language with automatic garbage collection relieves the programmer of the burden of coding when free's should take place

- Yeah!

- Let's look at (automatic) garbage collection…

Gabage Collection (GC)

- The goal of garbage collection is to not run out of dynamically allocatable memory (due to garbage)
  - Includes unable to allocate a big enough piece due to fragmentation
- When should garbage be collected?
  1. Immediately, when it turns into garbage?
  2. When you run out of allocatable memory (or just before)?
  3. Every once in a while?
- There's a trade-off among
  - On-going overhead costs
  - Latency (dead time) while GC takes place
  - Getting it right…

Mark-and-Sweep GC

- Java doesn't define what GC method must be used
  - There are many
  - These slides try to present a general sense

- Mark-and-Sweep
  - Mark: find all accessible memory
  - Sweep: move the accessible memory into a contiguous region, leaving behind continguous empty space

Mark-and-Sweep



"Mark" to avoid going around cycles forever.

(Marked traversal has application outside of GC in processing graphs that may have loops.)

The blank portions are not currently in use:
either free or garbage

Mark-and-Sweep



I haven't tried to sweep in a logical order, so don't read anything into the order

Mark-and-Sweep



The objects in a cycle become garbage when the root pointer is over-written.

They'll be collected next time GC is run.

Why Doesn't C++ Do Mark-and-Sweep GC?



C++ Can’t Mark-and-Sweep



- *Contrary to the goal of going faster than humanly possible, plus…*
- *It can’t identify root pointers*
  - *Pointers can masquerade as int’s*
  - *int’s can masquerade as pointers*
  - *No runtime information about “type”*

An Alternative GC: Reference Counting



- *Count the number of pointers to each hunk of memory*
- *Increment count when a new pointer is created*
- *Decrement when a pointer is "lost"*
    - *Assign a new value to the pointer*
- *Free if the count reaches zero*

Reference Counting Failure



Cycles in the graph are a problem!

Reference Counting

- Pro's
  - Lowish overhead
    - You're not moving huge hunks of memory around
  - Garbage collected as soon as it become garbage (sort of)
  - Typically low latency per GC event
    - Okay, could be slow if you're cascading deletion of an enormous linked list, but that's part of the cost of that data structure
      - (i.e., use something else if it bother you)

- Con's
  - Space overhead, possibly (if objects are small)
  - Doesn't always work

C++ and Memory Management

- The original approach is "just get it right" – debug until you do
  - Can be hard to get it right
  - Run valgrind, hope your tests will provoke a leak if one is exists, and then fix it
  - In very dynamic situations, you end up implementing ref counting

- Problems that can arise if you get it wrong
  - Memory leaks
  - Double free's
  - Dangling pointers (multiple pointers to one block of memory and not all are reset when the memory is freed)

- We need help!

RAII Idiom to the Rescue

- RAII – resource acquisition is initialization

```
template <class T>
class Ptr {
  public:
   Ptr() { ptr_ =  new T; }
   ~Ptr() { if ( ptr_ ) delete ptr_; }
   T & operator*() { return *ptr_; }
  private:
    T * ptr_;
};
```

RAII

```
template <class T>
class Ptr {
 public:
  Ptr() { ptr_ =  new T; }
  ~Ptr() { if ( ptr_ ) delete ptr_; }
  T & operator*() { return *ptr_; }
 public:
   T * ptr_;
};
```

vs.

```
void otherSub(int n)
{
  int * pI = new int;
  *pI = n;
  otherSub2(pI);
  delete pI;
}
```

```
void sub(int n)
{
  Ptr<int> pI;
  *pI = n;
  sub2(pI);
}
```

Q: Is there any difference?
A: Yes.

Both should be checking return code from new, but ignore that…

RAII - Yes!

```
void sub(int n)
{
  Ptr<int> pI;
  *pI = n;
  sub2(pI);
}
```

Q: Is there any difference?
A: Yes.

```
void otherSub(int n)
{
  int * pI = new int;
  *pI = n;
  otherSub2(pI);
  delete pI;
}
```

Suppose an exception occurs while executing sub2/otherSub2…

Both should be checking return code from new, but ignore that…

Limitations of Our Crude RAII Attempt

```
template <class T>
class Ptr {
 public:
  Ptr() { ptr_ =  new T; }
  ~Ptr() { if ( ptr_ ) delete ptr_; }
  T & operator*() { return *ptr_; }
 private:
   T * ptr_;
};
```

What’s wrong with this code?

```
void sub(Ptr<int> p) {
}

int main(int argc, char *argv[])
{
  Ptr<int> pInt;
  Ptr<int> pOtherInt;

  *pInt = 4;
  pOtherInt = pInt;

  sub(pInt);

  int * rawPtr = &*pInt;

  return 0;
}
```

Limitations of Our Crude RAII Attempt

```
template <class T>
class Ptr {
 public:
  Ptr() { ptr_ =  new T; }
  ~Ptr() { if ( ptr_ ) delete ptr_; }
  T & operator*() { return *ptr_; }
 private:
   T * ptr_;
};
```

```
void sub(Ptr<int> p) {
}

int main(int argc, char *argv[])
{
 Ptr<int> pInt;
 Ptr<int> pOtherInt;

 *pInt = 4;
 pOtherInt = pInt;      // Memory leak

 sub(pInt);             // Dangling Pointer

 int * rawPtr = &*pInt;

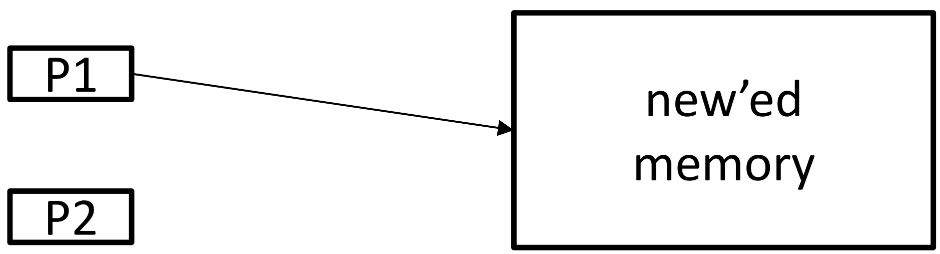
 return 0;
}                       // Double Free
```

Overcoming the Flaws

- Sometimes scope isn't sufficiently flexible to determine lifetime
  - Let's use reference counting of the thing pointed at
- Sometimes scope is exactly the right lifetime…
- Sometimes copying pointers is a problem…
  - Let's override (or maybe disable) copy construction and assignment
- The STL provides implementations that do all this for us!

C++ Smart Pointers – std::unique_ptr<T>

- ```
  std::unique_ptr<int> OnlyPtr(new int(5));     // or...
  auto OnlyPtr = std::make_unique<int>(5);    // since C++14
  std::cout << *OnlyPtr << std::endl;
  ```
  - For the special case is when there should be only a single (unique) pointer to the allocated memory
  - std::unique_ptr<T> deletes copy constructor and (normal) assignment
  - It (of course) deletes the allocated memory on destruction

- There are methods to
  - Cause deletion now (and set unique_ptr to nullptr)
  - Produce the pointer as a regular pointer (!)
  - Other bad ideas (and some good ones)
  ```

First (Special) Case: There Can Be Only One Ptr

I can write correct code if there's only one pointer to each hunk of dynamically allocated memory and it's freed when that pointer is lost

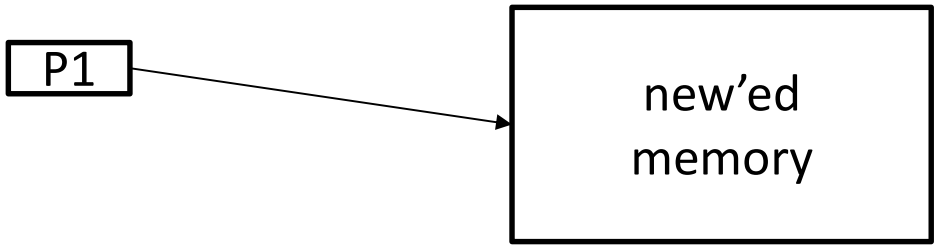
- No leaks
- No double frees
- No dangling pointers

Issue: I have to make sure there's never more than one copy of the pointer

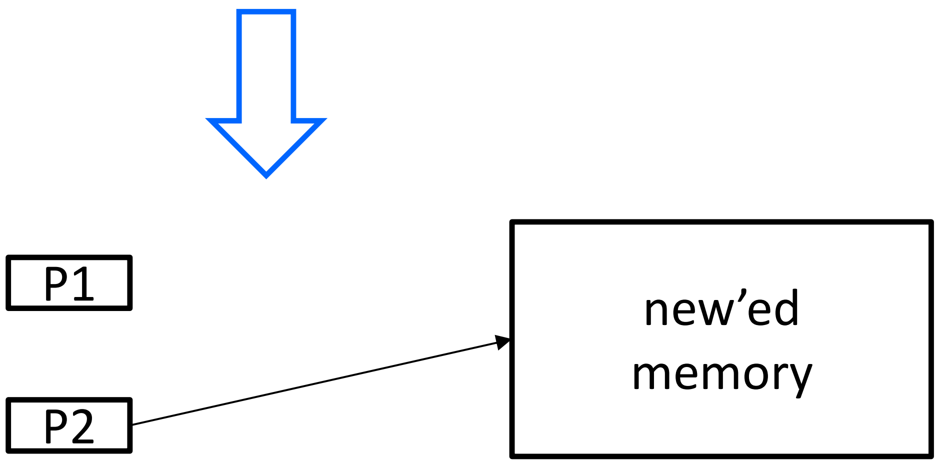


P2 = P1; *// I want this to be a compile time error!*

Special Case: There Can Be Only One Ptr




```
std::unique_ptr<int> P2(P1.release());
```



More General Case – std::shared_ptr<T>

- std::shared_ptr<T> implements reference counting
  - Can have any number of pointers to dynamically allocated memory
  - Copy and assignment operations are overloaded
    - A = B;
      - Increment the reference count of memory B is pointing at, if any
      - Decrement the reference count of memory A was pointing at, if any

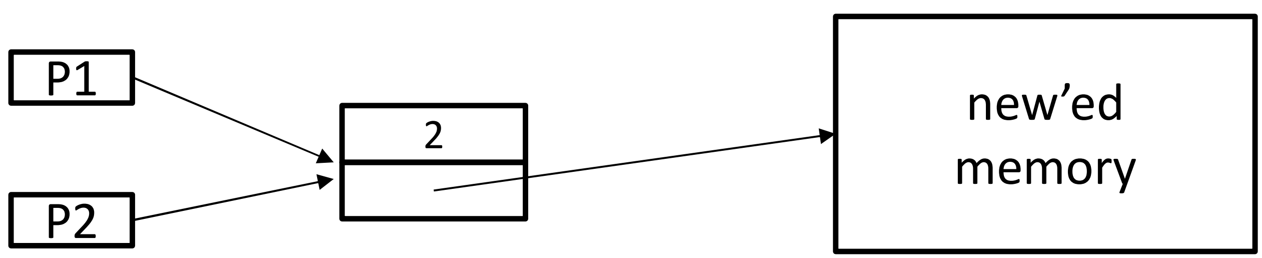
```
std::shared_ptr<int> FirstPtr(new int(5));        // don't do this...
auto SecondPtr = std::make_shared<int>(10);  // since C++11
SecondPtr = FirstPtr;     // memory holding 10 is deleted
FirstPtr = nullptr;        // no delete takes place
```

std::shared_ptr issue…

- If there can be many pointers pointing to the same memory, how can I know when to delete it
  - I can't search for pointers
    - Because it's too expensive and because I can't
  - Keeping a list of pointers associated with the memory would be very expensive
    - Have to update potentially two such lists each time a pointer gets a new value
- Solution: reference counting
  - Don't keep a list of pointers pointing to the memory object, just keep track of how many of them there are

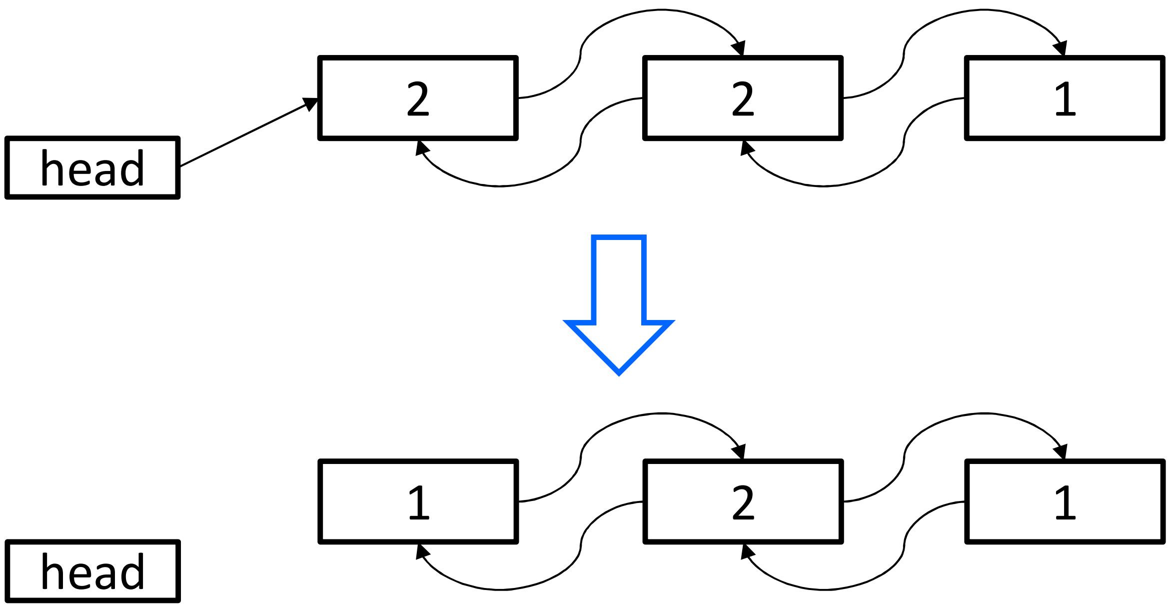
Reference Counting

- std::shared_ptr is a pointer object
- The reference count applies to the thing it points at, not to the pointer
- So, we can't allocate memory for the reference count in the shared_ptr object
  - And we can't allocate it in the object pointed to (in part because there's no universal base class for all C++ objects)



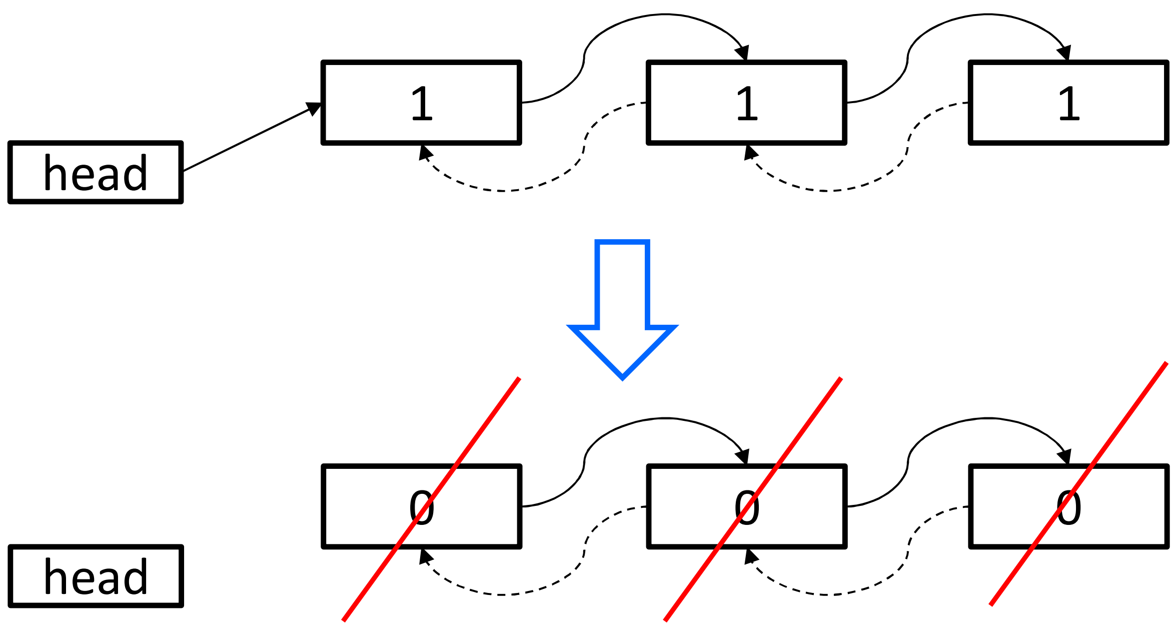
std::weak_ptr

- Reference counting has the problem that isolated cycles are never deleted

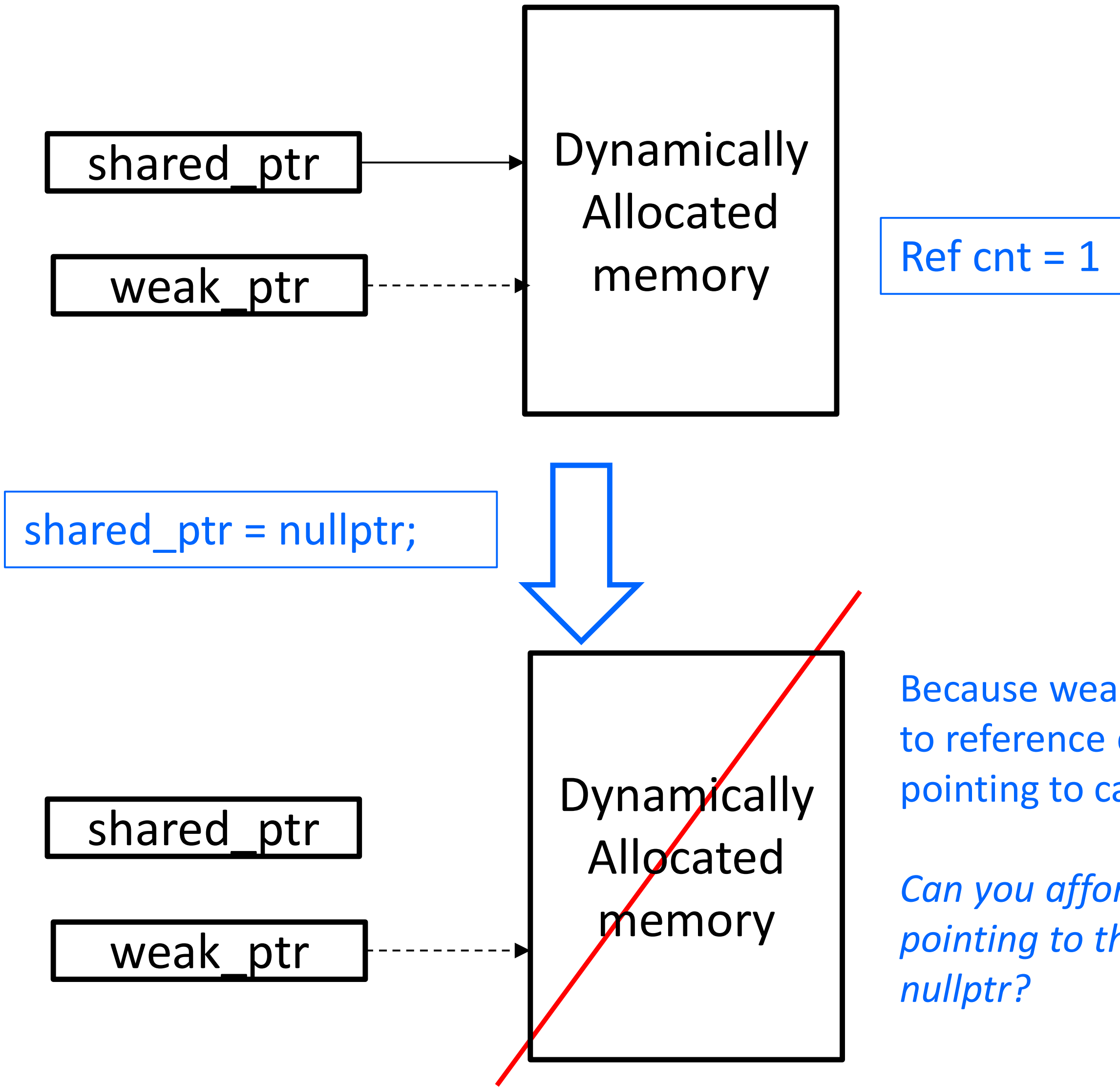


std::weak_ptr

- A weak_ptr is a pointer that doesn't contribute to the reference count



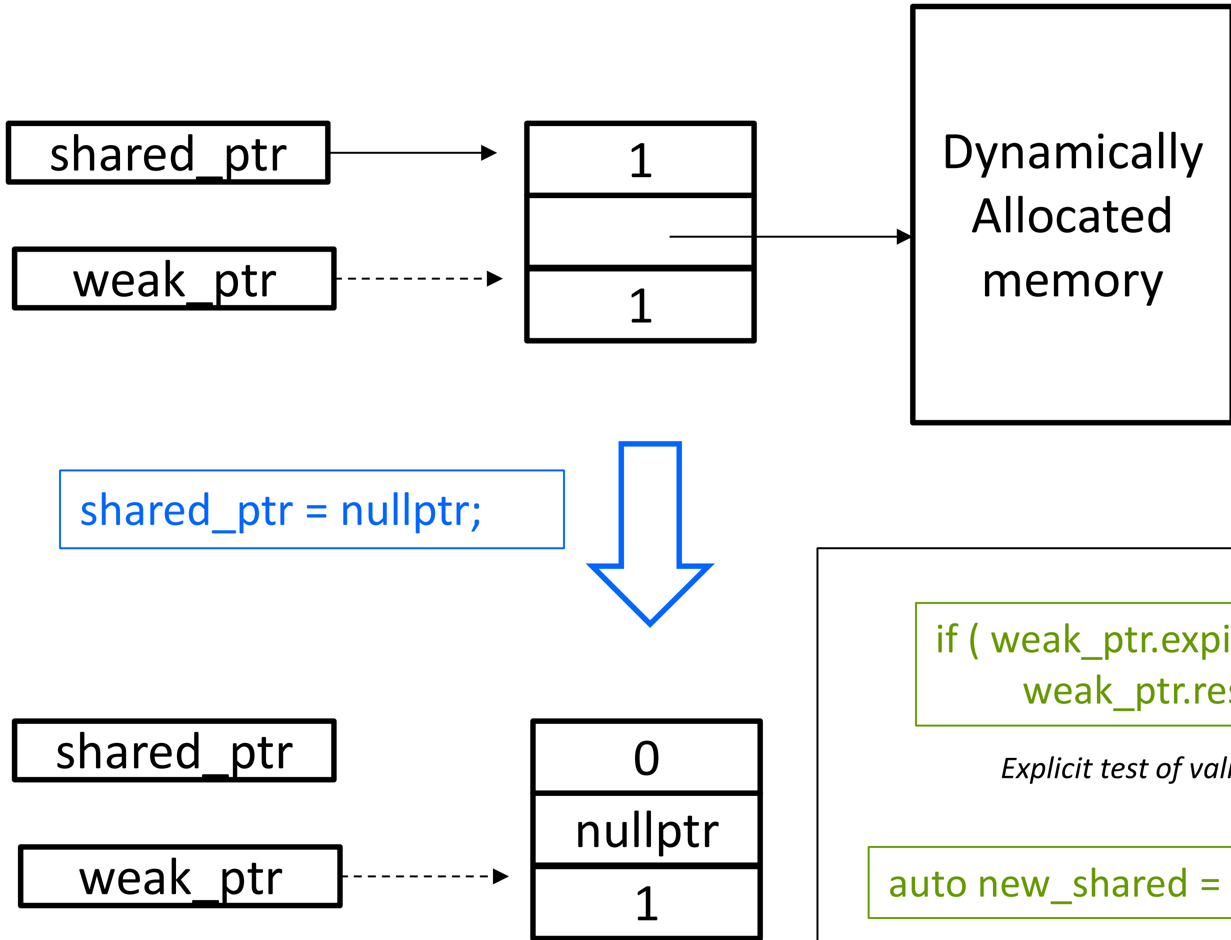
std::weak_ptr Issue?



Because weak_ptr's don't contribute to reference count, what they're pointing to can be deleted.

Can you afford to find all weak_ptr's pointing to the object and set them to nullptr?

weak_ptr Issue?




```
if ( weak_ptr.expired() )
    weak_ptr.reset();
```

Explicit test of validity

```
auto new_shared = weak_ptr.lock();
```

new_shared is a shared_ptr that is either nullptr or not

Summary

- Never use raw (C style) pointers
  - Use smart pointers
- Never use new()
  - Use make_unique<T> and make_shared<T> to construct pointers
- Never use delete
  - Use reset(), when needed

- There can be complications…
  - unique_ptr, in particular, has some unexpected interactions
    - Stand by for the next module

Bonus Slide

- The actual Java garbage collection techniques use multiple regions of memory to perform mark-and-sweep
- The motivation and use is similar to solutions to other memory management problems
  - E.g., managing virtual memory in operating systems
- If you're at all interested, I think you would find it fun to do some reading about it
  - Maybe with a friend