

COPYING & MOVING

LECTURE 14-1

JIM FIX, REED COLLEGE CS2-F20

LOGISTICS

▶ Office hours:

→ **THIS WEEK:** Tues & Weds 11am-12pm, 3-4:30pm

→ **NEXT WEEK:** Mon & Tues 11am-12pm, 3-4:30pm

▶ Final Exam on circuits, MIPS, & modern C++

→ I've posted a practice exam today.

→ I will post my solutions to the practice exam tomorrow night.

→ **Download/upload** the exam on Thursday, December 10th, 1-5pm.

▶ Homework 12 & 13 on lambdas, std::vector, inheritance

→ **Due:** Tuesday December 15th, 11:50pm

HEAP OBJECTS AS RESOURCES

- ▶ In the last lecture we looked at C++ STL "smart pointers". Three types:
 - **unique_ptr**: only one copy of the pointer; passed around with *moves*
 - **shared_ptr**: several copies; references counted with shared *copies*
 - **weak_ptr**: uncounted; can be *copied* and *moved* but may be stale
- ▶ These were developed in C++ 11 to provide the **RAII** discipline:
 - **Resource Acquisition Is Initialization**
 - Careful management of heap memory; object lifetimes.
 - A pointer to heap memory is a **resource** that must be managed.
 - Tricky issues: concurrency, robustness in the face of exceptions, etc.
- ▶ ***Reasoning about the heap is tricky. Need oversight and discipline.***

2016 GÖDEL PRIZE: LOGIC OF HEAP MEMORY

▶ From the *European Association for Theoretical Computer Science*

"The... Gödel Prize is awarded to Stephen Brookes and Peter W. O'Hearn for their invention of Concurrent Separation Logic, as described in the following two papers:"

→ S. Brookes, *"A Semantics for Concurrent Separation Logic."*

Theoretical Computer Science 375(1-3): 227-270 (2007)

→ P. W. O'Hearn, *"**Resources**, Concurrency, and Local Reasoning."*

Theoretical Computer Science 375(1-3): 271-307 (2007)

ON "CONCURRENT SEPARATION LOGIC"

- ▶ "Concurrent Separation Logic (CSL) is a revolutionary advance over previous proof systems for verifying properties of systems software, which commonly involve both pointer manipulation and shared-memory concurrency. For the last thirty years experts have regarded pointer manipulation as an unsolved challenge for program verification and shared-memory concurrency as an even greater challenge. Now, thanks to CSL, both of these problems have been elegantly and efficiently solved; and they have the same solution. Brookes and O'Hearn's approach builds on the Separation Logic for sequential programs due to O'Hearn and the late John Reynolds. The extension to treat concurrently executing programs communicating via shared state is highly non-trivial and involves a dynamic notion of resource ownership that supports modular reasoning."

RESOURCE MANAGEMENT VIEWPOINT

- ▶ An object's memory storage is a *resource*
 - It might be shared amongst several parts of the program
 - If so, treat it specially. Can't delete if shared.
 - It might not be shared. Maybe only one part of the program is *using* it.
 - When that part of the program is done with it, it should delete it.
- ▶ Some languages and language libraries work to make this explicit
 - They help your code manage *ownership*
- ▶ **E.g.** the **Rust** programming language
 - has a notion of *borrowing* an object; and transfer of single ownership
 - compiler has a *borrow checker*; based on *linear* types

TODAY'S PLAN

- ▶ WE BREAK **dc** WITH ONE SMALL CHANGE...
 - WE INVESTIGATE TWO TEST PROGRAMS:
 - A SIMPLE CLASS WITH A VALUE MEMBER
 - A SIMPLE CLASS WITH A HEAP-ALLOCATED MEMBER
 - WE DISCUSS:
 - COPY CONSTRUCTORS, COPY ASSIGNMENT
 - MOVE CONSTRUCTORS, MOVE ASSIGNMENT
- ▶ WE EXPLAIN & FIX THE BUG

ONE SMALL CHANGE

- ▶ Let's make a small change to my stack-based calculator **dc.c**

```
void output_top(Stck s) {
    if (!s.is_empty()) {
        std::cout << s.top() << std::endl;
    }
}
```

```
int main() {
    ...
    Stck s {100};
    std::string entry;
    do {
        output_top(s);
        // parse and handle entry
        ...
    } while (entry != q);
}
```


ONE SMALL CHANGE

- ▶ Let's make a small change to my stack-based calculator **dc.c**

```
void output_top(Stck s) {  
    if (!s.is_empty()) {  
        std::cout << s.top() << std::endl;  
    }  
}
```

```
int main() {  
    ...  
    Stck s {100};  
    std::string entry;  
    do {  
        output_top(s);  
        // parse and handle entry  
        ...  
    } while (entry != q);  
}
```

ONE SMALL CHANGE

- ▶ Let's make a small change to my stack-based calculator **dc.c**

```
void output_top(Stck s) {  
    if (!s.is_empty()) {  
        std::cout << s.top() << std::endl;  
    }  
}
```

```
int main() {  
    ...  
    Stck s {100};  
    std::string entry;  
    do {  
        output_top(s);  
        // parse and handle entry  
        ...  
    } while (entry != q);  
}
```

A MYSTERY

- ▶ Here is what happens when I recompile and run it...

```
$ ./dc
You've just run my version of the Unix calculator utility
'dc'.
It uses a stack to track intermediate calculations.
Enter a command just below (h for help):
p
[ ]
dc(23213,0x7fff7c877000) malloc: *** error for object
0x7fa93ae00000: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
```

A MYSTERY

- ▶ Here is what happens when I recompile and run it...

```
$ ./dc
You've just run my version of the Unix calculator utility
'dc'.
It uses a stack to track intermediate calculations.
Enter a command just below (h for help):
p
[ ]
dc(23213,0x7fff7c877000) malloc: *** error for object
0x7fa93ae00000: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
```

- ▶ Let's work a bit to explain why...

TWO SAMPLE CLASSES

- ▶ Today we examine some trickier aspects of C++ storage management.
- ▶ We'll reference two simple class definitions.
- ▶ The first class **v** has a single instance variable of type **int**

```
class V {  
private:  
    int x;  
public:  
    V(void);  
    V(int x0);  
    ~V(void);  
    friend V operator+(int i, V&& v);  
}
```

TWO SAMPLE CLASSES

- ▶ Today we examine some trickier aspects of C++ storage management.
- ▶ We'll reference two simple class definitions.
- ▶ The second one **R** has an instance variable of type **int***

```
class R {  
private:  
    int* a;  
public:  
    R(void);  
    R(int x0);  
    ~R(void);  
    friend R operator+(int i, R&& r);  
}
```

TWO SAMPLE CLASSES

- ▶ Today we examine some trickier aspects of C++ storage management.
- ▶ We'll reference two simple class definitions.
- ▶ They each can get **built two ways**:

```
class V {
private:
    int x;
public:
    V(void) : x {0} { };
    V(int x0) : x {x0} { };
    ~V(void);
    friend V operator+(int i, V&& v);
}
```

TWO SAMPLE CLASSES

- ▶ Today we examine some trickier aspects of C++ storage management.
- ▶ We'll reference two simple class definitions.
- ▶ They each can get **built two ways**:

```
class R {
private:
    int* a;
public:
    R(void) : a {nullptr} { };
    R(int x0) : a {new int[1]} { a[0] = x0};
    ~R(void);
    friend R operator+(int i, R&& r);
}
```


TWO SAMPLE CLASSES

- ▶ Today we examine some trickier aspects of C++ storage management.
- ▶ We'll reference two simple class definitions.
- ▶ They each can get **built two ways**:

```
class R {  
private:  
    int* a;  
public:  
    R(void) : a {nullptr} { };  
    R(int x0) : a {new int[1]} { a[0] = x0};  
    ~R(void);  
    friend R operator+(int i, R&& r);  
}
```

- ▶ We'll look at destructors, copying, **moving**.

FYI: TRACKING CONSTRUCTION

- ▶ *In the sample folder*, I have a second version of each that also stores an ID.

```
class V {
private:
    static int next_id;
    int id;
    int x;
    void give_id(void) { id = ++next_id; }
public:
    V(void) : x {0} { give_id(); };
    V(int x0) : x {x0} { give_id(); };
    ~V(void);
    friend V operator+(int i, V&& v);
}

int V::next_id = 0;
```

- ▶ I did this in my tests there to help track what's going on.

THE COPY CONSTRUCTOR

- ▶ A copy constructor is one that is used to construct an instance from another.
- ▶ Here is an example for the "value class" **V**:

```
V(const V& ov) : x {ov.x} { }
```

- ▶ Here we are simply copying the contents of another **V** instance **ov**

THE COPY CONSTRUCTOR

- ▶ A copy constructor is one that is used to construct an instance from another.
- ▶ Here is an example for the "value class" **V**:

```
V(const V& ov) : x {ov.x} { }
```

- ▶ Here we are simply copying the contents of another **V** instance **ov**
- ▶ The second line below gives its standard use:

```
V v1 {42}; // This calls the V(int) constructor.  
V v2 {v1}; // This calls the copy constructor.
```

THE COPY CONSTRUCTOR

- ▶ A copy constructor is one that is used to construct an instance from another.
- ▶ Here is an example for the "value class" **V**:

```
V(const V& ov) : x {ov.x} { }
```

- ▶ Here we are simply copying the contents of another **V** instance **ov**
- ▶ The second line below gives its standard use:

```
V v1 {42}; // This calls the V(int) constructor.  
V v2 {v1}; // This calls the copy constructor.
```

THE COPY CONSTRUCTOR

- ▶ A copy constructor is one that is used to construct an instance from another.
- ▶ Here is an example for the "value class" **V**:

```
V(const V& ov) : x {ov.x} { }
```

- ▶ Here we are simply copying the contents of another **V** instance **ov**

NOTE: the copy constructor is one

that matches this exact signature

- ▶ Here This second line below gives its standard use:

```
V v1 {42}; // This calls the V(int) constructor.  
V v2 {v1}; // This calls the copy constructor.
```

THE COPY CONSTRUCTOR

- ▶ A copy constructor is one that is used to construct an instance from another.
- ▶ Here is an example for the "value class" **V**:

```
V(const V& ov) : x {ov.x} { }
```

- ▶ Here we are simply copying the contents of another **V** instance **ov**
- ▶ **NOTE:** the copy construct gets applied in several other situations:
 - When a function is passed a **V** parameter *by value*

THE COPY CONSTRUCTOR

- ▶ A copy constructor is one that is used to construct an instance from another.
- ▶ Here is an example for the "value class" **V**:

```
V(const V& ov) : x {ov.x} { }
```

- ▶ Here we are simply copying the contents of another **V** instance **ov**
- ▶ **NOTE:** the copy construct gets applied in several other situations:
 - When a function is passed a **V** parameter *by value*
 - When a function returns a **V** by value

THE COPY CONSTRUCTOR

- ▶ A copy constructor is one that is used to construct an instance from another.
- ▶ Here is an example for the "value class" **V**:

```
V(const V& ov) : x {ov.x} { }
```

- ▶ Here we are simply copying the contents of another **V** instance **ov**

- ▶ **NOTE:** the copy construct gets applied in several other situations:

- When a function is passed a **V** parameter *by value*

- When a function returns a **V** by value

- It's also used when there is a trivial initialization assignment:

```
V v2 = V {v1};
```

COPY CONSTRUCTOR APPLICATIONS

- ▶ When a **V** is constructed using a **V**:

```
V v2 {v1};
```

- ▶ When a function is passed a **V** parameter by value:

```
int get_value(V v) { ... }  
...  
int i = get_value(v1);
```

- ▶ When a function returns a **V** by value:

```
V get_V(...) {  
  V my_v;  
  ...  
  return my_v;  
}  
...  
V their_v = get_V(...);
```

- ▶ When an assignment is actually a **V** initialization:

```
V v2 = V {v1};
```

THE COPY ASSIGNMENT OPERATOR

- ▶ A similarly behaving member is the **copy assignment operator**
- ▶ Here is an example for the "value class" **V**:

```
V& operator=(const V& ov) { x = ov.x; return *this; }
```

- ▶ It gets used most times that there is a **V** assignment:

```
V v1 {42};  
V v2 {87};  
...  
v2 = v1;
```

THE COPY ASSIGNMENT OPERATOR

- ▶ A similarly behaving member is the **copy assignment operator**
- ▶ Here is an example for the "value class" **V**:

```
V& operator=(const V& ov) { x = ov.x; return *this; }
```

- ▶ It gets used most times that there is a **V** assignment:

```
V v1 {42};  
V v2 {87};  
...  
v2 = v1;
```

THE COPY ASSIGNMENT OPERATOR

- ▶ A similarly behaving member is the **copy assignment operator**
- ▶ Here is an example for the "value class" **V**:

```
V& operator=(const V& ov) { x = ov.x; return *this; }
```

- ▶ It gets used most times that there is a **V** assignment:

```
V v1 {42};  
V v2 {87};  
V v3 {99};  
...  
v3 = v2 = v1;
```

- ▶ It has this weird signature returning the assigned object as a reference because some C programmers like to **chain assignments**.

THE COPY ASSIGNMENT OPERATOR

- ▶ A similarly behaving member is the **copy assignment operator**
- ▶ Here is an example for the "value class" **V**:

```
V& operator=(const V& ov) { x = ov.x; return *this; }
```

- ▶ There are cases where it might not get used...

```
V v1 {42};  
V v2 {87};  
V v3 {99};  
...  
V v4 = V {v3}; // This, we saw, uses the copy constructor.  
V v5 = V {101}; // This uses the V(int) constructor.  
v3 = V {789}; // And this uses move assignment
```

THE COPY ASSIGNMENT OPERATOR

- ▶ A similarly behaving member is the **copy assignment operator**
- ▶ Here is an example for the "value class" **V**:

```
V& operator=(const V& ov) { x = ov.x; return *this; }
```

- ▶ There are cases where it **might not get used...**

```
V v1 {42};  
V v2 {87};  
V v3 {99};  
...  
V v4 = V {v3}; // This, we saw, uses the copy constructor.  
V v5 = V {101}; // This uses the V(int) constructor.  
v3 = V {789}; // And this uses move assignment
```

THE COPY ASSIGNMENT OPERATOR

- ▶ A similarly behaving member is the **copy assignment operator**
- ▶ Here is an example for the "value class" **V**:

```
V& operator=(const V& ov) { x = ov.x; return *this; }
```

- ▶ There are cases where it **might not get used...**

```
V v1 {42};  
V v2 {87};  
V v3 {99};  
...  
V v4 = V {v3}; // This, we saw, uses the copy constructor.  
V v5 = V {101}; // This uses the V(int) constructor.  
v3 = V {789}; // And this uses move assignment
```

- ▶ WHY? Because **V{789}** is immediately discarded.

MOVE ASSIGNMENT

- ▶ Here is an example definition of a *move assignment operator*

```
V& operator=(V&& ov) { x = ov.x; return *this; }
```

- ▶ Here is that typical situation when it gets used

```
V v3 {99};  
...  
v3 = V {789};
```

- ▶ Since **V{789}** is a temporary object, it doesn't take up resources (i.e. no slot in the stack frame).
 - The object **v3** is seen to be "taking over its resources."
 - The temporary **V** is seen as "moving out", and **v3** is seen as "moving in."

MOVE ASSIGNMENT

- ▶ Here is an example definition of a **move assignment operator**

```
V& operator=(V&& ov) { x = ov.x; return *this; }
```

- ▶ Here again is a typical situation when it gets used

```
V v3 {99};  
...  
v3 = v {789};
```

- ▶ It has a weird annotation of its argument.
- ▶ This is an **R-value reference**
 - L-value expressions are ones that can appear on the LHS of an assignment
 - R-value expressions are ones that only appear on the RHS of an assignment...

MOVE ASSIGNMENT

- ▶ Here is an example definition of a **move assignment operator**

```
V& operator=(V&& ov) { x = ov.x; return *this; }
```

- ▶ Here again is a typical situation when it gets used

```
V v3 {99};  
...  
v3 = V {789};
```

- ▶ It has a weird annotation of its argument.
- ▶ This is an **R-value reference**
 - L-value expressions are ones that can appear on the LHS of an assignment
 - R-value expressions are ones that only appear on the RHS of an assignment... like **V{789}**

MOVE CONSTRUCTOR

- ▶ There is also a ***move constructor***

```
V(V&& ov) : x {ov.x} { }
```

- ▶ Here is a typical situation when it gets used:

```
V make_a_V(int x0) {  
    return V {x0};  
}
```

...

```
v3 = make_a_V(789);
```

MOVE CONSTRUCTOR

- ▶ There is also a ***move constructor***

```
V(V&& ov) : x {ov.x} { }
```

- ▶ Here is a typical situation when it gets used:

```
V make_a_V(int x0) {  
    return V {x0};  
}
```

...

```
v3 = make_a_V(789);
```

- ▶ Note again the use of an **R-value reference annotation**.

AN EXAMPLE USE OF &&

- ▶ I tried to demonstrate these things in the sample code for this lecture.
 - So far, in **samples/copy_move/cm_value_debug.cc**
 - Run **make** to build an executable **./cmvd** and look at its output.
- ▶ There's an additional definition:

```
class V {  
    ...  
    friend int operator+(int i, V&& v);  
};  
int operator+(int i, V&& v) { return i+v.x; }
```

AN EXAMPLE USE OF &&

- ▶ I tried to demonstrate these things in the sample code for this lecture.
 - So far, in **samples/copy_move/cm_value_debug.cc**
 - Run **make** to build an executable **./cmvd** and look at its output.
- ▶ There's an additional definition:

```
class V {  
    ...  
    friend int operator+(int i, V&& v);  
};  
int operator+(int i, V&& v) { return i+v.x; }
```

- ▶ Here is where **it is used**:

```
V v4 = V{1 + V {3}};
```

AN EXAMPLE USE OF &&

- ▶ I tried to demonstrate these things in the sample code for this lecture.
 - So far, in **samples/copy_move/cm_value_debug.cc**
 - Run **make** to build an executable **./cmvd** and look at its output.
- ▶ There's an additional definition:

```
class V {  
    ...  
    friend int operator+(int i, V&& v);  
};  
int operator+(int i, V&& v) { return i+v.x; }
```

- ▶ Here is where **it is used**:

```
V v4 = V{1 + V {3}};
```

- Note the use of an **R-value reference** in its definition.

CONTAINER CLASSES AND COPY/MOVE

- ▶ Recall my array-based class **R**, a companion to class **V**

```
class R {  
private:  
    int* a;  
public:  
    R(void) : a {nullptr} { };  
    R(int x0) : a {new int[1]} { a[0] = x0};  
    ~R(void) { if (a != nullptr) delete [] a; }  
}
```

- ▶ Note that I allocate the array upon construction with a value.
- ▶ Note that I wrote the default constructor to set a null pointer instead.

CONTAINER CLASSES AND COPY/MOVE

- ▶ Recall my array-based class **R**, a companion to class **V**

```
class R {
private:
    int* a;
public:
    R(void) : a {nullptr} { };
    R(int x0) : a {new int[1]} { a[0] = x0};
    ~R(void) { if (a != nullptr) delete [] a; }
}
```

- ▶ Note that I allocate the array upon construction with a value.
- ▶ Note that I wrote the default constructor to set a null pointer instead.

CONTAINER CLASSES AND COPY/MOVE

- ▶ Recall my array-based class **R**, a companion to class **V**

```
class R {  
private:  
    int* a;  
public:  
    R(void) : a {nullptr} { };  
    R(int x0) : a {new int[1]} { a[0] = x0};  
    ~R(void) { if (a != nullptr) delete [] a; }  
}
```

- ▶ Note that I allocate the array upon construction with a value.
- ▶ Note that I wrote the default constructor to **set a null pointer** instead.

CONTAINER CLASSES AND COPY/MOVE

- ▶ Recall my array-based class **R**, a companion to class **V**

```
class R {  
private:  
    int* a;  
public:  
    R(void) : a {nullptr} { };  
    R(int x0) : a {new int[1]} { a[0] = x0};  
    ~R(void) { if (a != nullptr) delete [] a; }  
}
```

- ▶ Note that I allocate the array upon construction with a value.
- ▶ Note that I wrote the default constructor to **set a null pointer** instead...
 - ... so that I could write move constructors that don't leak memory.

CONTAINER CLASSES AND COPY/MOVE

- ▶ Recall my array-based class **R**, a companion to class **V**

```
class R {  
private:  
    int* a;  
public:  
    R(void) : a {nullptr} { };  
    R(int x0) : a {new int[1]} { a[0] = x0};  
    ~R(void) { if (a != nullptr) delete [] a; }  
}
```

- ▶ Note that I allocate the array upon construction with a value.
- ▶ Note that I wrote the default constructor to set a null pointer instead.
- ▶ Note that I give back the array storage in the destructor, if not null.

CONTAINER CLASSES AND COPY/MOVE

- ▶ Recall my array-based class **R**, a companion to class **V**

```
class R {  
private:  
    int* a;  
public:  
    R(void) : a {nullptr} { };  
    R(int x0) : a {new int[1]} { a[0] = x0};  
    ~R(void) { if (a != nullptr) delete [] a; }  
}
```

- ▶ Note that I allocate the array upon construction with a value.
- ▶ Note that I wrote the default constructor to set a null pointer instead.
- ▶ Note that I give back the array storage in the destructor, if not null.

CONTAINER CLASSES AND COPY/MOVE

- ▶ Recall my array-based class **R**, a companion to class **V**

```
class R {  
private:  
    int* a;  
public:  
    R(void) : a {nullptr} { };  
    R(int x0) : a {new int[1]} { a[0] = x0};  
    ~R(void) { if (a != nullptr) delete [] a; }  
}
```

- ▶ Note that I allocate the array upon construction with a value.
- ▶ Note that I wrote the default constructor to set a null pointer instead.
- ▶ Note that I give back the array storage in the destructor, if not null.
- ▶ What should the copy/move members do?

COPY CONSTRUCTOR AND ASSIGNMENT

- ▶ Here are the copy operations for class **R**

```
R::R(const R& r) : a {new int[1]} {  
    a[0] = r.a[0];  
}
```

```
R& R::operator=(const R& r) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
    a = new int[1];  
    a[0] = r.a[0];  
    return *this;  
}
```

- ▶ They each perform a deep copy of the data structure.

COPY CONSTRUCTOR AND ASSIGNMENT

- ▶ Here are the copy operations for class **R**

```
R::R(const R& r) : a {new int[1]} {  
    a[0] = r.a[0];  
}
```

```
R& R::operator=(const R& r) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
    a = new int[1];  
    a[0] = r.a[0];  
    return *this;  
}
```

- ▶ They each perform a **deep copy** of the data structure.

COPY CONSTRUCTOR AND ASSIGNMENT

- ▶ Here are the copy operations for class **R**

```
R::R(const R& r) : a {new int[1]} {  
    a[0] = r.a[0];  
}
```

```
R& R::operator=(const R& r) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
    a = new int[1];  
    a[0] = r.a[0];  
    return *this;  
}
```

- ▶ They each perform a deep copy of the data structure.
- ▶ But we also have to deallocate the destination's old storage.

MOVE CONSTRUCTOR AND ASSIGNMENT

- ▶ Here are the move operations for class **R**

```
R::R(R&& r) {  
    a = r.a;  
    r.a = nullptr;  
}  
R& R::operator=(R&& r) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
    a = r.a;  
    r.a = nullptr;  
    return *this;  
}
```

- ▶ They can perform a shallow copy of the source object's data.

MOVE CONSTRUCTOR AND ASSIGNMENT

- ▶ Here are the move operations for class **R**

```
R::R(R&& r) {
    a = r.a;
    r.a = nullptr;
}
R& R::operator=(R&& r) {
    if (a != nullptr) {
        delete [] a;
    }
    a = r.a;
    r.a = nullptr;
    return *this;
}
```

- ▶ They can perform a **shallow copy** of the source object's data.

MOVE CONSTRUCTOR AND ASSIGNMENT

- ▶ Here are the move operations for class **R**

```
R::R(R&& r) {
    a = r.a;
    r.a = nullptr;
}
R& R::operator=(R&& r) {
    if (a != nullptr) {
        delete [] a;
    }
    a = r.a;
    r.a = nullptr;
    return *this;
}
```

- ▶ They can perform a shallow copy of the source object's data.
- ▶ We still need to **give back the destination's old array** upon reassignment.

MOVE CONSTRUCTOR AND ASSIGNMENT

- ▶ Here are the move operations for class **R**

```
R::R(R&& r) {  
    a = r.a;  
    r.a = nullptr;  
}  
R& R::operator=(R&& r) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
    a = r.a;  
    r.a = nullptr;  
    return *this;  
}
```

- ▶ They can perform a shallow copy of the source object's data.
- ▶ We still need to give back the destination's old array upon reassignment.
- ▶ And it is standard practice to "clear out" the source of the move.

MOVE CONSTRUCTOR AND ASSIGNMENT

- ▶ Here are the move operations for class **R**

```
R::R(R&& r) {  
    a = r.a;  
    r.a = nullptr;  
}  
R& R::operator=(R&& r) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
    a = r.a;  
    r.a = nullptr;  
    return *this;  
}
```

```
R::~~R(void) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
}
```

- ▶ They can perform a shallow copy of the source object's data.
- ▶ We still need to give back the destination's old array upon reassignment.
- ▶ We *clear out* the source of the move in preparation for its destruction.

SHALLOW COPY CONSTRUCTOR AND ASSIGNMENT BUGGY!

- ▶ Here instead are shallow copy operations for class **R**

```
R::R(const R& r) : a {r.a} { }
```

```
R& R::operator=(const R& r) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
    a = r.a;  
    return *this;  
}
```

- ▶ With these, we would have instances of **R** *sharing* the same array **a**.
- ▶ The destructor would eventually "double delete" that shared pointer.
- ▶ **NOTE** that shallow copying is sometimes desirable...

SHALLOW COPY CONSTRUCTOR AND ASSIGNMENT BUGGY!

- ▶ Here instead are shallow copy operations for class **R**

```
R::R(const R& r) : a {r.a} { }
```

```
R& R::operator=(const R& r) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
    a = r.a;  
    return *this;  
}
```

- ▶ With these, we would have instances of **R** *sharing* the same array **a**.
- ▶ The destructor would eventually "double delete" that shared pointer.
- ▶ **NOTE** that shallow copying is sometimes desirable...
 - ...and the STL "smart pointers" will allow us to do sharing, in a smart way.

LET'S REVISIT THE MYSTERY BUG FROM THE START

► RECALL: my change to **dc.c**

```
void output_top(Stck s) {
    if (!s.is_empty()) {
        std::cout << s.top() << std::endl;
    }
}
```

```
int main() {
    ...
    Stck s {100};
    std::string entry;
    do {
        output_top(s);
        // parse and handle entry
        ...
    } while (entry != q);
}
```

LET'S REVISIT THE MYSTERY BUG FROM THE START

► **RECALL:** what happened when I ran it...

```
$ ./dc
You've just run my version of the Unix calculator utility 'dc'.
It uses a stack to track intermediate calculations.
Enter a command just below (h for help):
p
[ ]
dc(23213,0x7fff7c877000) malloc: *** error for object
0x7fa93ae00000: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
```

LET'S REVISIT THE MYSTERY BUG FROM THE START

► **RECALL:** what happened when I ran it...

```
$ ./dc
You've just run my version of the Unix calculator utility 'dc'.
It uses a stack to track intermediate calculations.
Enter a command just below (h for help):
p
[ ]
dc(23213,0x7fff7c877000) malloc: *** error for object
0x7fa93ae00000: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
```

Q: So what went wrong????

LET'S REVISIT THE MYSTERY BUG FROM THE START

► **RECALL:** what happened when I ran it...

```
$ ./dc
You've just run my version of the Unix calculator utility 'dc'.
It uses a stack to track intermediate calculations.
Enter a command just below (h for help):
p
[ ]
dc(23213,0x7fff7c877000) malloc: *** error for object
0x7fa93ae00000: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
```

Q: So what went wrong????

A: I don't define a copy constructor for **Stack**. The default one does a shallow copy. When I pass **s** by value to **output_top** it is passed by value. The two stack objects share a pointer to an array's data. It exits, the destructor gets called. It deletes the **elements** pointer.

ONE POSSIBLE FIX

```
void output_top(Stack s) { // passed by value; copies
    if (!s.is_empty()) {
        std::cout << s.top() << std::endl;
    }
}

int main() {
    ...
    Stack s {100};
    std::string entry;
    do {
        output_top(s);
        // parse and handle entry
        ...
    } while (entry != q);
}
```

ONE POSSIBLE FIX

```
void output_top(Stck &s) { // pass s by ref, no copy made
    if (!s.is_empty()) {
        std::cout << s.top() << std::endl;
    }
}

int main() {
    ...
    Stck s {100};
    std::string entry;
    do {
        output_top(s);
        // parse and handle entry
        ...
    } while (entry != q);
}
```

SUMMARY

- ▶ COPY CONSTRUCTORS
- ▶ COPY ASSIGNMENT
- ▶ MOVE CONSTRUCTORS
- ▶ MOVE ASSIGNMENT
- ▶ ... are each used by the C++ compiler in various ways.
- ▶ If you are rolling your own data structures, then you need to become an expert and understand their subtleties.
- ▶ *Probably best to learn what's provided by the C++ Standard Template Library*

TL;DR SUMMARY

- ▶ EXPLICIT MEMORY MANGEMENT...
 - ESPECIALLY THE ABILITY TO ALLOCATE ON THE STACK **AND** IN THE HEAP
 - ▶ ...MAKES C++ A VERY COMPLEX LANGUAGE TO LEARN.
-
- ▶ *Probably **still** need to understand quite a bit to understand what's provided by the C++ Standard Template Library*

RECALL: SMART POINTERS IN THE C++ STL

- ▶ The C++ STL provides three template types (`#include <memory>`)
 - `std::shared_ptr<T>`: used to reference an object shared by several code components. It maintains a count of these. *Copying* a shared pointer increments this count. If a `shared_ptr` variable loses scope or if an object with a `shared_ptr` component is `deleted`, it is decremented.
 - `std::weak_ptr<T>`: only constructable from a `shared_ptr` without incrementing its count. Used many ways, including in cyclic structures.
- ▶ There is a third type. Explaining it is tricky now: *copying* versus *moving*
 - `std::unique_ptr<T>`: used to reference an object owned by one code component (i.e. one variable). It cannot be *copied*. It can be *moved*.

A SHARED_PTR SINGLY LINKED LIST

```
#include <memory>

class node {
public:
    int data;
    std::shared_ptr<node> next;
    node(int value) : data {value}, next {nullptr} { }
    ~node(void) { }
};

class llist {
private:
    std::shared_ptr<node> first;
    std::shared_ptr<node> last;
public:
    llist(void) : first {nullptr}, last {nullptr} { }
    ~llist(void) { // NOTHING HERE!! }
    void prepend(int value);
    void append(int value);
    void remove(int value);
};
```

A SHARED_PTR SINGLY LINKED LIST

```
#include <memory>

class node {
public:
    int data;
    std::shared_ptr<node> next;
    node(int value) : data {value}, next {nullptr} { }
    ~node(void) { }
};

class llist {
private:
    std::shared_ptr<node> first;
    std::shared_ptr<node> last;
public:
    llist(void) : first {nullptr}, last {nullptr} { }
    ~llist(void) { // NOTHING HERE!! }
    void prepend(int value) { ... // next slides }
    void append(int value) { ... // next slides }
    void remove(int value) { ... // next slides }
};
```

LINKED LIST SHARED_PTR NODE ALLOCATION

```
void llist::prepend(int value) {  
    std::shared_ptr<node> newNode {new node {value}};  
    newNode->next = first;  
    first = newNode;  
    if (last == nullptr) {  
        last = first;  
    }  
}
```

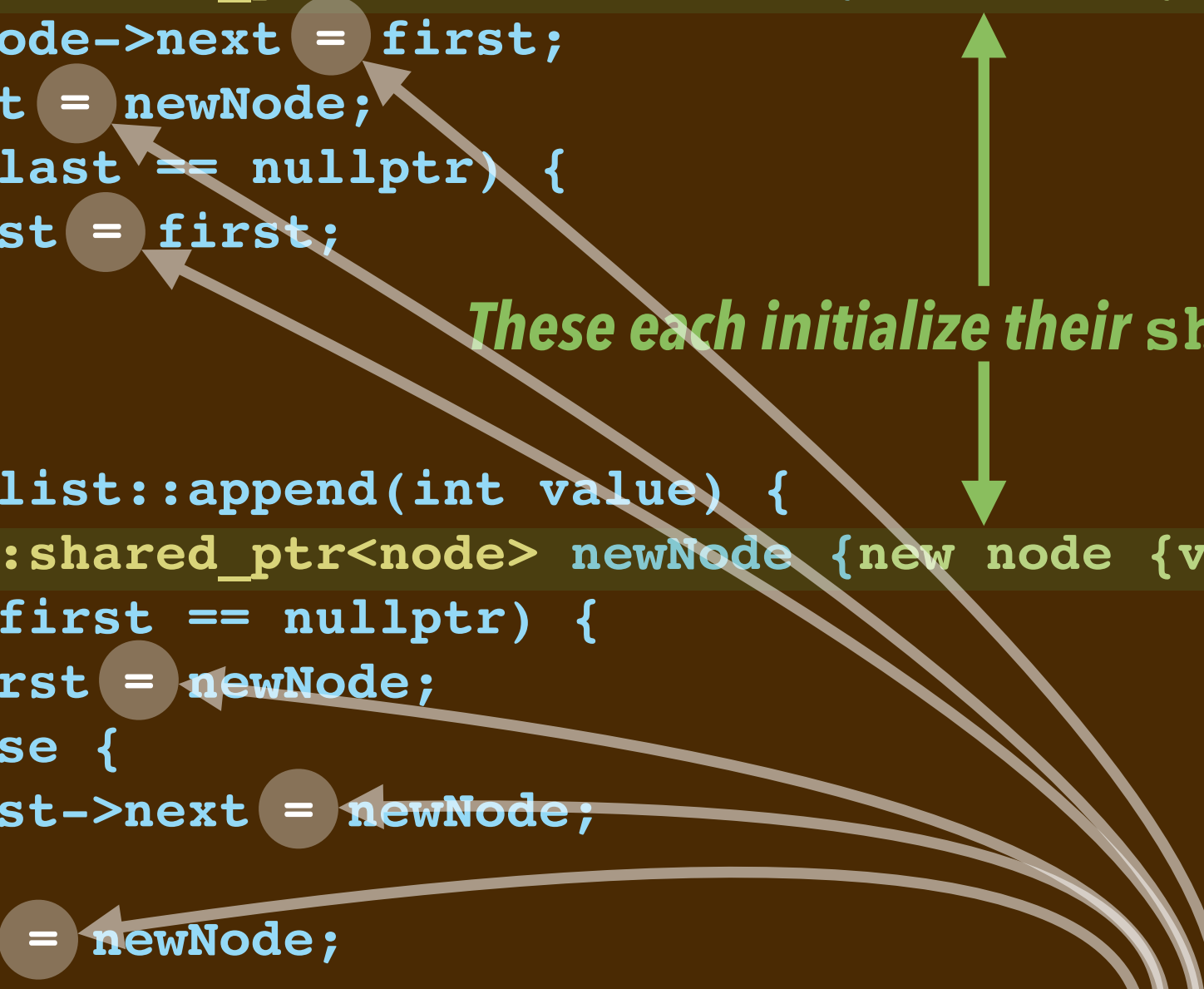
These each initialize their shared_ptr count to 1.



```
void llist::append(int value) {  
    std::shared_ptr<node> newNode {new node {value}};  
    if (first == nullptr) {  
        first = newNode;  
    } else {  
        last->next = newNode;  
    }  
    last = newNode;  
}
```

LINKED LIST SHARED_PTR SHARING

```
void llist::prepend(int value) {  
    std::shared_ptr<node> newNode {new node {value}};  
    newNode->next = first;  
    first = newNode;  
    if (last == nullptr) {  
        last = first;  
    }  
}
```



These each initialize their shared_ptr count to 1.

```
void llist::append(int value) {  
    std::shared_ptr<node> newNode {new node {value}};  
    if (first == nullptr) {  
        first = newNode;  
    } else {  
        last->next = newNode;  
    }  
    last = newNode;  
}
```

These copy assignments each increment their shared_ptr count.

LINKED LIST **SHARED_PTR** REMOVE METHOD

```
void llist::remove(int value) {
    std::shared_ptr<node> follow {nullptr};
    std::shared_ptr<node> current {first};
    while (current != nullptr && current->data != value) {
        follow = current;
        current = current->next;
    }
    if (current != nullptr) {
        if (follow == nullptr) {
            first = current->next;
            if (current->next == nullptr) {
                last = first;
            }
        } else {
            follow->next = current->next;
            if (current->next == nullptr) {
                last = follow;
            }
        }
    }
}
```

LINKED LIST **SHARED_PTR** REMOVE METHOD

```
void llist::remove(int value) {
    std::shared_ptr<node> follow {nullptr};
    std::shared_ptr<node> current {first};
    while (current != nullptr && current->data != value) {
        follow = current;
        current = current->next;
    }
    if (current != nullptr) {
        if (follow == nullptr) {
            first = current->next;
            if (current->next == nullptr) {
                last = first;
            }
        } else {
            follow->next = current->next;
            if (current->next == nullptr) {
                last = follow;
            }
        }
    }
}
```

**Unlinking current
decreases its `shared_ptr`
reference count.**

LINKED LIST **SHARED_PTR** REMOVE METHOD

```
void llist::remove(int value) {
    std::shared_ptr<node> follow {nullptr};
    std::shared_ptr<node> current {first};
    while (current != nullptr && current->data != value) {
        follow = current;
        current = current->next;
    }
    if (current != nullptr) {
        if (follow == nullptr) {
            first = current->next;
            if (current->next == nullptr) {
                last = first;
            }
        } else {
            follow->next = current->next;
            if (current->next == nullptr) {
                last = follow;
            }
        }
    }
}
```

Unlinking current decreases its shared_ptr reference count.

E.g. This copy assignment takes current's shared_ptr out of follow->next.

LINKED LIST **SHARED_PTR** REMOVE METHOD

```
void llist::remove(int value) {
    std::shared_ptr<node> follow {nullptr};
    std::shared_ptr<node> current {first};
    while (current != nullptr && current->data != value) {
        follow = current;
        current = current->next;
    }
    if (current != nullptr) {
        if (follow == nullptr) {
            first = current->next;
            if (current->next == nullptr) {
                last = first;
            }
        } else {
            follow->next = current->next;
            if (current->next == nullptr) {
                last = follow;
            }
        }
    }
}
```

Here `current` loses scope; removed node's reference count goes to 0 and is reclaimed.

NO DESTRUCTOR CODE NEEDED

```
class node {
    int data;
    std::shared_ptr<node> next;
    node(int value) : data {value}, next {nullptr} { }
    ~node(void) { }
};

class llist {
    std::shared_ptr<node> first;
    std::shared_ptr<node> last;
    llist(void) : first {nullptr}, last {nullptr} { }
    ~llist(void) { // NOTHING HERE!! }
    void prepend(int value);
    void append(int value);
    void remove(int value);
};
```

- ▶ When an **llist**'s storage is reclaimed, **first** and **last** are decremented.
 - This leads to a cascading series of automatic reclamations of **nodes**.

WHY IT WORKS: SINGLY LINKED LIST

STACK FRAME



HEAP MEMORY



count
1

count
1

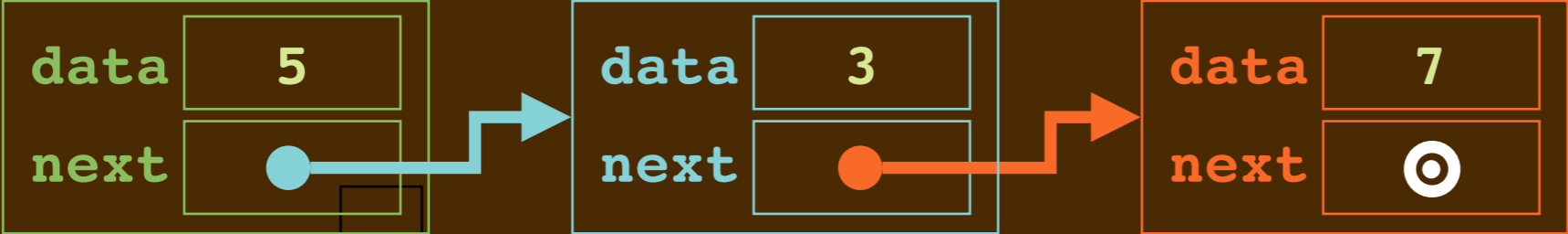
count
2

WHY IT WORKS: SINGLY LINKED LIST

STACK FRAME



HEAP MEMORY

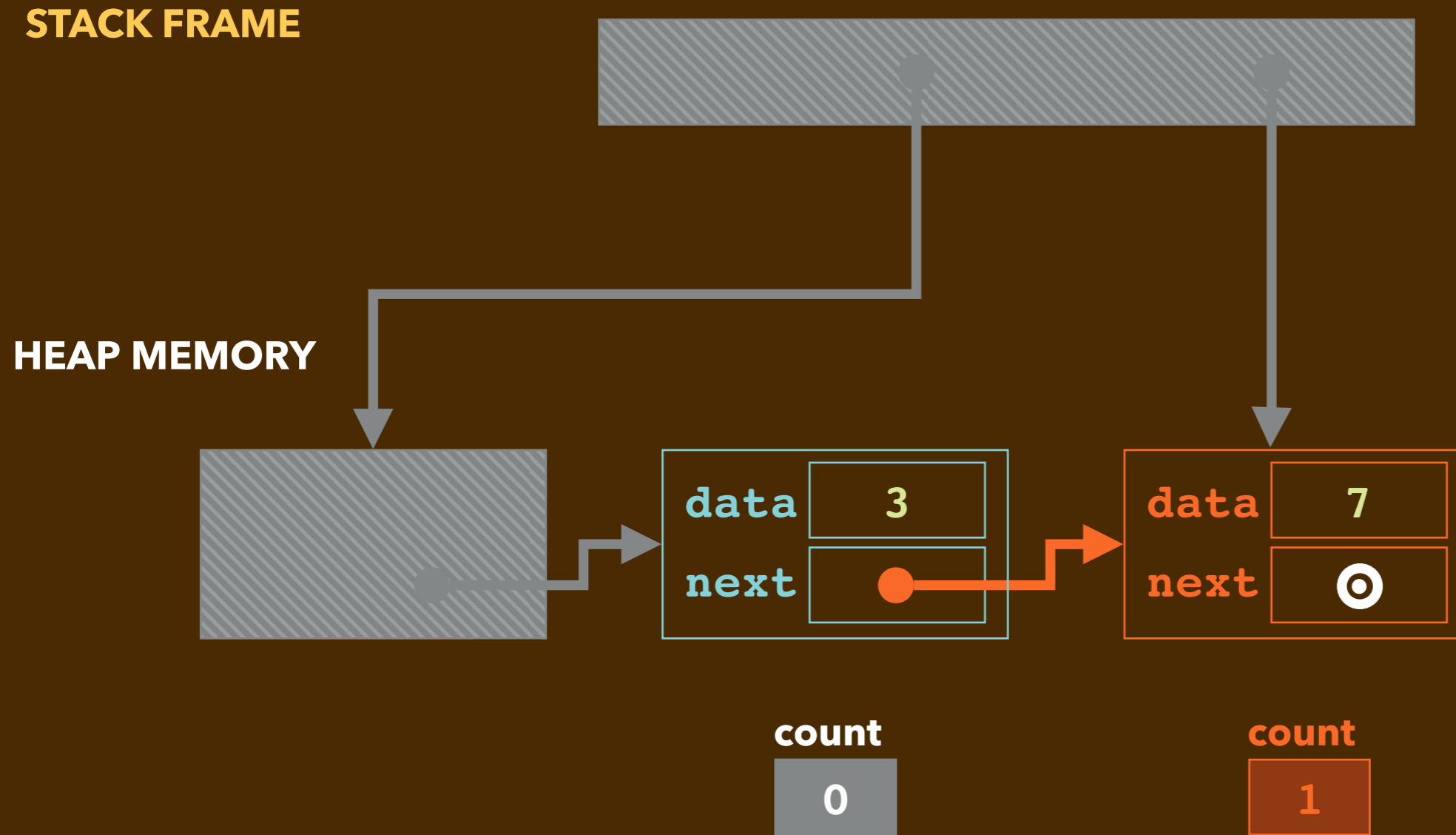


count
0

count
1

count
1

WHY IT WORKS: SINGLY LINKED LIST



WHY IT WORKS: SINGLY LINKED LIST

STACK FRAME



HEAP MEMORY



count

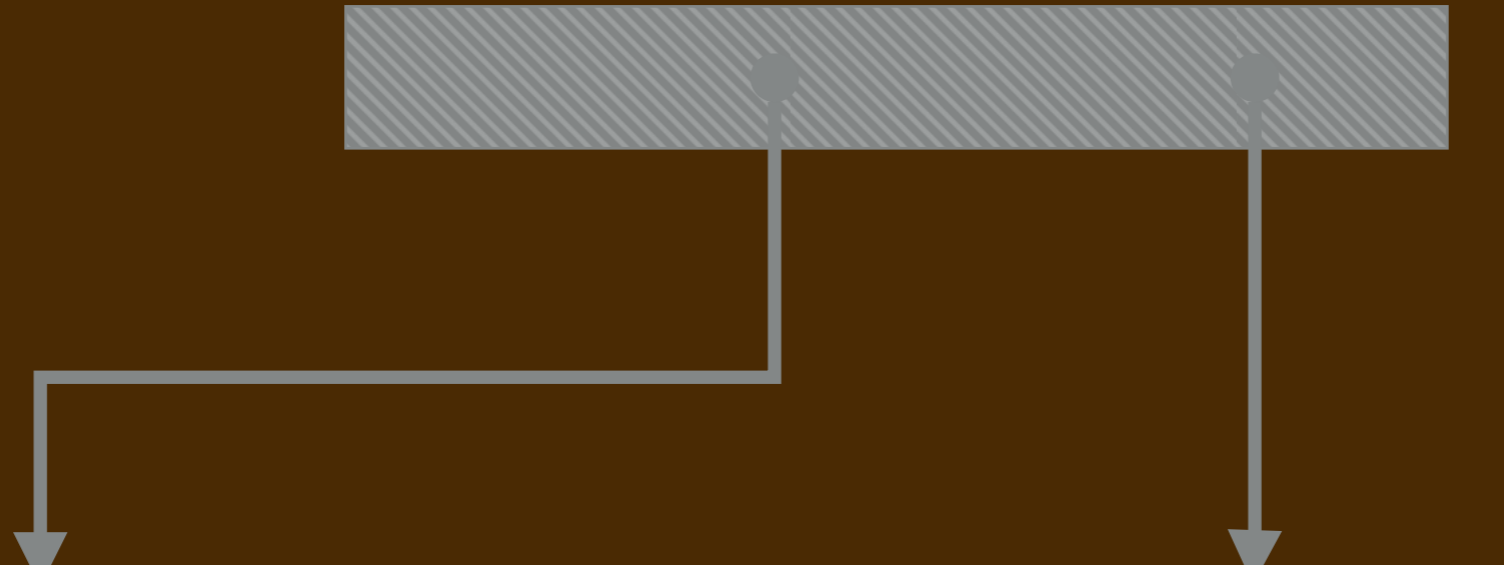
0

WHY IT WORKS: SINGLY LINKED LIST

STACK FRAME



HEAP MEMORY



A SHARED_PTR SINGLY LINKED LIST SUMMARY

- ▶ By using **shared_ptr**, every reference to a node is counted.
- ▶ When a new node is made, a **shared_ptr** is invented with a count of 1.
 - It has an underlying raw pointer obtained from **new**.
- ▶ When a relink happens:
 - A non-null reference's count decrements.
 - Another reference's count increments.
- ▶ When a reference count goes to 0:
 - The underlying raw pointer is **deleted**.
 - If non-null, its **next** reference's count is decremented.
- ▶ The *code never explicitly calls delete*.

A SHARED_PTR DOUBLY LINKED LIST

```
#include <memory>

class dnode {
public:
    int data;
    std::shared_ptr<dnode> next;
    std::shared_ptr<dnode> prev;
    dnode(int value) : data {value}, next {nullptr}, prev {nullptr} { }
};

class dbllist {
private:
    std::shared_ptr<dnode> first;
    std::shared_ptr<dnode> last;
public:
    dbllist(void) : first {nullptr}, last {nullptr} { }
    void append(int value) { // similar code as before ;
    void prepend(int value);
    void remove(int value);
}
```

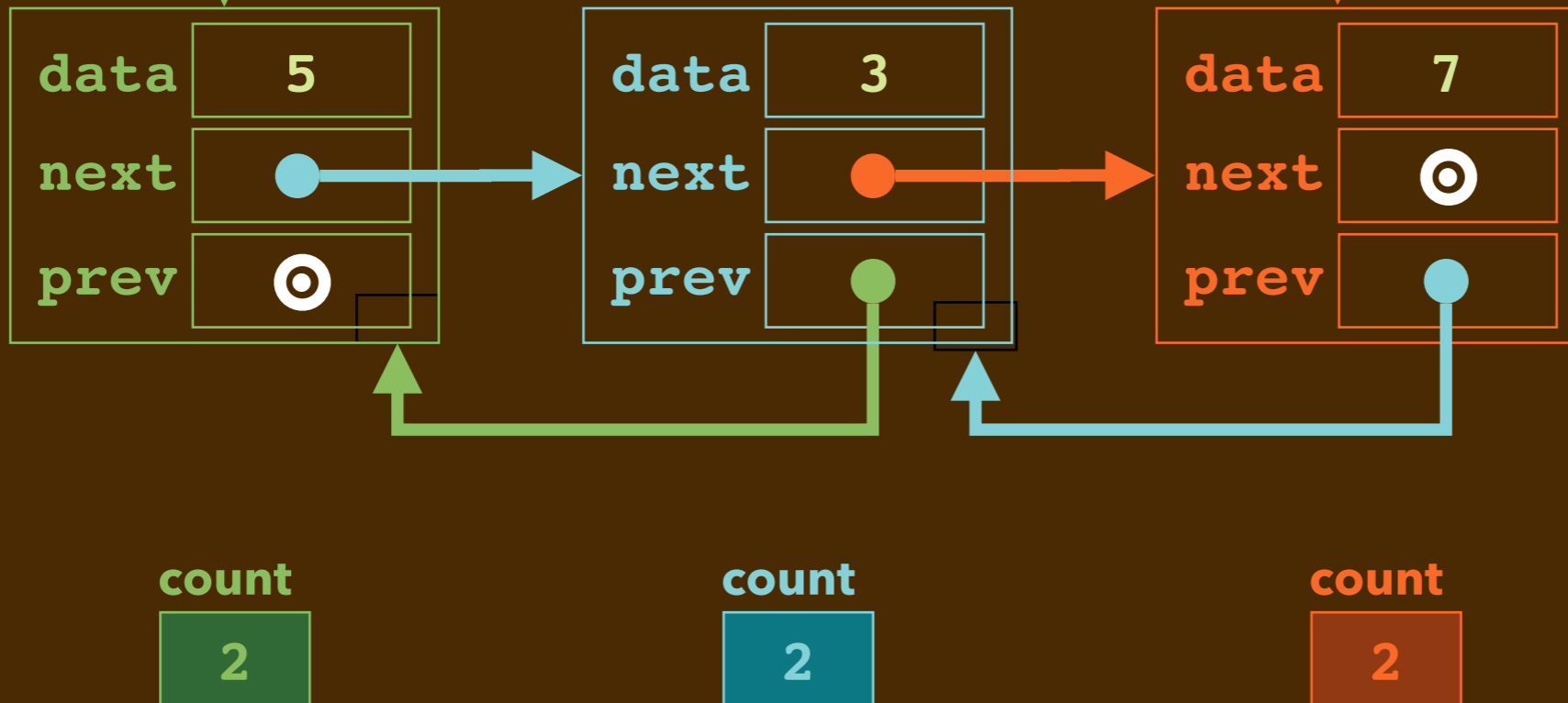
BUG: linked list nodes aren't reclaimed

WHY IT FAILS: DOUBLY LINKED LIST

STACK FRAME



HEAP MEMORY

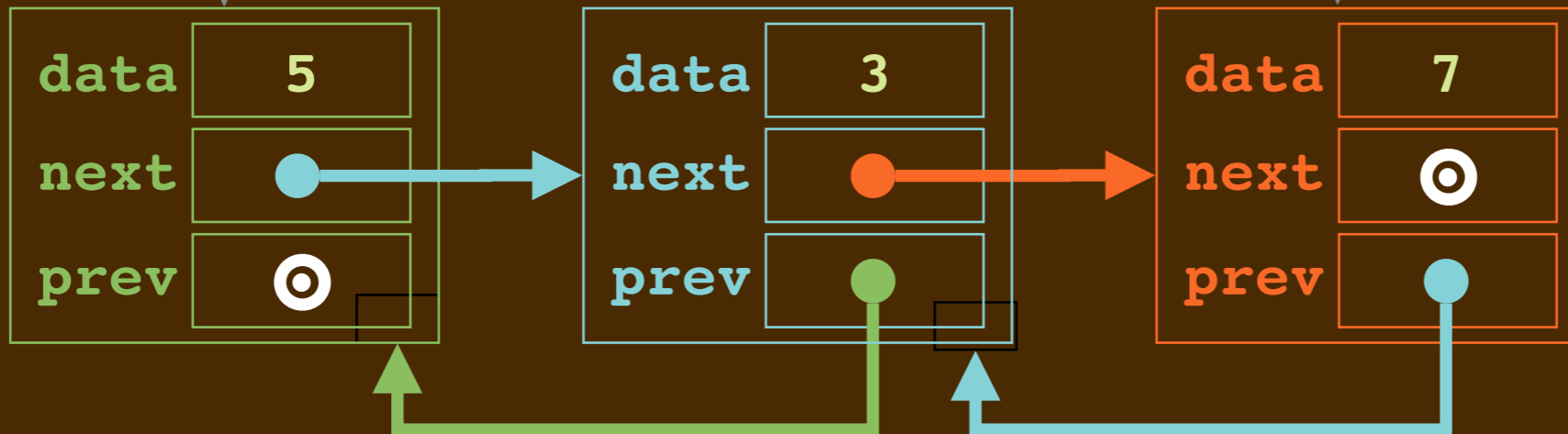


WHY IT FAILS: DOUBLY LINKED LIST

STACK FRAME



HEAP MEMORY



FIX #1: A DESTRUCTOR THAT UNLINKS PREV POINTERS

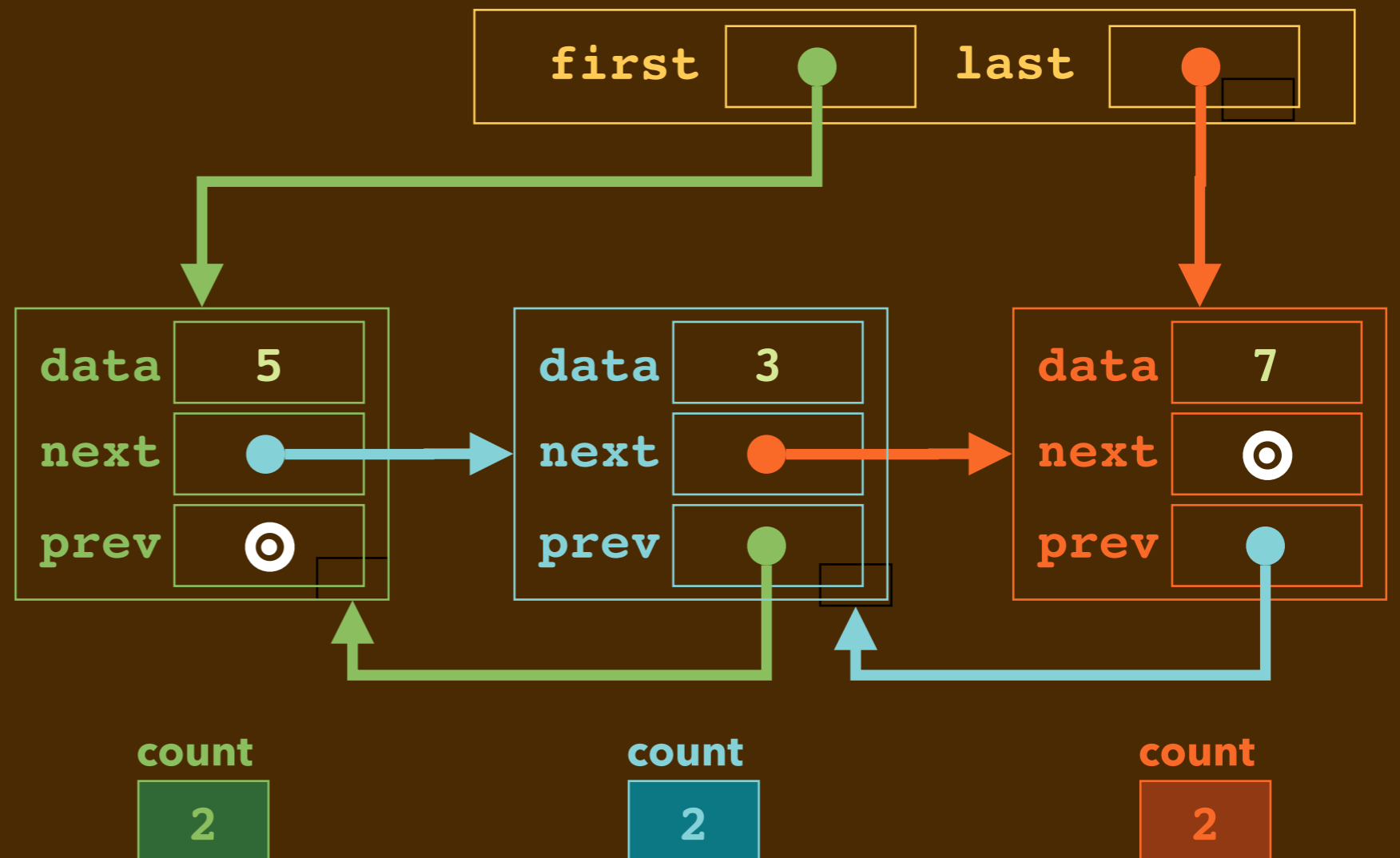
```
class dbllist {
private:
    std::shared_ptr<dnnode> first;
    std::shared_ptr<dnnode> last;
public:
    dbllist(void) : first {nullptr}, last {nullptr} { }
    ~dbllist(void); // next slide
    void append(int value) { // similar code as before ;
    void prepend(int value);
    void remove(int value);
}
```

FIX #1: A DESTRUCTOR THAT UNLINKS EACH PREV

```

dbllist::~~dbllist(void) {
    for (std::shared_ptr<dnode> current = first;
        current != nullptr;
        current = current->next) {
        current->prev = nullptr;
    }
}

```

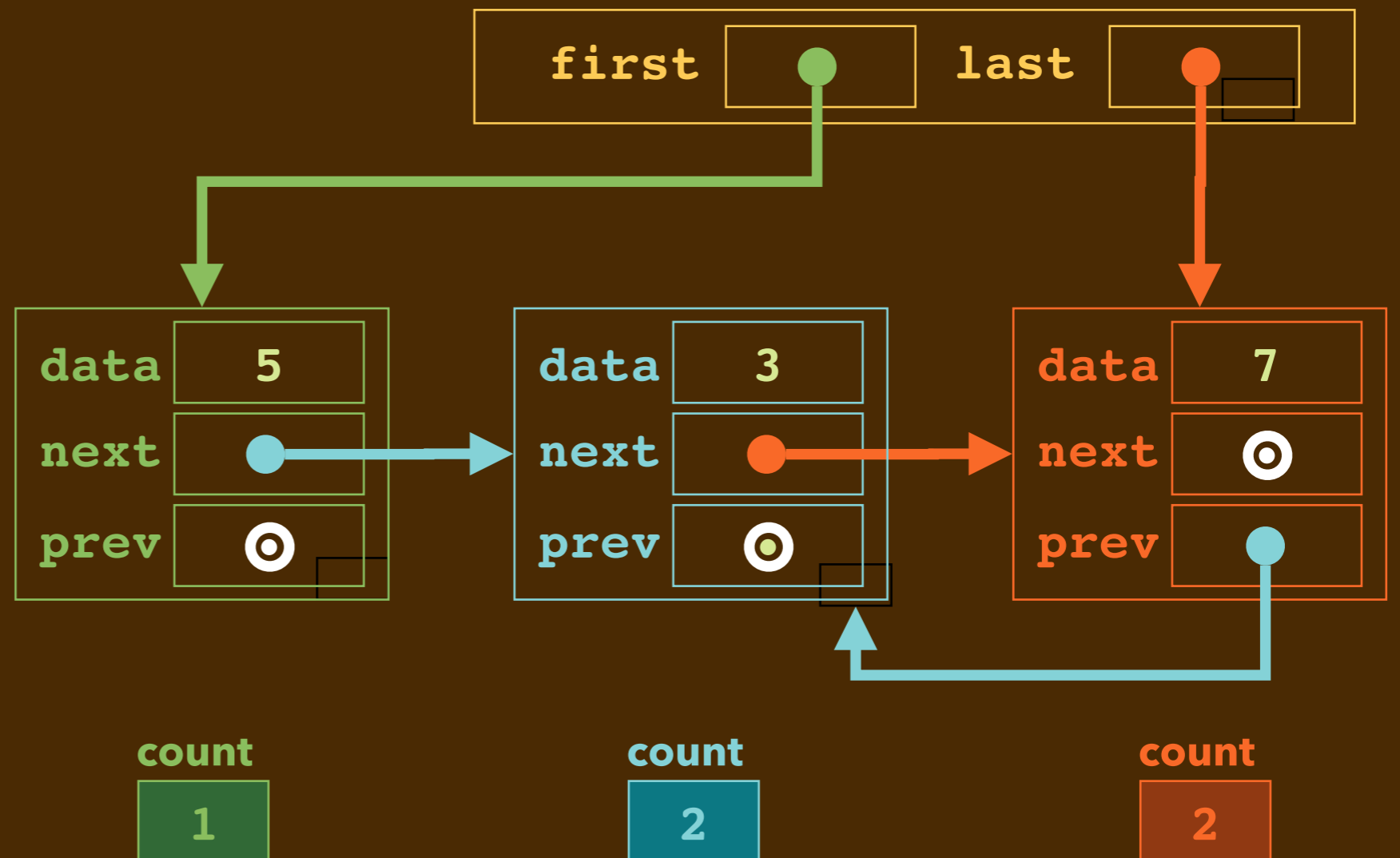


FIX #1: A DESTRUCTOR THAT UNLINKS EACH PREV

```

dbllist::~~dbllist(void) {
    for (std::shared_ptr<dnode> current = first;
        current != nullptr;
        current = current->next) {
        current->prev = nullptr;
    }
}

```

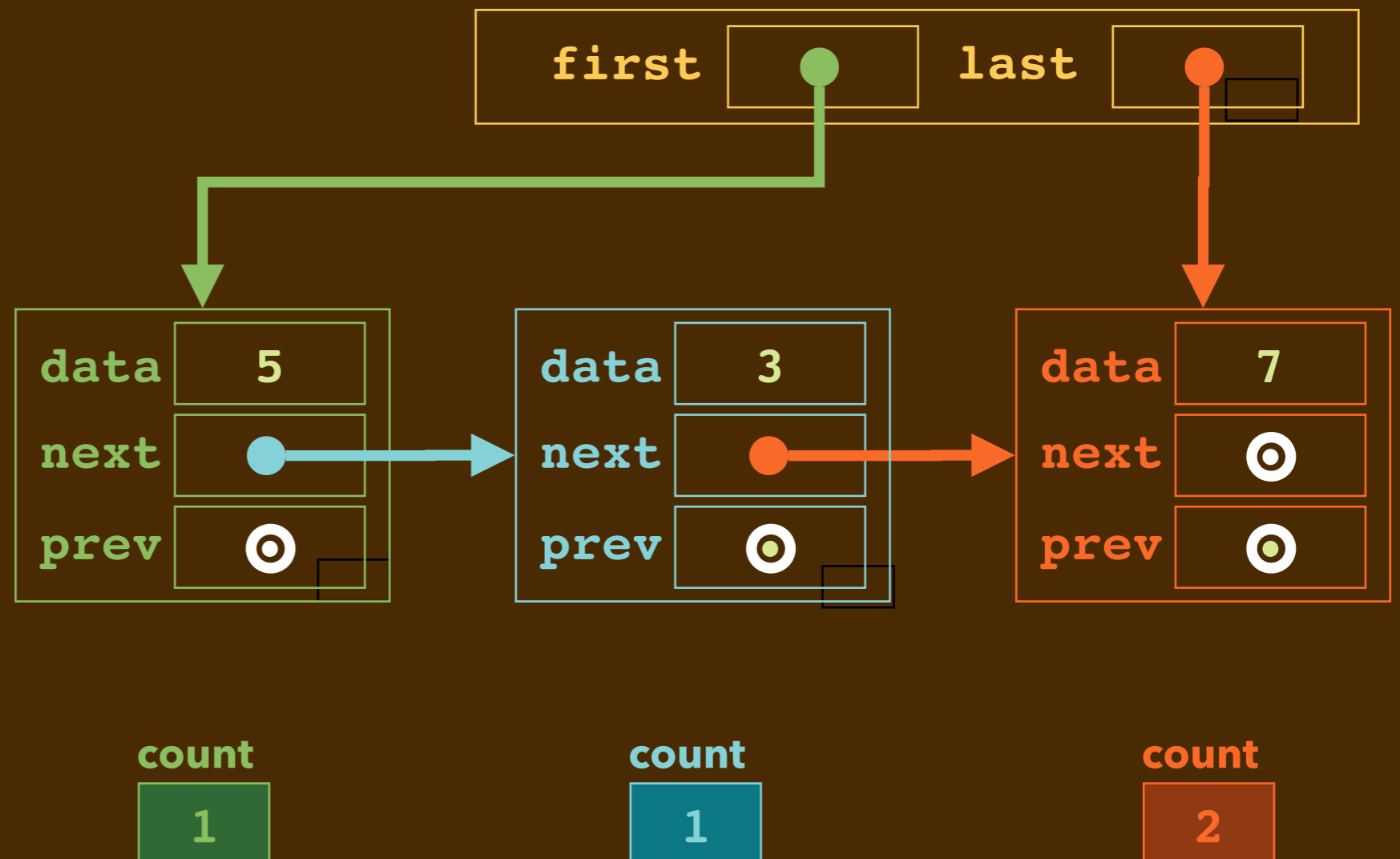


FIX #1: A DESTRUCTOR THAT UNLINKS EACH PREV

```

dbllist::~~dbllist(void) {
    for (std::shared_ptr<dnode> current = first;
        current != nullptr;
        current = current->next) {
        current->prev = nullptr;
    }
}

```

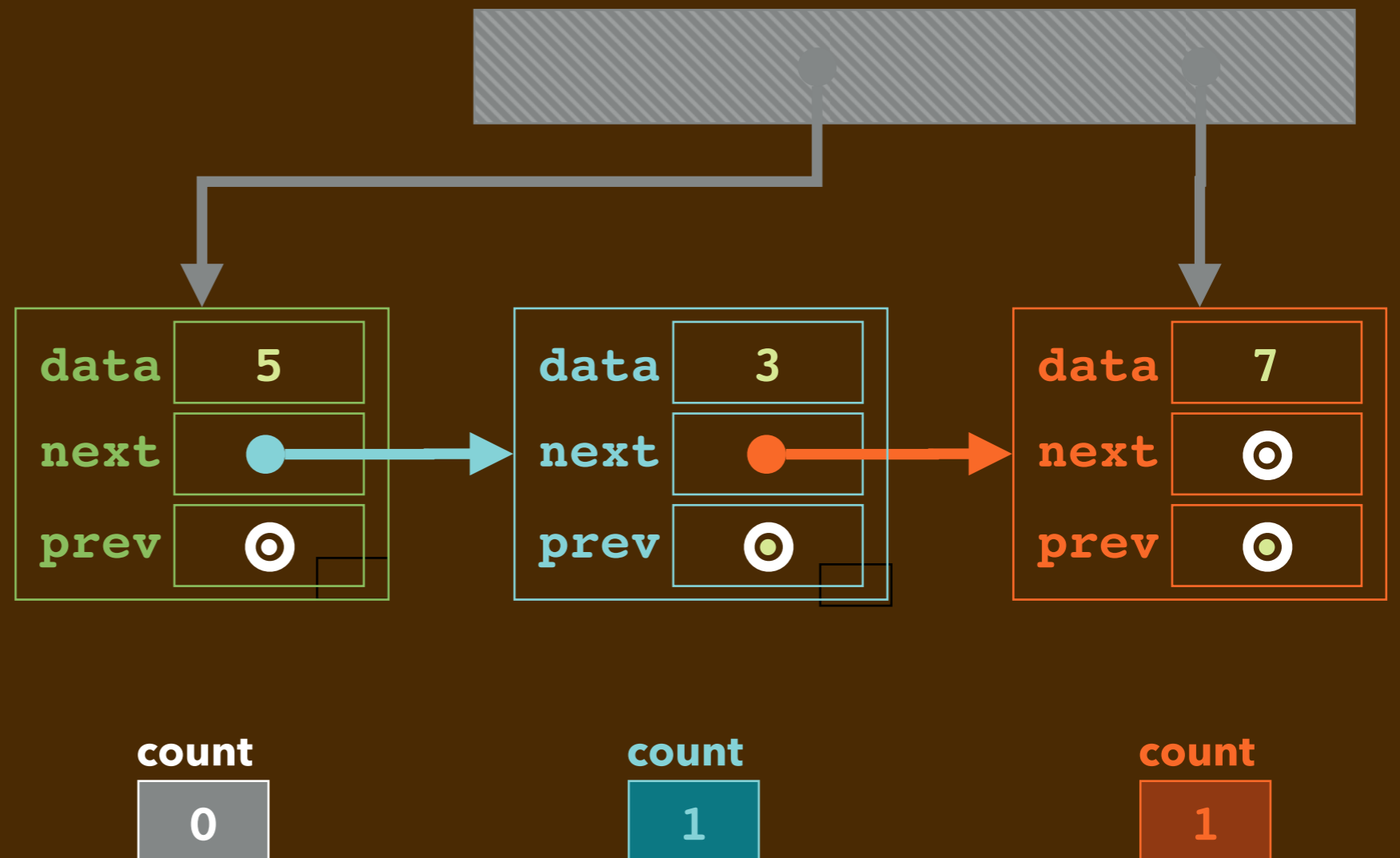


FIX #1: A DESTRUCTOR THAT UNLINKS EACH PREV

```

dbllist::~~dbllist(void) {
    for (std::shared_ptr<dnode> current = first;
        current != nullptr;
        current = current->next) {
        current->prev = nullptr;
    }
}

```

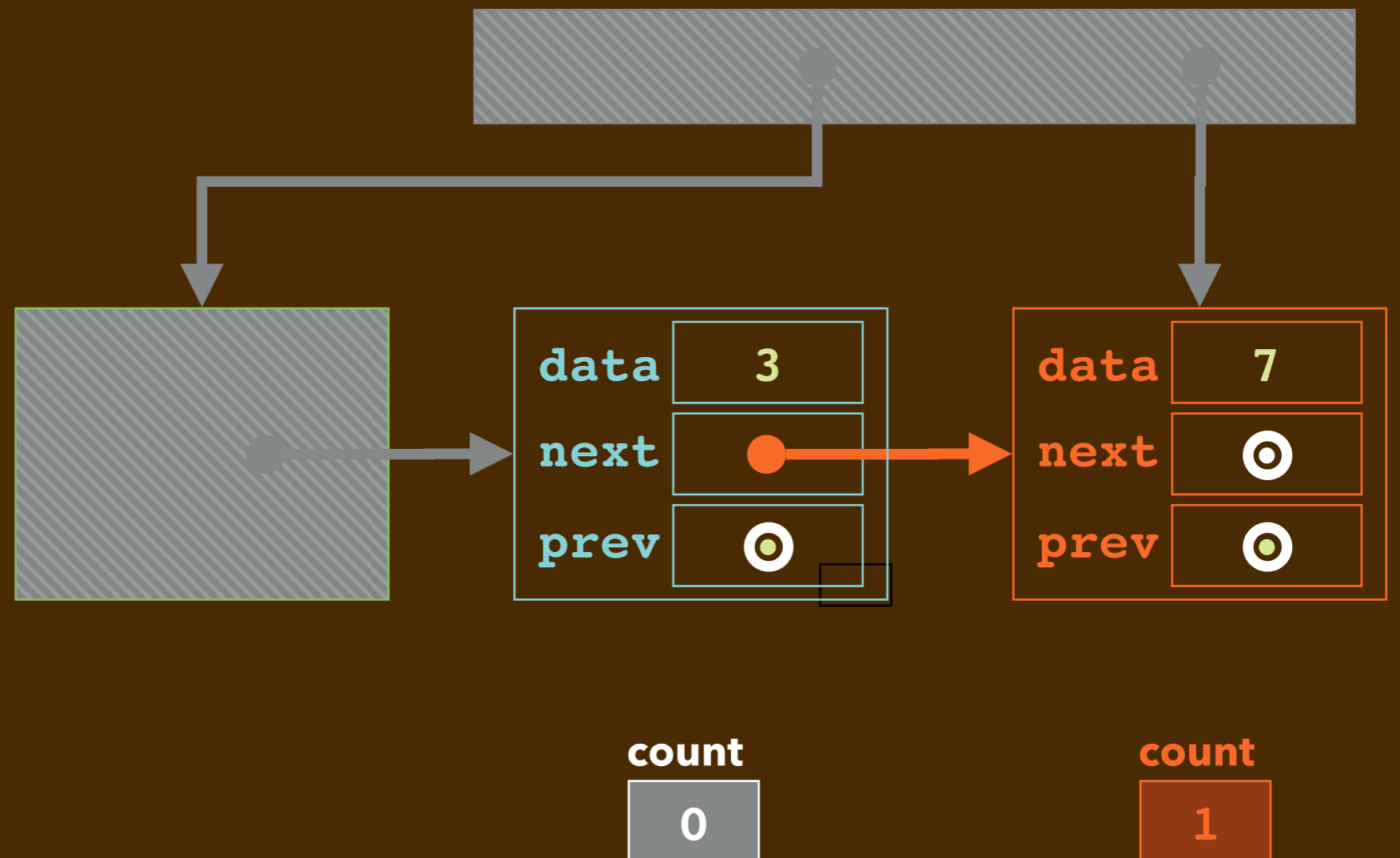


FIX #1: A DESTRUCTOR THAT UNLINKS EACH PREV

```

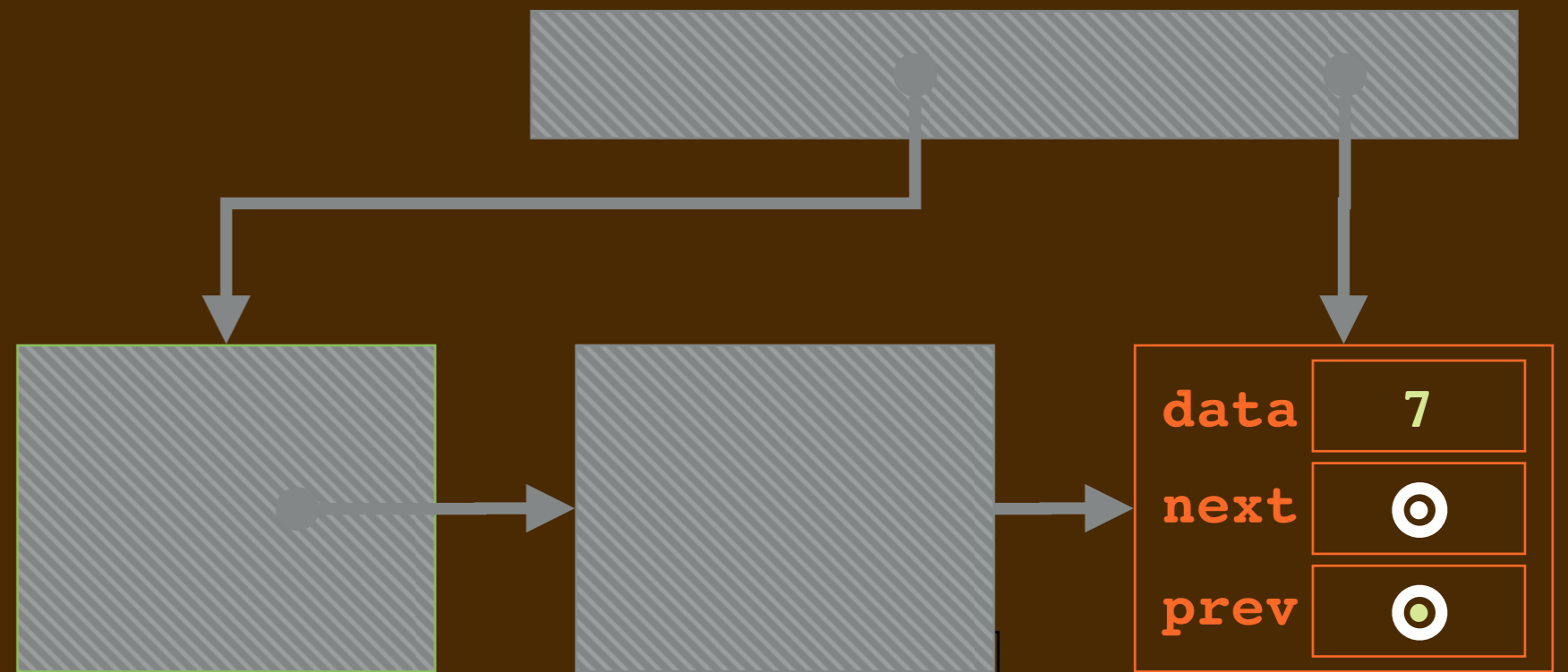
dbllist::~~dbllist(void) {
    for (std::shared_ptr<dnode> current = first;
        current != nullptr;
        current = current->next) {
        current->prev = nullptr;
    }
}

```



FIX #1: A DESTRUCTOR THAT UNLINKS EACH PREV

```
dbllist::~~dbllist(void) {  
    for (std::shared_ptr<dnode> current = first;  
         current != nullptr;  
         current = current->next) {  
        current->prev = nullptr;  
    }  
}
```

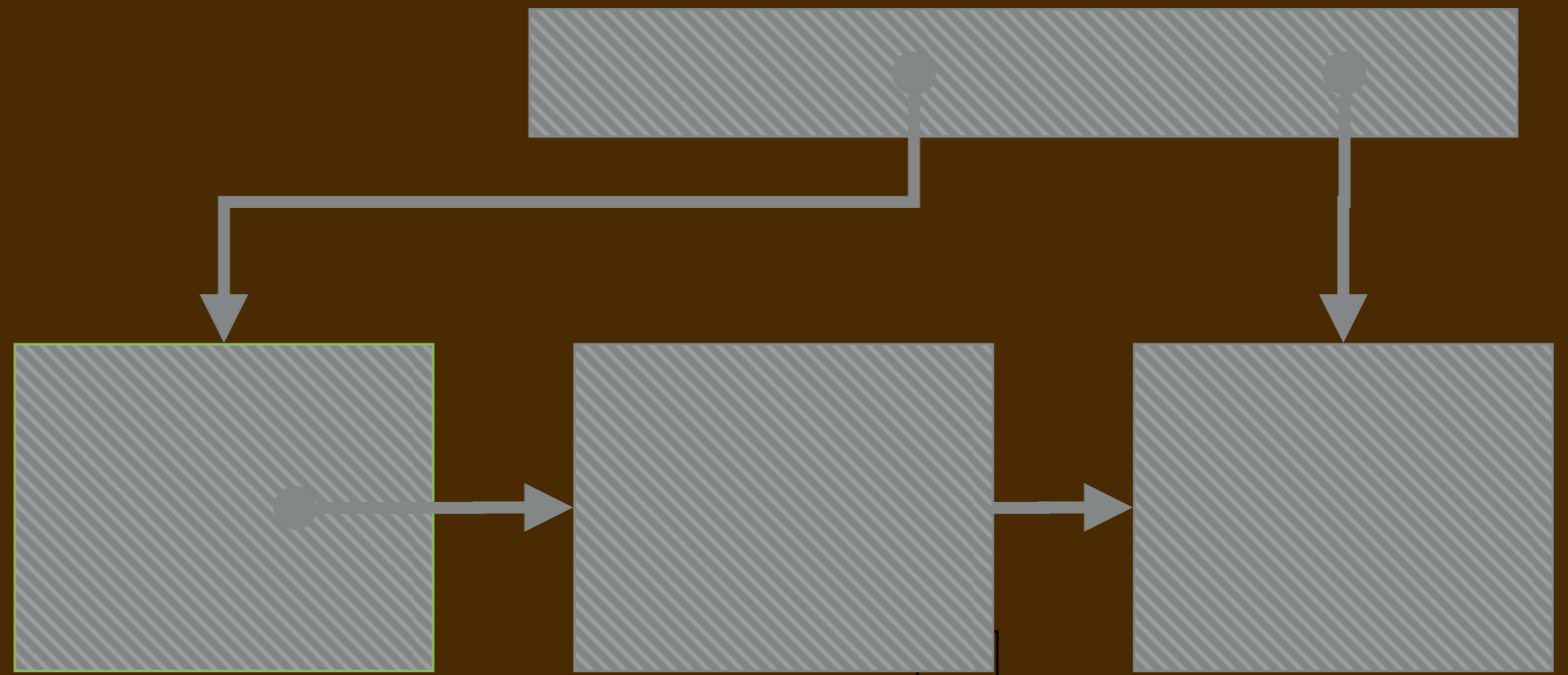


count

0

FIX #1: A DESTRUCTOR THAT UNLINKS EACH PREV

```
dbllist::~~dbllist(void) {  
    for (std::shared_ptr<dnode> current = first;  
         current != nullptr;  
         current = current->next) {  
        current->prev = nullptr;  
    }  
}
```

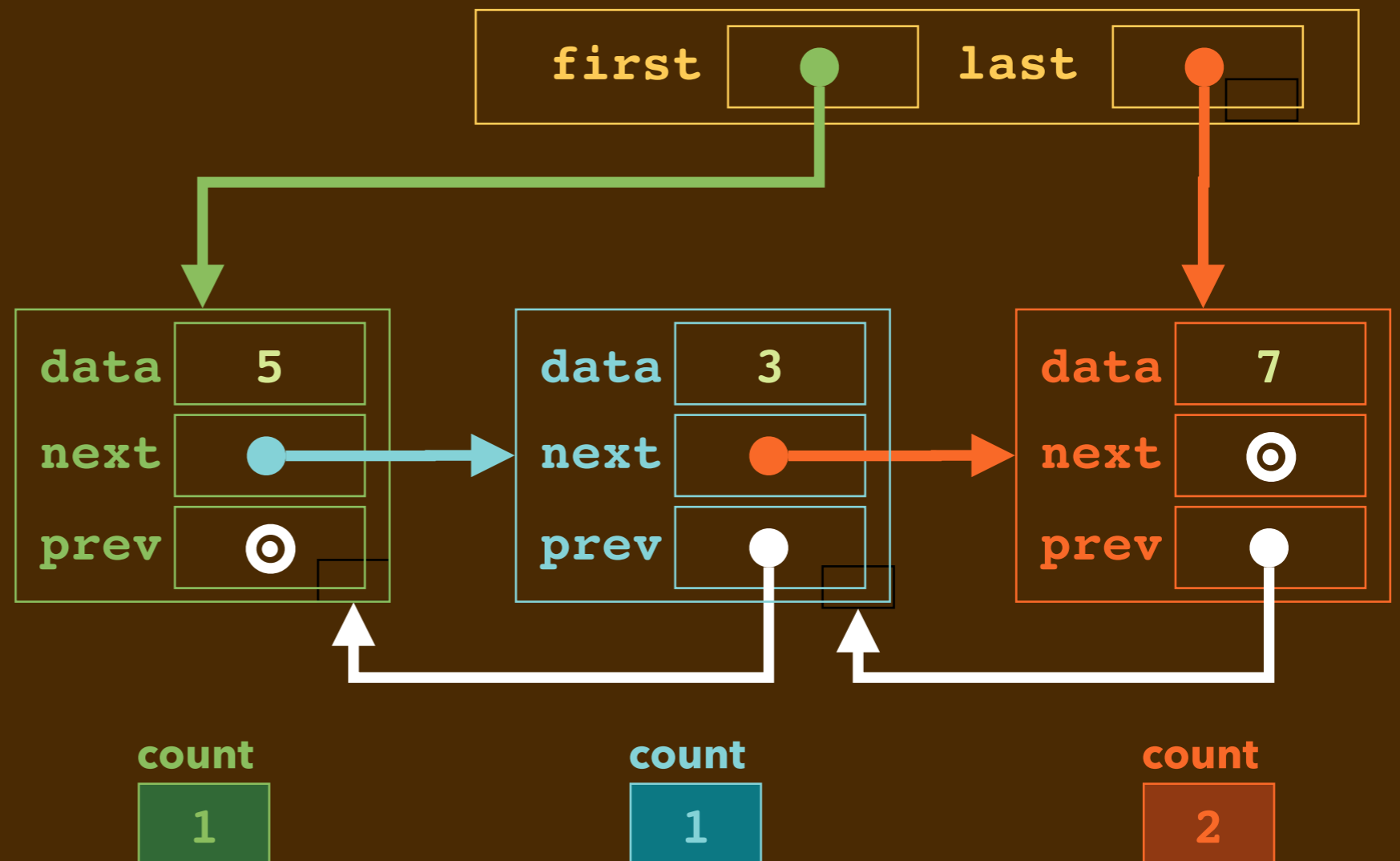


FIX #2: PREV POINTERS THAT DON'T COUNT

```

class dnode {
public:
    int data;
    std::shared_ptr<dnode> next;
    std::weak_ptr<dnode> prev;
    dnode(int value) : data {value}, next {nullptr}, prev {} {}
};

```

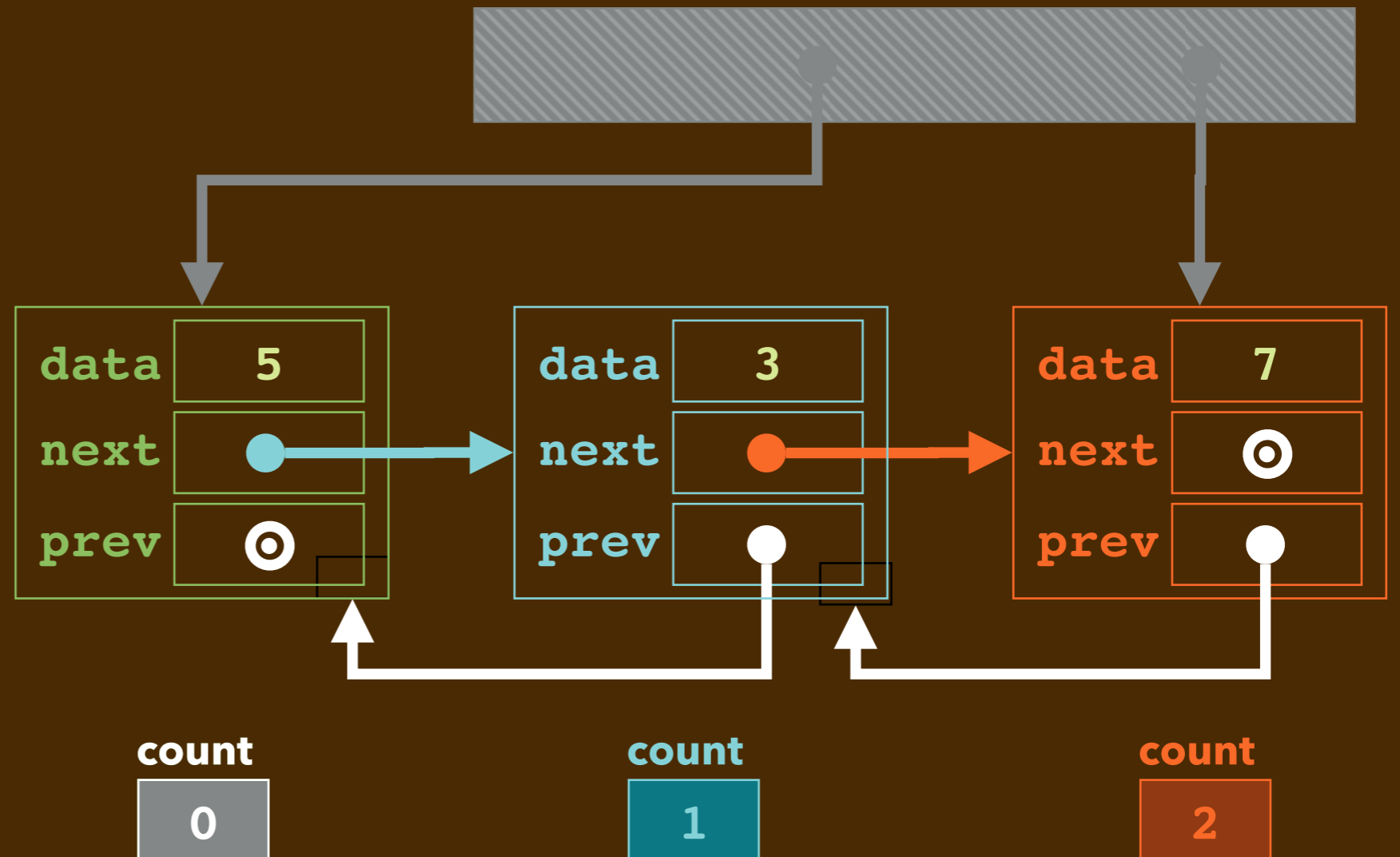


FIX #2: PREV POINTERS THAT DON'T COUNT

```

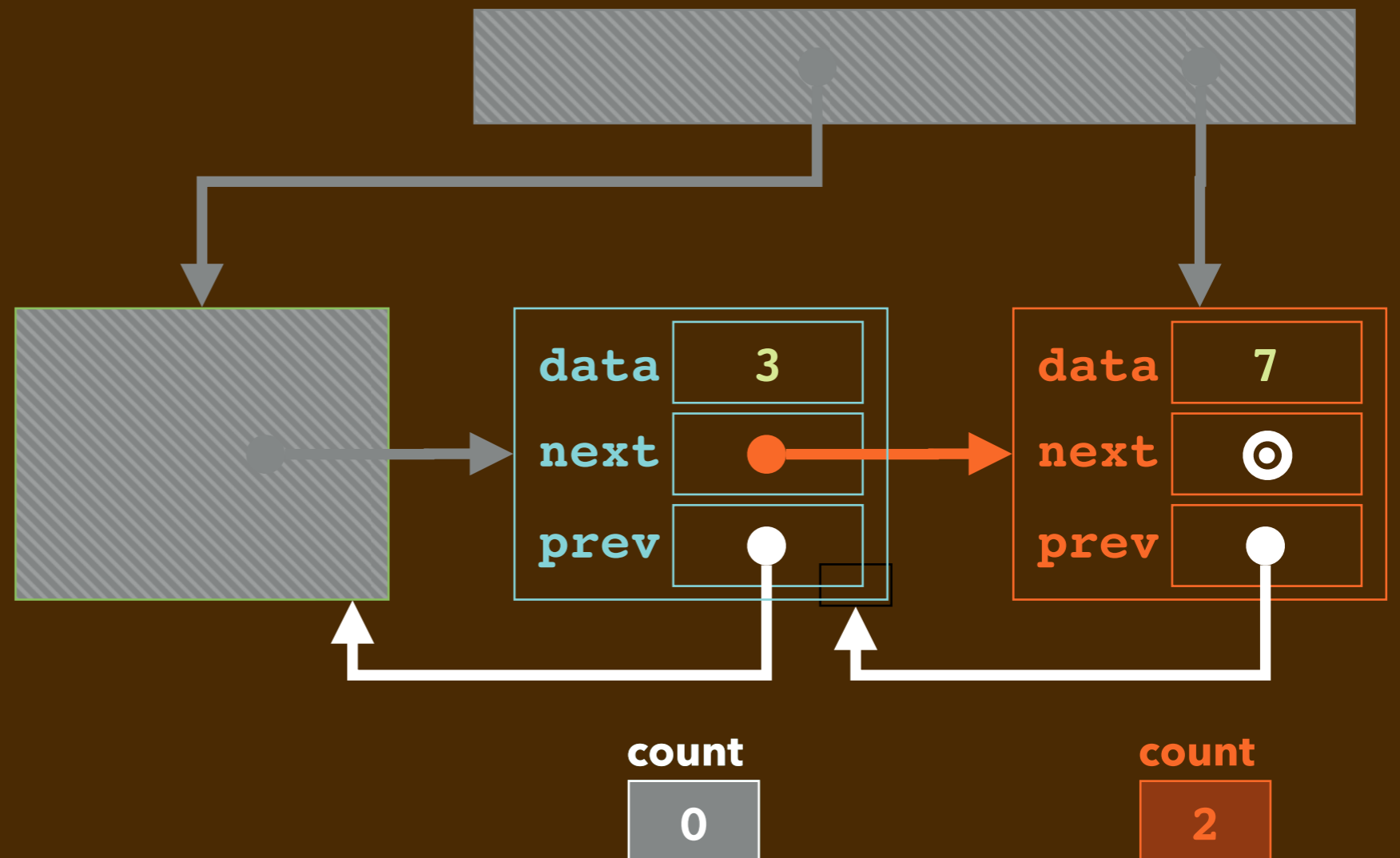
class dnode {
public:
    int data;
    std::shared_ptr<dnode> next;
    std::weak_ptr<dnode> prev;
    dnode(int value) : data {value}, next {nullptr}, prev {} { }
};

```



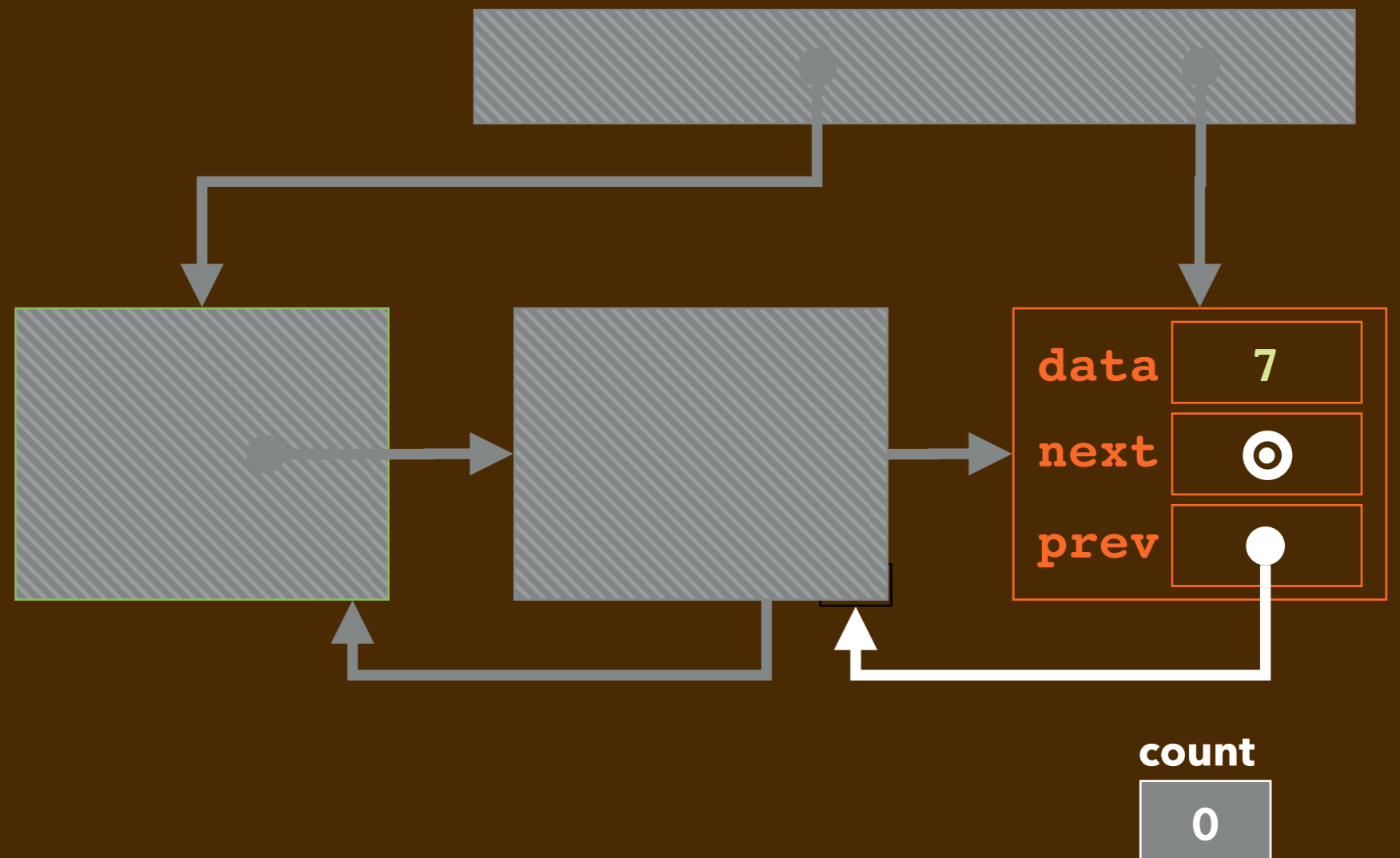
FIX #2: PREV POINTERS THAT DON'T COUNT

```
class dnode {  
public:  
    int data;  
    std::shared_ptr<dnode> next;  
    std::weak_ptr<dnode> prev;  
    dnode(int value) : data {value}, next {nullptr}, prev {} { }  
};
```



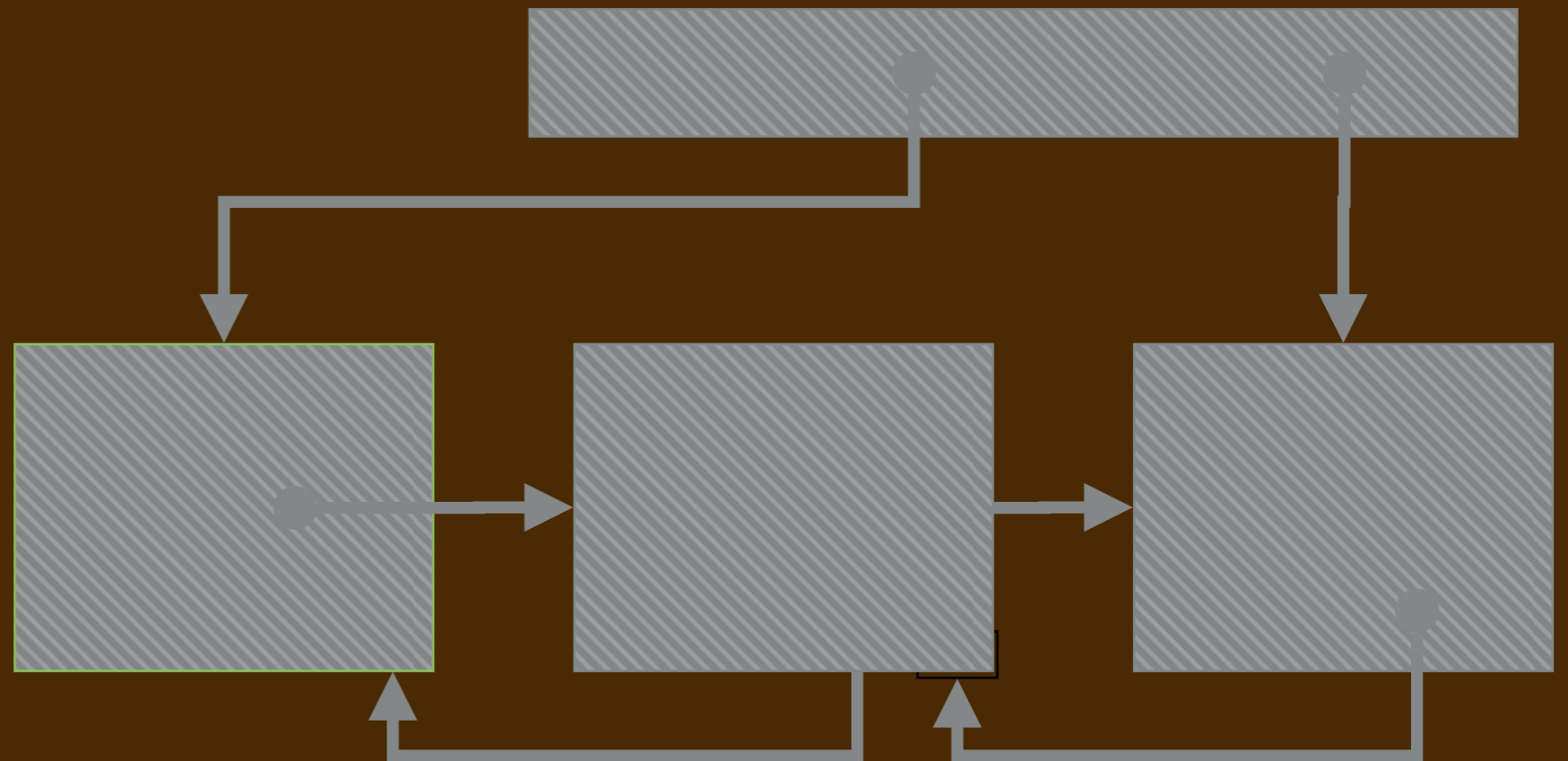
FIX #2: PREV POINTERS THAT DON'T COUNT

```
class dnode {  
public:  
    int data;  
    std::shared_ptr<dnode> next;  
    std::weak_ptr<dnode> prev;  
    dnode(int value) : data {value}, next {nullptr}, prev {} { }  
};
```



FIX #2: PREV POINTERS THAT DON'T COUNT

```
class dnode {  
public:  
    int data;  
    std::shared_ptr<dnode> next;  
    std::weak_ptr<dnode> prev;  
    dnode(int value) : data {value}, next {nullptr}, prev {} { }  
};
```



WORKING WITH WEAK_PTR IN REMOVE CODE

```
void remove(int value) {
    std::shared_ptr<dnode> current {first};
    while (current != nullptr && current->data != value) {
        current = current->next;
    }
    if (current != nullptr) {
        if (current == first) {
            first = current->next;
        } else {
            std::shared_ptr<dnode> prev {current->prev};
            prev->next = current->next;
        }
        if (current == last) {
            last = std::shared_ptr<dnode>{current->prev};
        } else {
            std::shared_ptr<dnode> next {current->next};
            next->prev = current->prev;
        }
    }
}
```

CHECK OUT MY SAMPLE CODE UNDER LECTURE 13-2

- ▶ I have four versions of linked lists that use **shared_ptr**:
 - **l1ist.cc**: what I just showed you with test code
 - **dbllist_*.cc**: three doubly-linked lists, each with test code
 - **_bad.cc**: because of circular paths in the data structure, *memory leak*
 - **_better.cc**: detaches **prev** links in `~dbllist()` to break cycles
 - **_best.cc**: uses **weak_ptr** for **prev** to break **shared_ptr** cycles