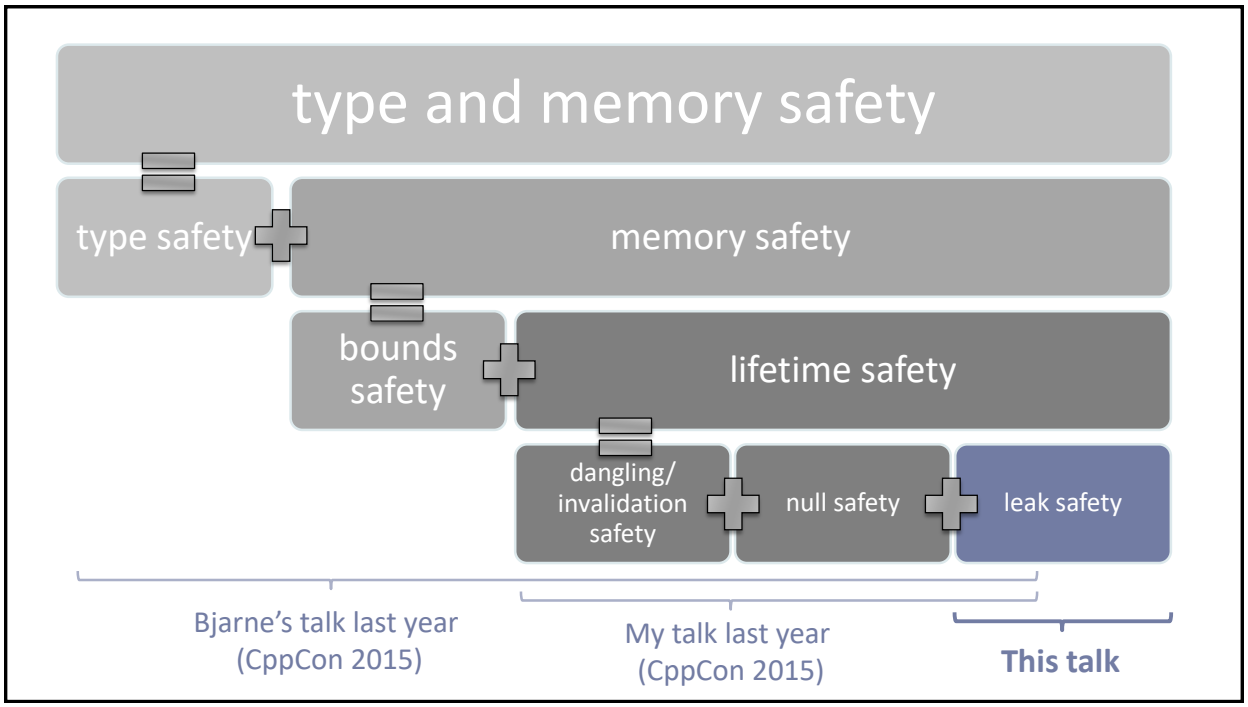


Leak-Freedom in C++… *By Default*

Herb Sutter

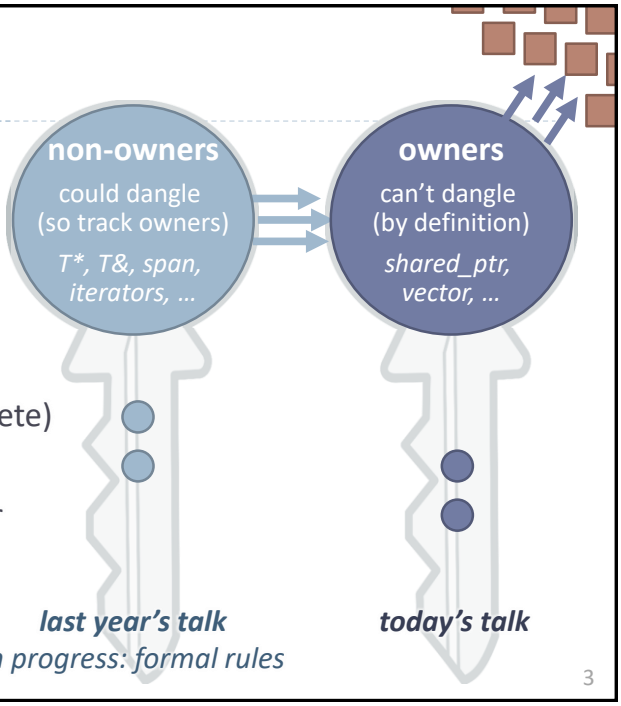
Lifetime guarantees

Lifetime safety targets these classes of bugs:

no dangling/invalid deref (use after delete)

no null deref

no leaks (delete objects when no longer used, and delete only once)



last year's talk
in progress: formal rules

today's talk

3

"Leak freedom in C++" poster

Strategy	Natural examples	Cost	Rough frequency
1. Prefer scoped lifetime by default (locals, members)	Local and member objects – directly owned	Zero: Tied directly to another lifetime	O(80%) of objects
2. Else prefer make_unique & unique_ptr or a container, if the object must have its own lifetime (i.e., heap) and ownership can be unique w/o owning cycles	Implementations of trees, lists	Same as new/delete & malloc/free **Automates** simple heap use in a library	O(20%) of objects
3. Else prefer make_shared & shared_ptr, if the object must have its own lifetime (i.e., heap) and shared ownership w/o owning cycles	Node-based DAGs, incl. trees that share out references	Same as manual reference counting (RC) **Automates** shared object use in a library	

Don't use owning raw *'s == don't use explicit *delete*

Don't create ownership cycles across modules by owning "upward" (violates layering)
Use *weak_ptr* to break cycles

5

Example: HAS-A

- **Q:** What's the natural ownership abstraction for containing another object?
  - Part of my class's representation.
- **A: Nonstatic data member.**
  - Surprise! (not really)

```
class MyClass {
   Data data;
   /*… */
};
```

6

Example: HAS-A

- **Q:** What's the natural ownership abst
  - Part of my class's representation.
- **A: Nonstatic data member.**
  - Surprise! (not really)

Key: Declares lifetime by construction.

Correct: Can see without looking at function bodies that there are no leaks.

Efficient: Zero extra allocation/tracking.

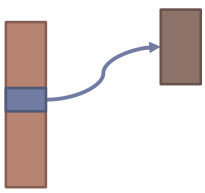
Robust: Strictly nested lifetime. Any bugs are sins of commission, not of omission.


```
class MyClass {
  Data data;
  /*… */
};
```

7

Example: Decoupled HAS-A

- **Q:** What's the natural ownership abstraction for a decoupled member?
  - Optional part (e.g., lazily initialized on demand).
  - Changeable part (e.g., could change derived type).
- **A: unique_ptr.**
  - Note: If it shouldn't be null, remember to write your own move operations appropriately.



```
class MyClass {
  unique_ptr<Data> pdata;
  /*… */
};
```

8

Example: Decoupled

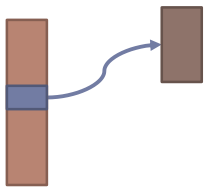
- **Q:** What's the natural ownership abst
  - Optional part (e.g., lazily initialized on c
  - Changeable part (e.g., could change de
- **A: unique_ptr.**
  - Note: If it shouldn't be null, remember to write your own move operations appropriately.

Key: Declares lifetime by construction.

Correct: Can see without looking at function bodies that there are no leaks.

Efficient: Equal space & time to correctly written manual new/delete by hand.

Robust: Strictly nested lifetime.
(But remember to prevent null-after-move.)

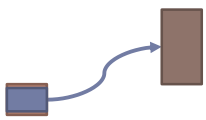


```
class MyClass {
   unique_ptr<Data> pdata;
   /*… */
};
```

9

Example: Pimpl Idiom (compilation firewall)

- **Q:** What's the natural ownership abstraction for a compilation firewall (Pimpl)?
- **A: const unique_ptr.**
  - Non-const can work too, but is more brittle because default move semantics are probably incorrect.
  - "Another/detached part of 'this' object."



```
template<class T>
using Pimpl = const unique_ptr<T>;

class MyClass {
   class Impl;        // defined in .cpp
   Pimpl<Impl> pimpl;
   /*… Note: declare destructor and write it elsewhere …*/
};
```

10

Example: Pimpl Idiom (

- **Q:** What's the natural ownership abstract
- **A: const unique_ptr.**
  - Non-const can work too, but is more brittle
    default move semantics are probably incorre
  - "Another/detached part of 'this' object."

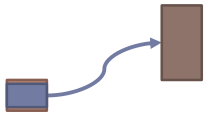
Key: Declares lifetime by construction.

Correct: Can see without looking at function bodies that there are no leaks.

Efficient: Equal space & time to correctly written manual new/delete by hand.

Robust: Strictly nested lifetime. Any bugs are sins of commission, not of omission.

```
template<class T>
using Pimpl = const unique_ptr<T>;

class MyClass {
   class Impl;        // defined in .cpp
   Pimpl<Impl> pimpl;
   /*… Note: declare destructor and write it elsewhere …*/
};
```

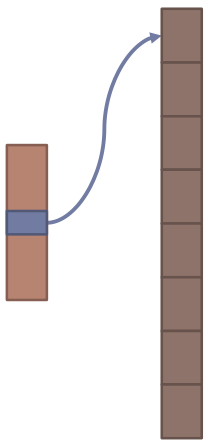


11

Example: Dynamic array member

- **Q:** How can we express a fixed-but-dynamic-size member array?
- **A: const unique_ptr<[]>.**
  - "Another/detached part of 'this' object."
  - (Can remove *const* if you want whole-array move and handling for nullness.)

```
class MyClass {
   const unique_ptr<Data[]> array;
   int array_size;    // or: Data* end;
   /*…*/
   MyClass(size_t num_data)
      : array(make_unique<Data[]>(num_data))
   { }
};
```

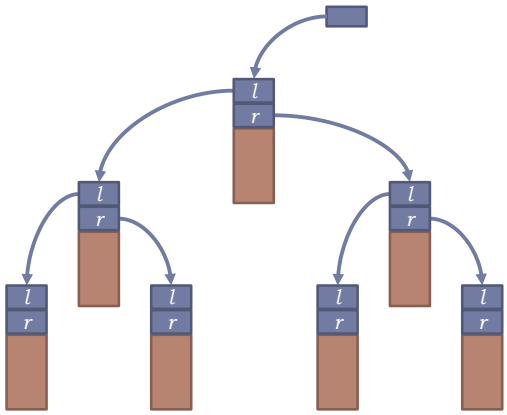


12

Example: Tree (usually encapsulated)

- **Q:** What's the natural ownership abstraction for a tree?
- **A: unique_ptr.**

```
class Tree {
  struct Node {
    vector<unique_ptr<Node>> children;

    /*… data …*/
  };
  unique_ptr<Node> root;
  /*…*/
};
```



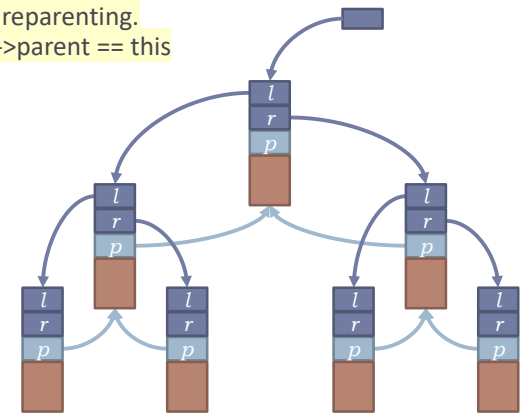
13

Example: Tree (usually encapsulated)

- **Q:** What's the natural ownership abstraction for a tree?
- **A: unique_ptr.**

  Only manual part: Update parent pointer when reparenting.
  Enforce invariant: left->parent == this && right->parent == this

```
class Tree {
  struct Node {
    vector<unique_ptr<Node>> children;
    Node* parent;
    /*… data …*/
  };
  unique_ptr<Node> root;
  /*…*/
};
```



14

Example: Tree (usual

- **Q:** What's the natural ownership abstract
- **A: unique_ptr.**
  - Only manual part: Update parent pointer wh
    Enforce invariant: left->parent == this && ri

```
class Tree {
  struct Node {
    vector<unique_ptr<Node>> children;
    Node* parent;
    /*… data …*/
  };
  unique_ptr<Node> root;
  /*…*/
};
```

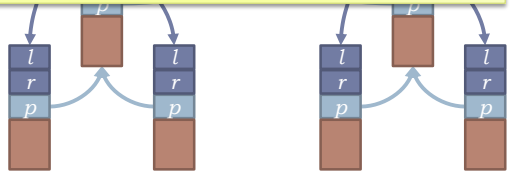
Key: Declares lifetime by construction.

Correct: Can see without looking at function bodies that there are no leaks.

Efficient: Equal space & time to correctly written manual new/delete by hand.

Robust: Any bugs are sins of commission, not of omission. (Except for parent invariant.)

Exercise for the reader: Write the *reparent()* function that encapsulates updating the invariant. (Not hard, but more than two lines.)



15

Happiness is…

… not overflowing

Releasing subtrees: Recursive vs. iterative

Recursive (default)

- Destructors are recursive by default. Example:

```
void release_subtree(unique_ptr<Node> n)
{
}  // destroy n here
   // calls n->~Node()
   // → n->children[0]->~Node();
   // → n->children[0]->children[0]->~Node();
   // → n->children[0]->children[0]->children[0]->~Node();
   // → n->children[0]->children[0]->children[0]->children[0]->~Node();
   // → n->children[0]->children[0]->children[0]->children[0]->children[0]->~Node();
   // → n->children[0]->children[0]->children[0]->children[0]->children[0]->children[0]->ch
   // → n->children[0]->children[0]->children[0]->children[0]->children[0]->children[0]->ch
   // → n->children[0]->children[0]->children[0]->children[0]->children[0]->children[0]->ch
   // → n->children[0]->children[0]->children[0]->children[0]->children[0]->children[0]->ch
   // → n->children[0]->children[0]->children[0]->children[0]->children[0]->children[0]->ch
   // → n->children[0]->children[0]->children[0]->children[0]->children[0]->children[0]->children[0]->ch
```

18

Releasing subtrees: Recursive vs. iterative

Recursive (default)

- Destructors are recursive by default. Example:

```
void release_subtree(unique_ptr<Node> n)
{
}  // destroy n here
   // calls n->~Node()
   // → n->children[0]->~Node();
   // → n->children[0]->children[0]->~Node();
   // → etc.
```

- Pro: Automatic and correct.
- Con: Unbounded stack depth.

Iterative (manual options)

- Iterate by hand. Here's one example:

```
void release_subtree(unique_ptr<Node> n)
{
   while (n->children.size() > 0) {
      auto leaf = &n->children;
      while (leaf[0]->children.size() > 0)
         leaf = &leaf[0]->children;
      leaf.pop_front(); // found leaf, drop it
   }
} // destroy *n itself here
```

- Con: Manual optimization.
- Pro: Bounded stack depth.
  - Note: This is O(N logN). There is an O(N) method but it doesn't fit on the slide.

19

Releasing N-node subtree: Extra cost (on top of O(N) dtors)

	+ Time	+ Stack space	+ Heap space
Default: prompt, recursive	—	O(N)	—
Iterative, in-place: prompt, navigate subtree and run dtors bottom-up (previous slide)	+ O(N logN)	—	—
Iterative, copy: Move ptrs to the nodes to be pruned to a flattened local-scope heap list, then run dtors iteratively	+ O(N)	—	O(N)
Iterative, deferred: Move ptrs to the nodes to be pruned to a flattened side list, run destructors later iteratively	+ O(N) *– dtors*	—	O(N)

Additional overhead, on top of running all the O(N) Node destructors themselves

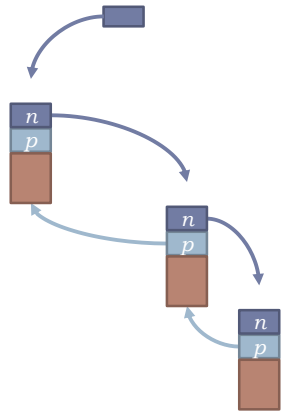
"—" means minimal, O(K) with low K

20

Example: Doubly linked list

- **Q:** What's the natural ownership abstraction for a doubly linked list?
- **A: unique_ptr.**
  - Only manual part: Update previous pointer when relinking. Enforce invariant: next->prev == this

```
class LinkedList {
  struct Node {
    unique_ptr<Node> next;

    Node* prev;
    /*… data …*/
  };
  unique_ptr<Node> root;
  /*…*/
};
```

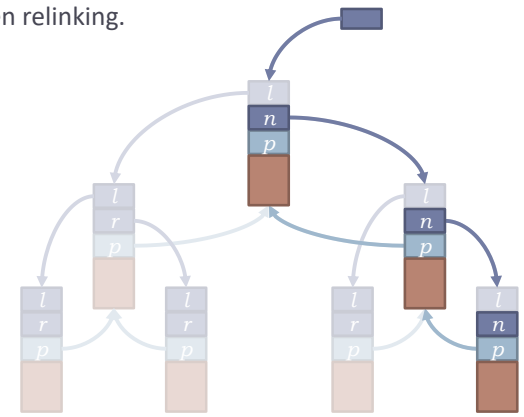


21

Example: Doubly linked list

- **Q:** What's the natural ownership abstraction for a doubly linked list?
- **A: unique_ptr.**
  - Only manual part: Update previous pointer when relinking. Enforce invariant: next->prev == this

```
class LinkedList {
  struct Node {
    unique_ptr<Node> next;

    Node* prev;
    /*… data …*/
  };
  unique_ptr<Node> root;
  /*…*/
};
```



22

Example: Doubly link

- **Q:** What's the natural ownership abstract
- **A: unique_ptr.**
  - Only manual part: Update previous pointer
    Enforce invariant: next->prev == this

```
class LinkedList {
  struct Node {
    unique_ptr<Node> next;

    Node* prev;
    /*… data …*/
  };
  unique_ptr<Node> root;
  /*…*/
};
```

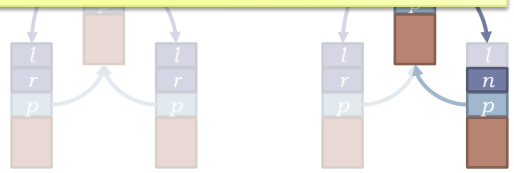
Key: Declares lifetime by construction.

Correct: Can see without looking at function bodies that there are no leaks.

Efficient: Equal space & time to correctly written manual new/delete by hand.

Robust: Any bugs are sins of commission, not of omission. (Except for parent invariant.)

But, especially for the list case, be careful about destructor recursion. Typically, you want to prune iteratively.

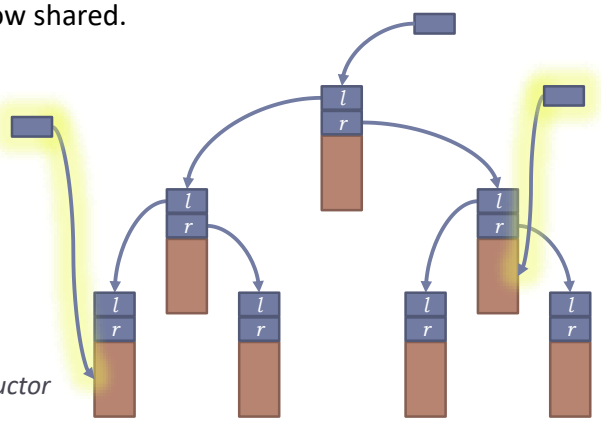


23

Example: Tree that hands out strong refs

- **Q:** What's the natural ownership abstraction for a tree that hands out strong references to data?
- **A: shared_ptr.** After all, ownership is now shared.

```
class Tree {
  struct Node {
    vector<shared_ptr<Node>> children;
    Data data;
  };
  shared_ptr<Node> root;
  shared_ptr<Data> find(/*…*/) {
    /*…*/ return {spn, &(spn->data)};
  }  // note: uses shared_ptr aliasing constructor
};
```

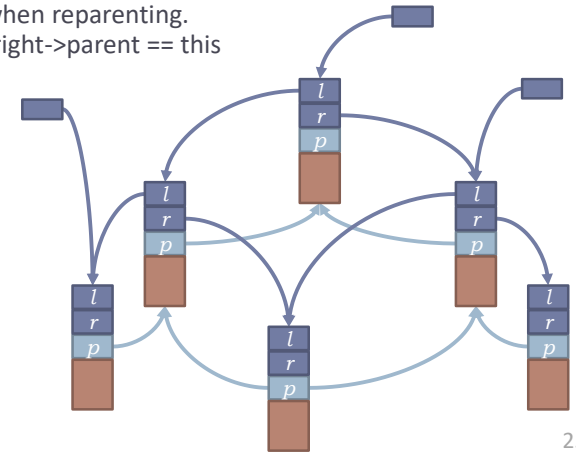


24

Example: DAG of heap objects

- **Q:** What's the natural ownership for a node-based container that's a DAG?
- **A: shared_ptr.**
  - Only manual part: Update parent pointer when reparenting.
    Enforce invariant: left->parent == this && right->parent == this

```
class DAG {
  struct Node {
    vector<shared_ptr<Node>> children;
    vector<Node*> parents;
    /*… data …*/
  };
  vector<shared_ptr<Node>> roots;
  /*…*/
};
```



25

Example: DAG of hea

- **Q:** What's the natural ownership for a no
- **A: shared_ptr.**
  - Only manual part: Update parent pointer wh
    Enforce invariant: left->parent == this && ri

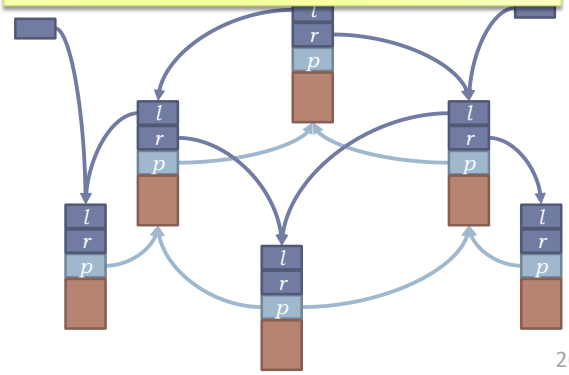
Key: Declares lifetime by construction.

Correct: Can see without looking at function bodies that there are no leaks.

Efficient: Equal space & time to correctly written manual new/delete + RC by hand.

Robust: Any bugs are sins of commission, not of omission. (Except for parent invariant.)

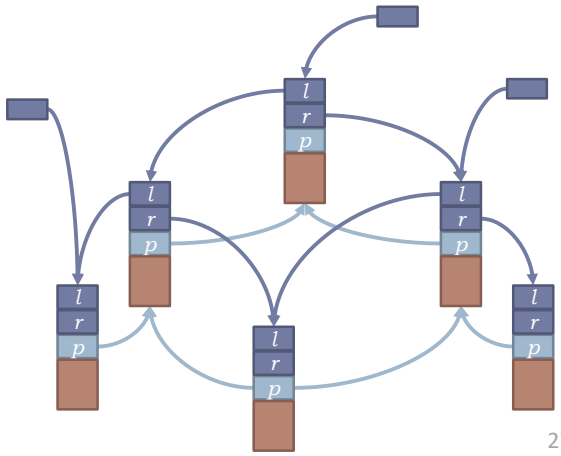
```
class DAG {
  struct Node {
    vector<shared_ptr<Node>> children;
    vector<Node*> parents;
    /*… data …*/
  };
  vector<shared_ptr<Node>> roots;
  /*…*/
};
```



26

Example: DAG of heap objs, unencapsulated

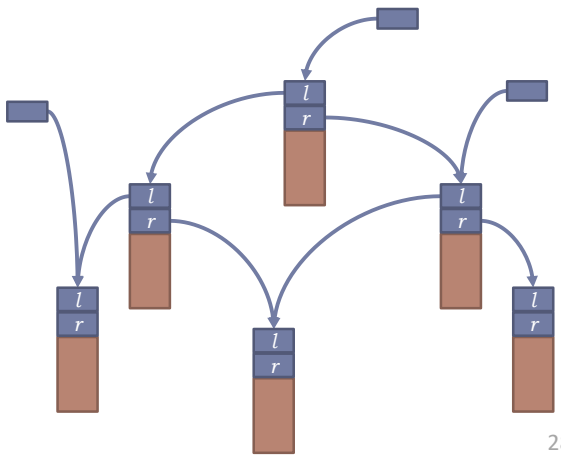
- **Q:** What's the natural ownership for objects that refer to each other w/o cycles?
- **A: shared_ptr.**
  - Logically the same as the previous example.
  - Objects can be of different types.
  - Objects can come from different modules.
  - Just don't violate layering…



27

Example: DAG of heap objs, unencapsulated

- **Q:** What's the natural ownership for objects that refer to each other w/o cycles?
- **A: shared_ptr.**
  - Logically the same as the previous example.
  - Objects can be of different types.
  - Objects can come from different modules.
    Might have parent pointers, or not.
  - Just don't violate layering…

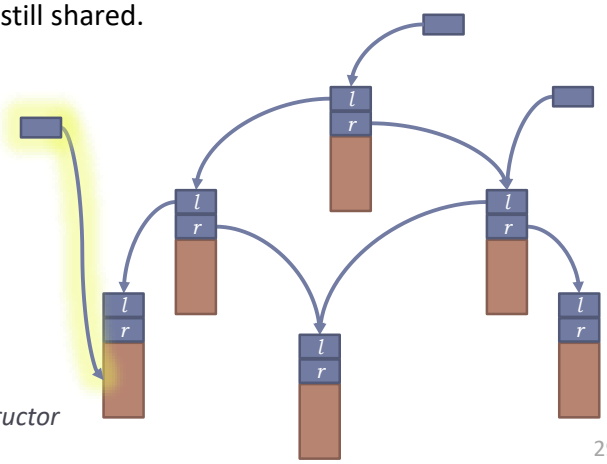


28

Example: DAG that hands out strong refs

- **Q:** What's the natural ownership abstraction for a DAG that hands out strong references to data?
- **A: shared_ptr.** As before, ownership is still shared.

```
class DAG {
 struct Node {
      vector<shared_ptr<Node>> children;
      vector<Node*> parents;
      /*… data …*/
   };
   vector<shared_ptr<Node>> roots;
   shared_ptr<Data> find(/*…*/) {
     /*…*/ return {spn, &(spn->data)};
   }  // note: uses shared_ptr aliasing constructor
};
```



29

Example: Factory

- **Q:** What's the natural ownership when a factory returns a heap object?
- **A: unique_ptr, or shared_ptr + make_shared.**
  - *unique_ptr* if the object might not be shared by the code that then uses it.
  - *shared_ptr* + *make_shared* if the object will be shared by the code that then uses it.

```
???_ptr<widget> make_widget( int id );
```

```
unique_ptr<widget> make_widget( int id ) {
   return make_unique<widget>(id);
}
```

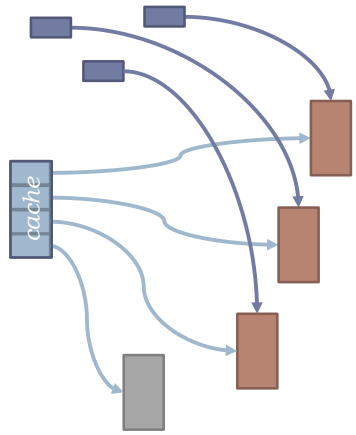
```
shared_ptr<widget> make_widget( int id ) {
   return make_shared<widget>(id);
}
```

30

Example: Factory + cache

- **Q:** What's the natural ownership when a cache of heap objects doesn't keep objects alive (objects should live only as long as there are cache-external users)?
- **A: shared_ptr + weak_ptr.**

```
shared_ptr<widget> make_widget( int id ) {
  static map<int, weak_ptr<widget>> cache;
  static mutex mut_cache;
  lock_guard<mutex> hold( mut_cache );
  auto sp = cache[id].lock();
  if( !sp ) cache[id] = sp = load_widget( id );
  return sp;
}
```



31

"Leak freedom in C++" poster

Strategy	Natural examples	Cost	Rough frequency
1. Prefer scoped lifetime by defa (locals, members)	Local and member objects – directly owned	Zero: Tied directly to another lifetime	O(80%) of objects
2. Else prefer make_unique & unique_ptr or a container, if the ob must have its own lifetime (i.e., heap) ownership can be unique w/o owning cycles	Implementations of trees, lists	Same as new/delete & malloc/free **Automates** simple heap use in a library	O(20%) of objects
3. Else prefer make_shared & shared_ptr, if the object must have own lifetime (i.e., heap) and shared ownership w/o owning cycles	Node-based DAGs, incl. trees that share out references	Same as manual reference counting (RC) **Automates** shared object use in a library	

Don't use owning raw *'s == don't use explicit *delete*

Don't create ownership cycles across modules by owning "upward" (violates layering)
Use *weak_ptr* to break cycles

32

"Leak freedom in C++" poster

Strategy	Natural examples	Cost	Rough frequency
1. Prefer scoped lifetime by defa (locals, members)	Local and member objects – directly owned	Zero: Tied directly to another lifetime	O(80%) of objects
2. Else prefer make_unique & unique_ptr or a container, if the ob must have its own lifetime (i.e., heap) ownership can be unique w/o owning cycles	Implementations of trees, lists	Same as new/delete & malloc/free **Automates** simple heap use in a library	O(20%) of objects
3. Else prefer make_shared & shared_ptr, if the object must have own lifetime (i.e., heap) and shared ownership w/o owning cycles	Node-based DAGs, incl. trees that share out references	Same as manual reference counting (RC) **Automates** shared object use in a library	

Don't use owning raw *'s == don't use explicit *delete*

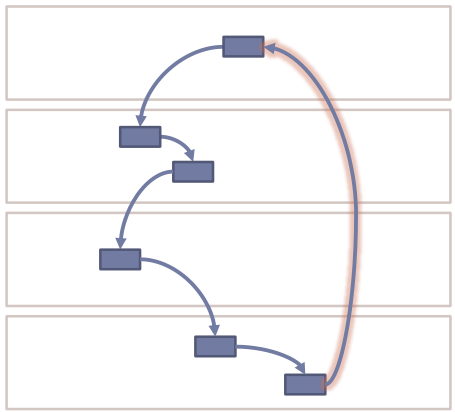
Don't create ownership cycles across modules by owning "upward" (violates layering)
Use *weak_ptr* to break cycles

33

Example: Cross-module cycle (prevent)

- **Q:** How can we avoid creating a cycle across modules?
- **A:** Don't own "upwards" (which violates layering).
  - How: Don't pass an owner down to "unknown code" that might store it.
  - Simple example: Storing a *shared_ptr*.

```
void bad(const shared_ptr<X>& x) {
   obj.register(x);
};
```

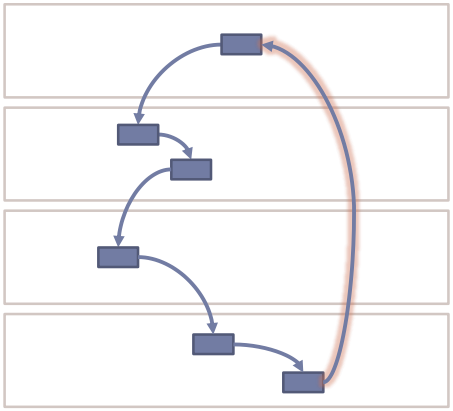


34

Example: Cross-module cycle (prevent)

- **Q:** How can we avoid creating a cycle across modules?
- **A:** Don't own "upwards" (which violates layering).
  - How: Don't pass an owner down to "unknown code" that might store it.
  - Most commonly: Don't store an owner in a library **callback**.
    - NB: Direct parallel to "don't call unknown code while holding a lock" for concurrency.

```
void bad(const shared_ptr<X>& x) {
   obj.on_draw([=]{ x->extra_work(); });
};
```



Example: Cross-module cycle (prevent)

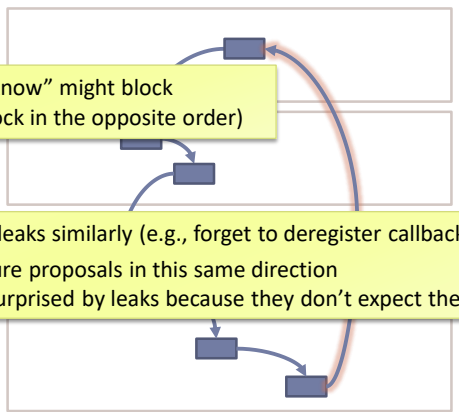
- **Q:** How can we avoid creating a cycle across modules?
- **A:** Don't ov…).
  - How: Don't pass an owner down to "unknown code" that might store it.
  - Most comm… owner in a …
    - NB: Direct parallel to "don't call unknown code while holding a lock" for conc…

"unknown" == unknown relevant behavior
here, code that we "don't know" might store it…

… and here, code that we "don't know" might block
(in a way where another thread could block in the opposite order)

not "just a C++ problem" – Java/C# leaks similarly (e.g., forget to deregister callbacks)
⇒ Java/C# have feature proposals in this same direction
(their programmers are even more surprised by leaks because they don't expect them)

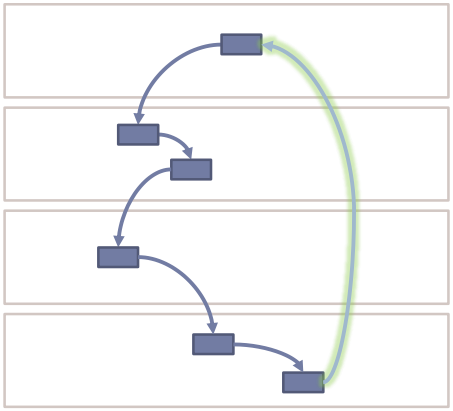
```
void bad(const sha…
   obj.on_draw([=]{ x->extra_work(); });
};
```



Example: Cross-module cycle (prevent)

- **Q:** How can we avoid creating a cycle across modules?
- **A:** Don't own "upwards" (which violates layering).
  - How: Don't pass an owner down to "unknown code" that might store it.
  - Most commonly: Don't store an owner in a **callback**.
    - NB: Direct parallel to "don't call unknown code while holding a lock" for concurrency.

```
void good(const shared_ptr<X>& x) {
   obj.on_draw([w = weak_ptr<X>(x)]{
      if (auto x = w.lock()) x->extra_work(); });
};
```

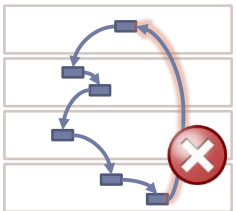


37

Two categories of cycles

Inter-module (compositional)

- Example: Composing two libs, each of which exposes access to heap objects.
  - Cycle is typically **heterogeneous**.
- **Answer: "Don't do that"** (just covered).
  - Don't violate layering by owning upward.
  - Don't store an owner in a callback.

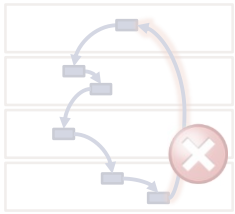


38

Two categories of cycles

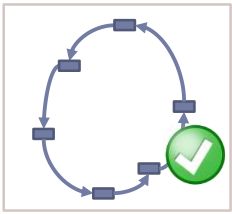
Inter-module (compositional)

- Example: Composing two libs, each of which exposes access to heap objects.
  - Cycle is typically **heterogeneous**.
- **Answer: "Don't do that"** (just covered).
  - Don't violate layering by owning upward.
  - Don't store an owner in a callback.



Intra-module (local)

- Example: Graph.
  - Cycle can be made **homogeneous**.
- Legitimate and reasonable.
  - Not fully automated by *std::* facilities.
  - What should we do *today*?
    Could we maybe do better *tomorrow*?

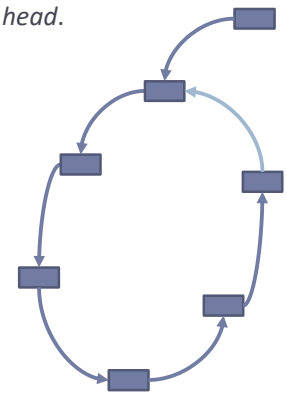


39

Example: Circular list

- **Q:** What's the natural ownership abstraction for a circular list?
- **A: Mostly unique_ptr** (in today's portable C++).
  - Only manual part: *next()* to connect last/sentinel node to *head*.
  - Note: Easier because this is a **static cycle**.

```
class CircularList {
   class Node {
      unique_ptr<Node> next;  // private
      unique_ptr<Node>& head;
   public:
      auto get_next() { return next ? next.get() : head.get(); }
      /*… ctors and data …*/
   };
   unique_ptr<Node> head;
   /*…*/
};
```



40

Example: Circular lis

- **Q:** What's the natural ownership abstract
- **A: Mostly unique_ptr** (in today's portable
  - Only manual part: *next()* to connect last/sen
  - Note: Easier because this is a **static cycle**.

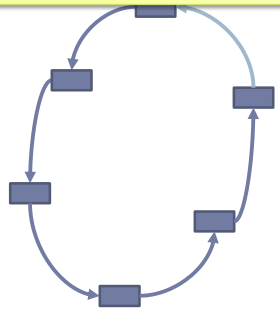
Key: Declares lifetime by construction.

Correct: Can see without looking at function bodies (except *next()*) that there are no leaks.

Efficient: Equal space & time to correctly written manual new/delete by hand.

Robust: Any bugs are sins of commission, not of omission. (Except for *next()* invariant.)

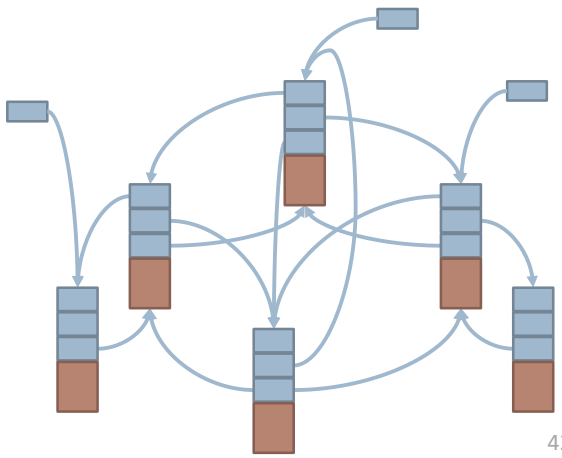
```
class CircularList {
   class Node {
      unique_ptr<Node> next;  // private
      unique_ptr<Node>& head;
   public:
      auto get_next() { return next ? next.get() : head.get(); }
      /*… ctors and data …*/
   };
   unique_ptr<Node> head;
   /*…*/
};
```



41

Example: Graph of heap objs, unencapsulated

- **Q:** What's the natural ownership abstraction for a **possibly-cyclic** graph of objects?

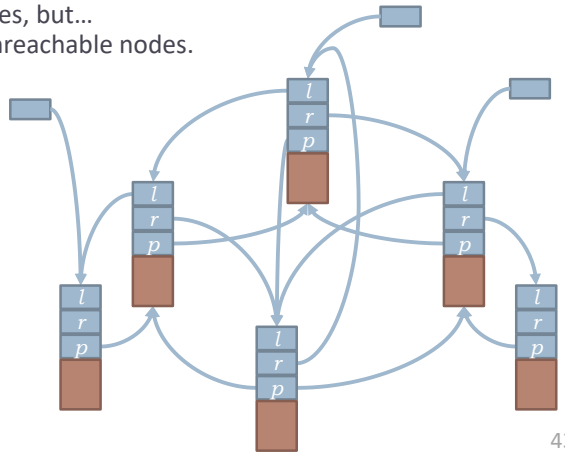


42

Example: Graph of nodes, encapsulated

- **Q:** What's the natural ownership abstraction for a **possibly-cyclic** graph of nodes?
- **A: Partial** (in today's portable C++).
  - Cleanest solution is *vector<unique_ptr>* nodes, but…
    … **manual detection** is needed to identify unreachable nodes.

```
class Graph {
  struct Node {
    vector<Node*> children;
    vector<Node*> parents;
    /*… data …*/
  };
  vector<Node*> roots;
  vector<unique_ptr<Node>> nodes;  // ???
  /*…*/
};
```



43

Example: Graph

- **Q:** What's the natural ownership abstract
- **A: Partial** (in today's portable C++).
  - Cleanest solution is *vector<unique_ptr>* nod
    … **manual detection** is needed to identify ur

```
class Graph {
  struct Node {
    vector<Node*> children;
    vector<Node*> parents;
    /*… data …*/
  };
  vector<Node*> roots;
  vector<unique_ptr<Node>> nodes;  // ???
  /*…*/
};
```

Partly declares ownership by construction.

I know the nodes won't leak when the *Graph* is destroyed.

But I don't know without looking at function bodies that there are no leaks of *Node*s during the lifetime of the *Graph*.

Manually traversing the nodes to discover the unused ones is manual memory management.

Manually calling *nodes.erase(unreachable_node);* is functionally the same as manually calling *delete unreachable_node;*.

Our goal is **automatic** memory management.

earlier this week
at CppCon…

"How many of you know how to find a cycle in a directed graph?"

— Chandler Carruth, Wednesday

in "Garbage In, Garbage Out: Arguing about Undefined Behavior with Nasal Demons"

Paging **Tobias Langner**…

```
set<Node*> GetConnectedNodes() {
    set<Node*> result;
    vector<Node*> stack; //manual stack to allow deeper nesting
    stack.push_back(root); //start from root
    while (!stack.empty()) { //continue until no work to do
```

Paging **Andre Kostur**…

```
set_difference(begin(allnodes), end(allnodes),
               begin(visited), end(visited),
               inserter(unreachable, begin(unreachable)));
```

Paging **Federico Lebron**…

```
void ShrinkToFit() {
    auto wahlberg = [this] {  // get it? because... Mark.
```

Example: Graph

Challenge: Implement *Graph::remove_unused_nodes()*

- **Q:** What's the natural ownership abstraction for a possibly-cyclic graph of nodes?
- **A: Partial** (in today's portable C++).
  - Cleanest solution is *vector<unique_ptr>* nodes, but…
    … **manual detection** is needed to identify unreachable nodes.

```
class Graph {
  struct Node {
    vector<Node*> children;
    vector<Node*> parents;
    /*… data …*/
  };
  vector<Node*> roots;
  vector<unique_ptr<Node>> nodes;  // ???
  /*…*/
};
```

49

Example: Graph

Challenge: Implement *Graph::remove_unused_nodes()*

- **Q:** What's the natural ownership abstract
- **A: Partial** (in today's portable C++).
  - Cleanest solution is *vector<unique_ptr>* nod
    … **manual detection** is needed to identify ur

```
class Graph {
  struct Node {
    vector<Node*> children;
    vector<Node*> parents;
    /*… data …*/
  };
  vector<Node*> roots;
  vector<unique_ptr<Node>> nodes;  // ???
  /*…*/
};
```

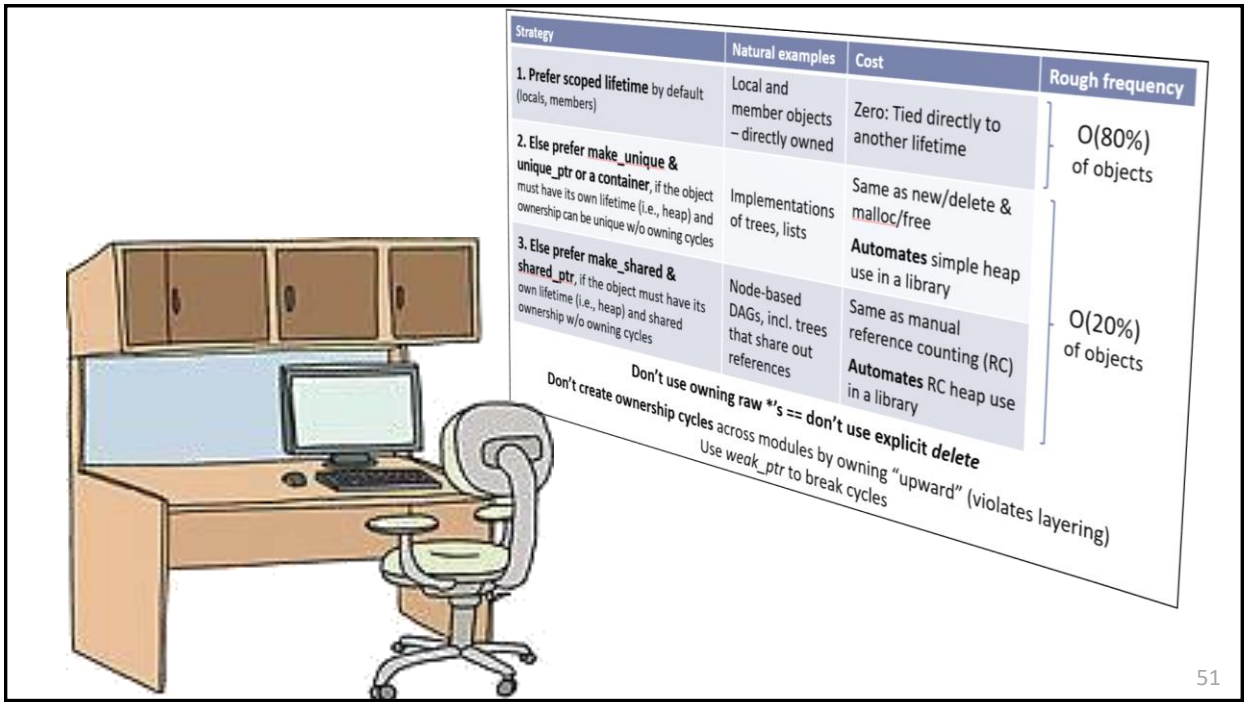
```
void Graph::remove_unused_nodes() {
  vector<const Node*> orphans(nodes.size());
  transform(nodes.begin(), nodes.end(), orphans.begin(),
            [](const auto& x){ return x.get(); });
  sort(orphans.begin(), orphans.end());

  for (const auto& node : all_nodes())  // or longer traversal code
    orphans.erase(lower_bound(
            orphans.begin(), orphans.end(), node.get()));

  for (const auto o : orphans)
    nodes.erase(find_if(nodes.begin(), nodes.end(),
            [o](const auto& x){ return x.get() == o; }));
}
```

Pro: Probably have a traversal function anyway.

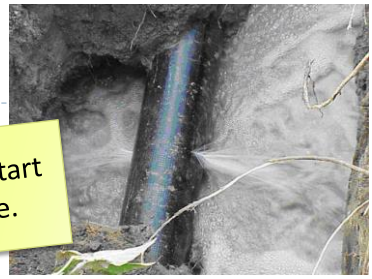
Con: "*nodes.erase*" is just as manual as "*delete*."

Con: Trimming requires full traversal + overhead.

Con: Pattern, needs to be handwritten each time.



*std::*_ptr* limitations



- Ownership cycles ⇒ **leak** objects and memory.
  - *shared_ptr* uses reference counting.
  - Plain reference counting leaks cycles.
  - Can't always be naturally broken using a *weak_ptr*.
- Unbounded pointer copy cost ⇒ exceed **real-time** execution deadlines.
  - *shared_ptr* calls destructors promptly. (Usually a great thing!)
  - But objects can own other objects transitively.
  - Plain assignment to a *shared_ptr* can have arbitrarily high cost. (Similar issue with EH.)
- Unbounded destructor depth ⇒ exceed **stack limits** in constrained environments.
  - Scoped/*unique_ptr*/*shared_ptr* lifetime calls destructors recursively (nested).
  - As above, ownership trees can be deep.
  - Releasing a tree of objects requires having sufficient stack space.

53

Example: Graph

- **Q:** What's the natural ownership abstract
- **A: Partial** (in today's portable C++).
  - Cleanest solution is *vector<unique_ptr>* nod
    … **manual detection** is needed to identify ur

```
class Graph {
  struct Node {
    vector<Node*> children;
    vector<Node*> parents;
    /*… data …*/
  };
  vector<Node*> roots;
  vector<unique_ptr<Node>> nodes;  // ???
  /*…*/
};
```

Challenge: Implement *Graph::remove_unused_nodes()*

Alternatives move space/time overhead around. Example: Add an age count to each node, ++ on each full traversal, then *remove_unused_nodes()* is a 1-pass algorithm to trim nodes for which *age < curr_age*.

But it's **manual** memory management. We'd like not to write a custom sweep by hand in every situation where a cycle could occur.

Q: Can we automate any part of this lifetime as a reusable library, like we automated new/delete with unique)ptr and RC with *shared_ptr*?

Releasing N-node … Extra cost (on top of O(N) dtors)

Recall from earlier:

Remember recursive (default, but can consume unbounded stack)

vs.

iterative (manual, but stack-friendly)?

	+ Time	+ Stack space	+ Heap space
Defau…	—	O(N)	—
Iterati… naviga…	+ O(N logN)	—	—
Iterativ… pruned… then ru…	+ O(N)	—	O(N)
Iterative, deferred: Move ptrs to the nodes to be pruned to a flattened side list, run destructors later iteratively	+ O(N) *– dtors*	—	O(N)

Additional overhead, on top of running all the O(N) Node destructors themselves

"—" means minimal, O(K) with low K

55

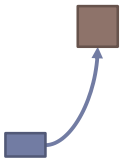
W A R N I N G

EXPERIMENT IN PROGRESS

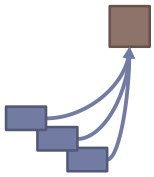
github.com/hsutter/gcpp

56

1 object
1 owner
unique_ptr



1 object
N owners
shared_ptr



these work great when a single object's lifetime is manageable in isolation
(e.g., just looking at immediate neighbors)

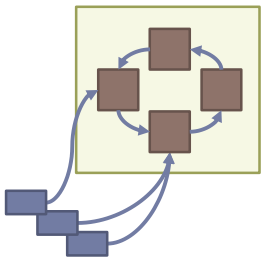
57

1 object
1 owner
unique_ptr

1 object
N owners
shared_ptr

these work great when a single object's lifetime is manageable in isolation
(e.g., just looking at immediate neighbors)

***N* objects**
1 or *N* owners
deferred_ptr
(experimental)



reachability is a property of the whole group
⇒ not detectable from 1 object or subgroup

58

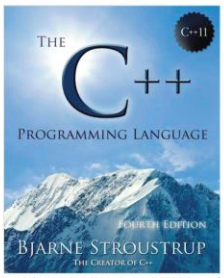
earlier this week at CppCon…

— Bjarne Stroustrup, Monday

in "The Evolution of C++: Past, Present, and Future"

Morgan Stanley

C++ in two lines



- Direct map to hardware
  - of instructions and fundamental data types
  - Initially from C
  - Future: use novel hardware better (caches, multicores, GPUs, FPGAs, SIMD, …)
- Zero-overhead abstraction
  - Classes, inheritance, generic programming, …
  - Initially from Simula (where it wasn't zero-overhead)
  - Future: Type- and resource-safety, concepts, modules, concurrency, …

Stroustrup - CppCon'16 12

earlier this week at CppCon…

— Jason Turner, Wednesday

in "Rich Code for Tiny Machines"

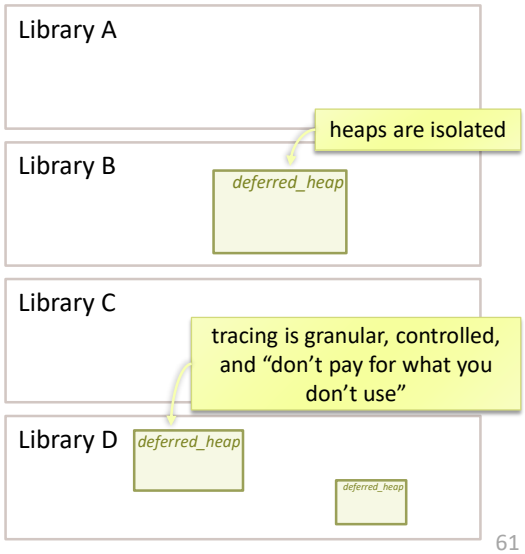
ZERO OVERHEAD ABSTRACTIONS

- Objects
- Methods
- (Delegating) Constructors / Destructors
- Lambdas with/without captures
- Structured bindings (aka destructuring)
- Function calls that have been inlined
- If-init
- Templates (variadic, recursive)
- Standard algorithms

Jason Turner CppCon 2016 @lefticus
http://tinyurl.com/RichCodeCppCon2016

Overview: *deferred_heap* + *deferred_ptr*

- A *deferred_heap* owns a "bubble" of objects that can point to each other arbitrarily.
  - Local heaps ⇒ isolation, composability.
  - Cycles must stay within one heap.
- *.make<T>()* creates an object, stores its destructor, and returns *deferred_ptr<T>*.
- *.collect()* traces this heap, and destroys objects no longer reachable from roots.
  - Fully "don't pay for what you don't use."
- *~deferred_heap()* runs any pending destructors, nulls any *deferred_ptr*s that outlive the heap, and releases its memory all at once like a region.



61

A working demo & source of ideas
Not production-quality (but feedback welcome)

Tries to automate things that require custom code today
⇒ another fallback "when needed," not a primary option

an experiment with
deferred and **unordered** destruction

Goal is to automate tracing
⇒ destructors run **later**, with **deterministic scope and timing**

Emphasis is on **destructors**
⇒ **object** lifetime, not just memory lifetime
⇒ no new "finalizer" concept

In destructors, pointers to other deferred objects are **null**
⇒ safely unordered, enforces "no accidental use of a destroyed object" and "no resurrection"
⇒ destructors of an object tree can be run **iteratively**, not nested

github.com/hsutter/gcpp

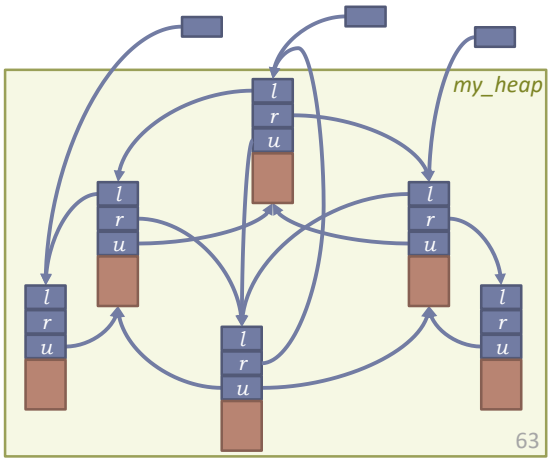
62

Example: Graph

- **Q:** What's the natural ownership abstraction for a possibly-cyclic graph?
- (experimental) **A: deferred_ptr.**
  - No manual part needed:
    Can use *deferred_ptr* to the parent too.

Private heap just for this object, modular & composable

```
class Graph {
  deferred_heap my_heap;
  struct Node {
    deferred_ptr<Node> left, right, up;
    /*… data …*/
  };
  vector<deferred_ptr<Node>> roots;
  /*…*/
};
```



Example: Graph

- **Q:** What's the natural ownership abstracti
- (experimental) **A: deferred_ptr.**
  - No manual part needed:
    Can use *deferred_ptr* to the parent too.

Private heap just for this object, modular & composable

```
class Graph {
  deferred_heap my_heap;
  struct Node {
    deferred_ptr<Node> left, right, up;
    /*… data …*/
  };
  vector<deferred_ptr<Node>> roots;
  /*…*/
};
```

Key: Declares lifetime by construction.

I know without looking at function bodies that there are no leaks. (I don't have to look at the body of *.collect()*, just call it when desired.)

```
void Graph::remove_unused_nodes() {
  heap.collect();
}
```

Pro: Automates tracing, doesn't need to be handwritten with custom code each time.

Pro: No manual "*delete*" masquerading as "*nodes.erase*."

Con: Trimming still requires full traversal + overhead (but limited to this heap ⇒ granular control).

Example: DAG that hands out strong refs

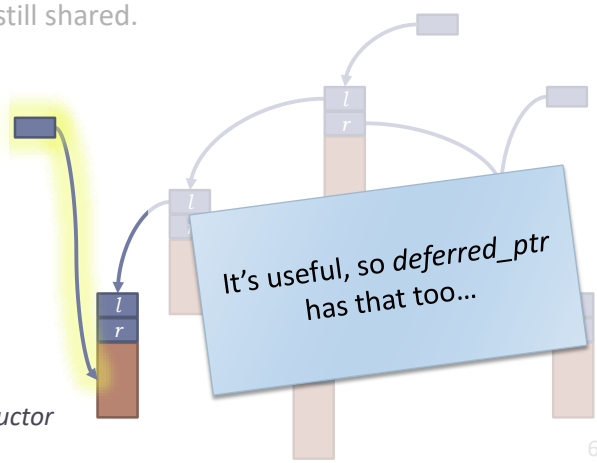
- **Q:** What's the natural ownership abstraction for a DAG that hands out strong refs?

Recall from earlier:

Remember this example of the tree that handed out strong references, and the *shared_ptr* aliasing constructor?

p is still shared.

```
    > parents;
    /*... data ...*/
  };
  vector<shared_ptr<Node>> roots;
  shared_ptr<Data> find(/*...*/) {
    /*...*/ return {spn, &(spn->data)};
  } // note: uses shared_ptr aliasing constructor
};
```

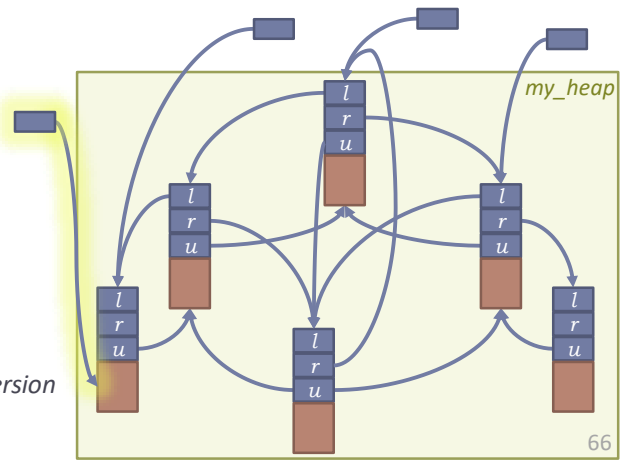


65

Example: Graph that hands out strong refs

- **Q:** What's the natural ownership abstraction for a possibly-cyclic graph?
- (experimental) **A: deferred_ptr.**

```
class Graph {
  deferred_heap my_heap;
  struct Node {
    deferred_ptr<Node> left, right, up;
    /*... data ...*/
  };
  vector<deferred_ptr<Node>> roots;
  /*...*/
  deferred_ptr<Data> find(/*...*/) {
    /*...*/ return dpn.ptr_to(&Node::data);
  } // note: uses deferred_ptr aliasing conversion
};
```

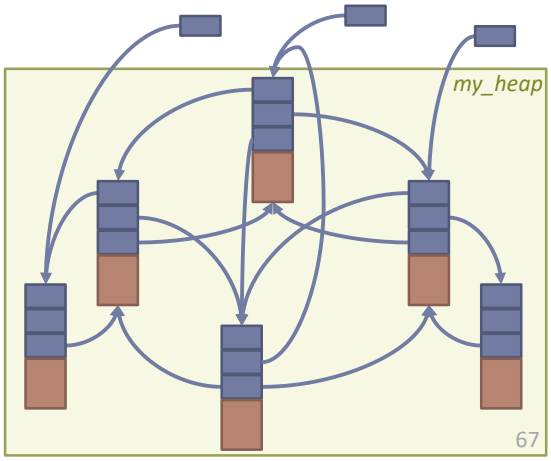


66

Example: N-ary Graph

- **Q:** What if you want an N-ary Graph (not just left and right children)?
- (experimental) **A: vector<T, deferred_allocator<T>>**
  - Convenience alias: *deferred_vector<T>*

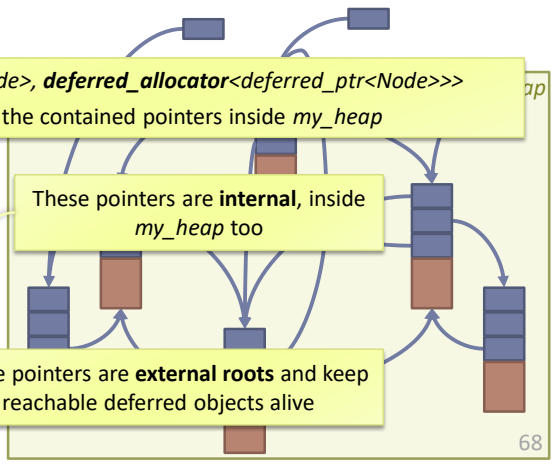
```
class Graph {
  static deferred_heap my_heap;
  struct Node {
    deferred_vector<deferred_ptr<Node>>
      children {my_heap}, parents {my_heap};
    /*… data …*/
  };
  vector<deferred_ptr<Node>> roots;
  /*…*/
};
```



Example: N-ary Graph

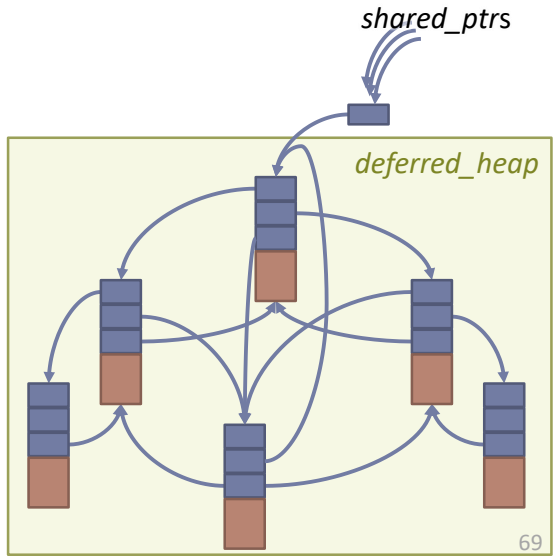
- **Q:** What if you want an N-ary Graph (not just left and right children)?
- (experimental) **A: vector<T, deferred_allocator<T>>**
  - Convenience alias: *deferred_vector<T>*

```
class Graph {
  static deferred_heap my_heap;
  struct Node {
    deferred_vector<deferred_ptr<Node>>
      children {my_heap}, parents {my_heap};
    /*… data …*/
  };
  vector<deferred_ptr<Node>> roots;
  /*…*/
};
```



Composition: RC lifetime → deferred lifetime

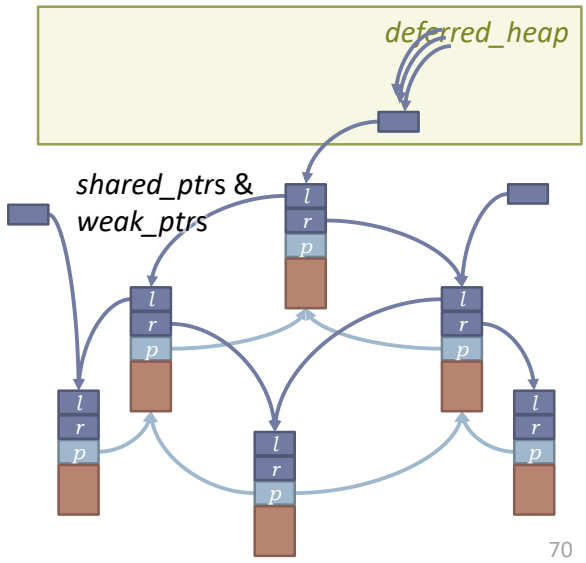
- Consider a graph of deferred objects rooted in an RC'd object.
  - The entire graph has **deterministic** RC lifetime, and gets **ordered** cleanup w.r.t. other such graphs.
  - The deferred objects nevertheless get to easily have **cycles** w.r.t. each other.
- Examples:
  - A *Graph* object and its internal nodes.
  - **Region** lifetime = deallocate-at-once, but with real object destruction.



69

Composition: Deferred lifetime → RC lifetime

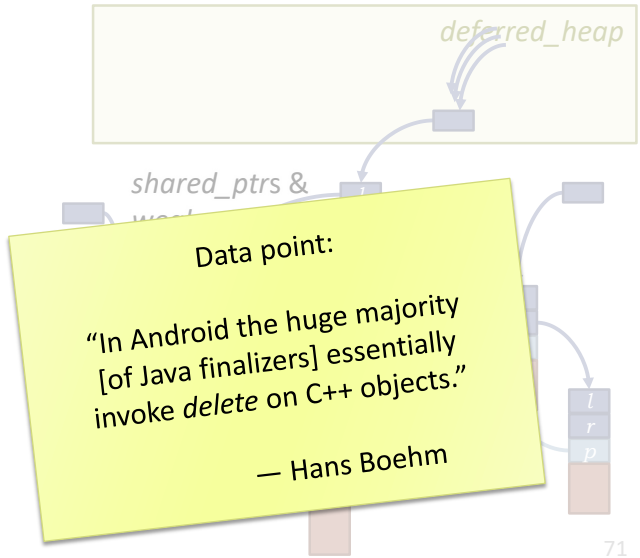
- Consider a graph of RC'd objects rooted in a deferred object.
  - The entire graph has **lazy** lifetime, and can easily participate in **cycles** with other such graphs in the same *deferred_heap*.
  - The RC objects nevertheless get **ordered** cleanup w.r.t. each other, and can call each other.
- Example:
  - A *Tree* object with lazy lifetime, but ordered internal node destruction within the tree.



70

Composition: Deferred lifetime → RC lifetime

- Consider a graph of RC'd objects rooted in a deferred object.
  - The entire graph has **lazy** lifetime, and can easily participate in **cycles** with other such graphs in the same *deferred_heap*.
  - The RC objects nevertheless get **ordered** cleanup w.r.t. each other, and can call each other.
- Example:
  - A *Tree* object with lazy lifetime, but ordered internal node destruction within the tree.



71

*std::*_ptr* limitations

- Ownership cycles ⇒ **leak** objects and memory.
  - *shared_ptr* uses reference counting.
  - Plain reference counting leaks cycles.
  - Can't always be naturally broken using a *weak_ptr*.
- Unbounded pointer copy cost ⇒ exceed **real-time** execution deadlines.
  - *shared_ptr* calls destructors promptly. (Usually a great thing!)
  - But objects can own other objects transitively.
  - Plain assignment to a *shared_ptr* can have arbitrarily high cost. (Similar issue with EH.)
- Unbounded destructor depth ⇒ exceed **stack limits** in constrained environments.
  - Scoped/*unique_ptr*/*shared_ptr* lifetime calls destructors recursively (nested).
  - As above, ownership trees can be deep.
  - Releasing a tree of objects requires having sufficient stack space.

72

*std::*_ptr* limitations

- Ownership cycles ⇒ **leak** objects and memory.
  - *shared_ptr* uses reference counting.
  - nting leaks cycles.
  - urally broken using a *weak_ptr*.
- Unbounded pointer copy cost ⇒ exceed **real-time** execution d
  - *shared_ptr* calls destructors promptly. (Usually a great thing!)
  - But objects can own other objects transitively.
  - Plain assignment to a *shared_ptr* can have arbitrarily high cost. (Similar issue with EH.)
- Unbounded destructor depth ⇒ exceed **stack limits** in constrained environment
  - Scoped/*unique_ptr*/*shared_ptr* lifetime calls destructors recursi
  - p trees can be deep.
  - Releasing a tree of objects requires having sufficient stack space.

Key word: **deferred** destruction

Assigning a *deferred_ptr* doesn't run destructors ⇒ predictable cost

Key word: **unordered** destruction

deferred_heap ::collect() runs destructors iteratively ⇒ less stack depth

73

Note: *deferred_ptr* has limitations too!

- **More total cost** (and/but moved around in different places).
  - It moves work to different places, and does more total work/overhead than *unique_ptr* or *shared_ptr*.
  - Prefer *unique_ptr* or *shared_ptr* in that order where possible, as usual. You should be using them much more frequently than *deferred_ptr*.
- **A demo** of a technique for deferred and unordered destruction.
  - Experimental code to generate ideas you can draw from.
  - Not production quality.

74

Leak freedom: In order of preference

Strategy	Natural examples	Cost	Rough frequency
1. Prefer scoped lifetime by default (locals, members)	Local and member objects – directly owned	Zero: Tied directly to another lifetime	O(80%) of objects
2. Else prefer make_unique & unique_ptr or a container, if the object must have its own lifetime (i.e., heap) and ownership can be unique w/o owning cycles	Implementations of trees, lists	Same as new/delete & malloc/free **Automates** simple heap object use in a library	O(20%) of objects
3. Else prefer make_shared & shared_ptr, if the object must have its own lifetime (i.e., heap) and shared ownership w/o owning cycles	Node-based DAGs, incl. trees that share out references	Same as manual reference counting (RC) **Automates** shared object use in a library	
4. Else defer destruction and have logic to trace unreachable objects (maybe eventually automate in a library?)	Graphs, real-time code, limited stacks	Deferred destructors and tracing logic	O(1%) of objects

76

Leak Freedom in C++… *By Default*

Questions?

FAQs

- Q: Is this garbage collection?
  - Of course, as is reference counting (e.g., *shared_ptr*).
- Q: I meant, is this like tracing GC in other popular languages?
  - No. It collects objects, not garbage.
  - Also: It's granular and opt-in, not global and default.
- Q: Is this related/similar to the Boehm collector?
  - No. It runs destructors, and tracing is granular (scoped to an individual heap) and accurate (not conservative).
- Q: Is this like a region?
  - It's a superset.
  - *deferred_heap* = region + $collect()_{opt}$ + destructors + nulling roots.

81

deferred_heap requirements & constraints

Do this… (appropriate for C++)

- **Opt-in:** Not the primary allocator, available to use just when you need it
- **Zero-overhead == "don't pay for what you don't use":** Cost ∝ GC allocations only
- **Deterministic:** Collection scope + timing
- **Real destructors:** No separate "finalizer" concept needed
- **Solid resource cleanup:** Enforce "don't access other nontrivially destructible objects during cleanup" (can't write it), prevent "resurrection" (can't write it)

… not that (ok for other languages)

- Forced-in: Default/only allocator, must fight the language to get out
- Global cost (incl. "conservative" GC): Incurs overhead on code/data that doesn't use it
- Nondeterministic: Hard to control pauses
- Distinct-but-related finalizer: Leads to brittle "Dispose(bool)" patterns due to overlap
- Complex resource cleanup: Unenforced guidance to "don't access other finalizable objects" guidance, allows re-reachability "resurrection"

82