LAMBDAS: SMART POINTERS

LECTURE 13–2

JIM FIX, REED COLLEGE CS2–S20

TODAY'S PLAN

- We continue surveying the C++ *Standard Template Library* (or ***C++ STL***)
  - we'll finish looking at "lambda" expressions
  - we'll look at "smart pointers" in **`#include <memory>`**
    - **`std::shared_ptr<T>`**
    - **`std::weak_ptr<T>`**
    - **`std::unique_ptr<T>`**

C++ SYNTAX FOR A FUNCTION OBJECT

Here is the C++ syntax for an "anonymous" function object:

`[ ] (`*parameters*`) -> `*result-type*` { `*body*` }`

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones) -> int { return 10*tens+ones; } // two_digit
```

SYNTAX FOR A FUNCTION OBJECT

Here is the syntax for an anonymous function object:

[*captures*] (*parameters*) -> *result* { *body* }

- ***captures***: list of variables from the context that are used *by value* or *by reference* in the rule.
- ***parameters***: the function's parameters and their types
- ***result***: type of the returned result
- ***body***: code that computes the return result from the parameters

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones) -> int { return 10*tens+ones; } // two_digit
```

SYNTAX FOR A FUNCTION OBJECT

Here is the syntax for an anonymous function object:

```
[ captures ] ( parameters )  ->  result  {  body  }
```

- ***captures***: list of variables from the context that are used *by value* or *by reference* in the rule.
- ***parameters***: the function's parameters and their types
- ***result***: type of the returned result
- ***body***: code that computes the return result from the parameters

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones) -> int { return 10*tens+ones; } // two_digit
```

SYNTAX FOR A FUNCTION OBJECT

Here is the syntax for an anonymous function object:

[***captures*** **] (** ***parameters*** **) –> *result* { *body* }**

- ***captures***: list of variables from the context that are used *by value* or *by reference* in the rule.
- ***parameters***: the function's parameters and their types
- ***result***: type of the returned result
- ***body***: code that computes the return result from the parameters

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones) -> int { return 10*tens+ones; } // two_digit
```

SYNTAX FOR A FUNCTION OBJECT

Here is the syntax for an anonymous function object:

```
[ captures ] ( parameters )  ->  result  {  body  }
```

- ***captures***: list of variables from the context that are used *by value* or *by reference* in the rule.
- ***parameters***: the function's parameters and their types
- ***result***: type of the returned result
- ***body***: code that computes the return result from the parameters

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones) -> int { return 10*tens+ones; } // two_digit
```

SYNTAX FOR A FUNCTION OBJECT

Here is the syntax for an anonymous function object:

```
[ captures ] ( parameters )  ->  result  {  body  }
```

- ***captures***: list of variables from the context that are used *by value* or *by reference* in the rule.
- ***parameters***: the function's parameters and their types
- ***result***: type of the returned result
- ***body***: code that computes the return result from the parameters

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones) -> int { return 10*tens+ones; } // two_digit
```

NOTE: can span multiple lines.

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -
                                     std::cout << n << s
                                     std::cout << n << s
                                     std::cout << n << s
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

BTW, this outputs:

```
1
11
5
54321
54321
54321
```

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

NOTICE THE TYPES

```
std::function<bool(int)> isEven = [](int i) -> bool
                           { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                           { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                          { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                             std::cout << n << std::endl;
                             std::cout << n << std::endl;
                             std::cout << n << std::endl;
                           };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

NOTICE THE TYPES

At the top of this code:

```
#include <functional>
```

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

SYNTAX FOR A FUNCTION OBJECT'S TYPE

Here is the syntax for an anonymous function object's type:

`std::function<` ***result-type*** `(` ***types-of-parameters*** `)>`

- ***types-of-parameters***: the function's parameter types
- ***result-type***: type of the returned result
- **Example:**

```
std::function<void(bool,std::string)> maybePrint =
  [](bool yes, std::string message) -> void {
    if (yes) {
      std::cout << message << std::endl;
    };
std::function<int(int,int)> two_digit =
  [](int tens_digit, int ones_digit) -> {
    return tens_digit*10 + ones_digit;
  }
```

CAPTURE BY VALUE

- Lambdas are defined *within the context* of executable code.
  - stack variables are available, stack objects are available
  - pointers to heap objects are available
- Can *indicate that these things can be accessed* by the function object.

Example:

```
int tens_digit = 5;
std::function<int(int)> make_fifty_something =
  [tens_digit](int ones_digit) -> int {
     return tens_digit * 10 + ones_digit;
  };
std::cout << make_fifty_something(8) << std::endl;
std::cout << make_fifty_something(7) << std::endl;
```

CAPTURE BY VALUE

- Lambdas are defined *within the context* of executable code.
  - stack variables are available, stack objects are available
  - pointers to heap objects are available
- Can *indicate that these things can be accessed* by the function object.

Example:

```
int tens_digit = 5;
std::function<int(int)> make_fifty_something =
  [tens_digit](int ones_digit) -> int {
     return tens_digit * 10 + ones_digit;
  };
std::cout << make_fifty_something(8) << std::endl;
std::cout << make_fifty_something(7) << std::endl;
```

This outputs:

```
58
57
```

CAPTURE BY VALUE (CONT'D)

- NOTE: the variable is copied *by value*.
  - ➡ Subsequent changes aren't reflected because of this kind of capture.

Example:

```
int tens_digit = 5;
std::function<int(int)> make_fifty_something =
  [tens_digit](int ones_digit) -> int {
     return tens_digit * 10 + ones_digit;
  };
tens_digit--;
std::cout << make_fifty_something(8) << std::endl;
tens_digit--;
std::cout << make_fifty_something(7) << std::endl;
```

CAPTURE BY VALUE (CONT'D)

- NOTE: the variable is copied *by value*.
  - ➡ Subsequent changes aren't reflected because of this kind of capture.

Example:

```
int tens_digit = 5;
std::function<int(int)> make_fifty_something =
  [tens_digit](int ones_digit) -> int {
     return tens_digit * 10 + ones_digit;
  };
tens_digit--;
std::cout << make_fifty_something(8) << std::endl;
tens_digit--;
std::cout << make_fifty_something(7) << std::endl;
```

Also outputs:

```
58
57
```

CAPTURE BY REFERENCE

- Alternatively, you can *indicate that variables' values can be tracked and altered* by the function object.
- You indicate this with the by-reference annotation **&**.

Example:

```
int tens_varies = 5;
std::function<int(int)> make_umpty_something =
  [&tens_varies](int ones_digit) -> int {
     return tens_varies * 10 + ones_digit;
  };
tens_varies--;
std::cout << make_umpty_something(8) << std::endl;
tens_varies--;
std::cout << make_umpty_something(7) << std::endl;
```

CAPTURE BY REFERENCE

- Alternatively, you can *indicate that variables' values can be tracked and altered* by the function object.
- You indicate this with the by-reference annotation **&**.

Example:

```
int tens_varies = 5;
std::function<int(int)> make_umpty_something =
  [&tens_varies](int ones_digit) -> int {
     return tens_varies * 10 + ones_digit;
  };
tens_varies--;
std::cout << make_umpty_something(8) << std::endl;
tens_varies--;
std::cout << make_umpty_something(7) << std::endl;
```

Outputs:

```
48
37
```

HERE ARE SOME USES OF REFERENCE

```
int count = 0;
std::function<void(void)> increment =
      [&count](void) -> void { count++; }
int x = 10;
int y = 11;
std::function<void(void)> increment =
      [&x,&y](void) -> void {
  int tmp = x;
  x = y;
  y = tmp;
};
increment();
std::cout << count << " " << x << " " << y << std::endl;
increment();
swap();
std::cout << count << " " << x << " " << y << std::endl;
increment();
swap();
std::cout << count << " " << x << " " << y << std::endl;
```

HERE ARE SOME USES OF REFERENCE

```
int count = 0;
std::function<void(void)> increment =
      [&count](void) -> void { count++; }
int x = 10;
int y = 11;
std::function<void(void)> increment =
      [&x,&y](void) -> void {
  int tmp = x;
  x = y;
  y = tmp;
};
increment();
std::cout << count << " " << x << " " << y << std::endl;
increment();
swap();
std::cout << count << " " << x << " " << y << std::endl;
increment();
swap();
std::cout << count << " " << x << " " << y << std::endl;
```

This outputs:

```
1 10 11
2 11 10
3 10 11
```

HERE ARE SOME USES OF REFERENCE (CONT'D)

```
std::vector<int> v = {12, 8, 17};
std::function<void(void)> output =
  [&v](void) -> void {
    for (int e : v) {
      std::cout << e << " ";
    }
    std::cout << std::endl;
  };
std::function<void(int,int)> change =
   [&v](int index, int value) -> void { v.at(index) = value; };

output();
change(2,37);
output();
change(0,32);
output();
change(1,3);
output();
```

HERE ARE SOME USES OF REFERENCE (CONT'D)

```
std::vector<int> v = {12, 8, 17};
std::function<void(void)> output =
  [&v](void) -> void {
    for (int e : v) {
      std::cout << e << " ";
    }
    std::cout << std::endl;
  };
std::function<void(int,int)> change =
   [&v](int index, int value) -> void { v.at(index) = value; };

output();
change(2,37);
output();
change(0,32);
output();
change(1,3);
output();
```

HERE ARE SOME USES OF REFERENCE (CONT'D)

```
std::vector<int> v = {12, 8, 17};
std::function<void(void)> output =
  [&v](void) -> void {
    for (int e : v) {
      std::cout << e << " ";
    }
    std::cout << std::endl;
  };
std::function<void(int,int)> change =
   [&v](int index, int value) -> void { v.at(index) = value; };

output();
change(2,37);
output();
change(0,32);
output();
change(1,3);
output();
```

HERE ARE SOME USES OF REFERENCE (CONT'D)

```
std::vector<int> v = {12, 8, 17};
std::function<void(void)> output =
  [&v](void) -> void {
    for (int e : v) {
      std::cout << e << " ";
    }
    std::cout << std::endl;
  };
std::function<void(int,int)> change =
   [&v](int index, int value) -> void { v.at(index) = value; };

output();
change(2,37);
output();
change(0,32);
output();
change(1,3);
output();
```

This outputs:

```
12 8 17
12 8 37
32 8 37
32 3 37
```

USE WITHIN ALGORITHMS PACKAGE

```
std::vector<int> v {0,1,4,9,16,25,36,49,64};
int sum = 0;
std::for_each(v.begin(),v.end(),
              [&sum](int e) -> void { sum +=e; } );
std::cout << "sum = " << sum << std::endl;
// outputs sum = 204
```

CAPTURE LIST

RECALL the syntax for an anonymous function object:

`[` ***capture-list*** `]` `(` ***parameters*** `)` `->` ***result*** `{` ***rule*** `}`

- ***capture-list***: list of variables from the context that are used in the rule
- Can be used *by value* or *by reference*:
  - If you want the variable's value to be copied use no annotation
  - If you want the stack object/variable to be referenced, changed, use **&**

MUTABLE KEYWORD

You can also make variables that are captured *by value* changeable:

`[` ***capture-list*** `](` ***parameters*** `)` **`mutable ->`** ***result*** `{` ***body*** `}`

- The mutable lets the body change the new variables that are copied
- This demonstrates its effect:

```
int startAt = 100;
std::function<int(void)> dbl =
  [startAt](void) mutable -> int {
     startAt *= 2; return startAt;
  };
std::cout << dbl() << " " << startAt << std::endl;
std::cout << dbl() << " " << startAt << std::endl;
std::cout << dbl() << " " << startAt << std::endl;
```

MUTABLE KEYWORD

You can also make variables that are captured *by value* changeable:

`[` ***capture-list*** `](` ***parameters*** `)` **`mutable ->`** ***result*** `{` ***body*** `}`

- The mutable lets the body change the new variables that are copies
- This demonstrates its effect:

```
int startAt = 100;
std::function<int(void)> dbl =
  [startAt](void) mutable -> int {
     startAt *= 2; return startAt;
  };
std::cout << dbl() << " " << startAt << std::endl;
std::cout << dbl() << " " << startAt << std::endl;
std::cout << dbl() << " " << startAt << std::endl;
```

This outputs:

```
200 100
400 100
800 100
```

- The captured copy of the variable makes the lambda (internally) *stateful*.

SUMMARY OF C++ LAMBDA EXPRESSIONS

- C++ allows you to concisely express "function objects".
  - ➡ They are essentially one-time class instances with **`operator()`** defined.
- They are called *lambdas*
  - ➡ From the programming language **Lisp**: **`(lambda (n) (+ n 1))`**
  - ➡ Lisp's John McCarthy took them from Alonzo Church's "lambda calculus"
- Because C++ has a complex object memory model, must specify *captures*
  - ➡ overloads **`&`** syntax and uses keyword **`mutable`** to specify behavior
- Useful for many components defined in the **`algorithm`** STL

SOME WAYS OF HAVING POINTER-BASED CODE BREAK

- You use **&** on a stack variable/object in a function and that pointer gets exported outside a function. You try to access the component referenced by that pointer outside that function.
- You forget to initialize a pointer component. You access that component.
- You try to access a component referenced by a null pointer.
- Two data structures share a pointer, one calls a delete on it. You try to access it through the other data structure. Or maybe you try to delete again it there.
- You don't call delete on a pointer to a component no longer used by a data structure.

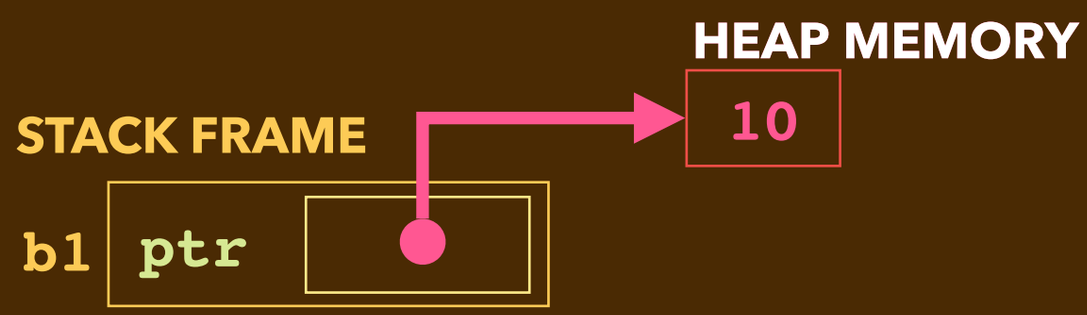
SOME WAYS OF HAVING POINTER-BASED CODE BREAK

- You use **&** on a stack variable/object in a function and that pointer gets exported outside a function. You try to access the component referenced by that pointer outside that function.
- You forget to initialize a pointer component. You access that component.
- You try to access a component referenced by a null pointer.
- Two data structures share a pointer, one calls a delete on it. You try to access it through the other data structure. Or maybe you try to delete again it there.
- You don't call delete on a pointer to a component no longer used by a data structure.
  - ***A lot of these problems arise because of "by hand" memory management***

A BUGGY BOX

```
class Box {
public:
  int* ptr;
  Box(int value) : ptr {new int {value}} { }
  Box(const Box& b) : ptr {b.ptr} { }
 ~Box(void) { delete ptr; }
};
```

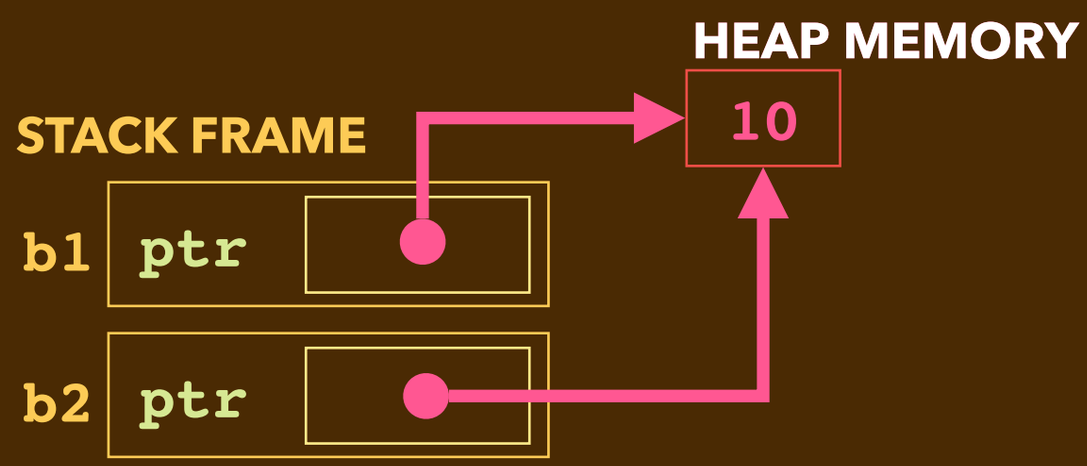
```
int main(void) {
  Box b1 { 10 };
  Box b2 { b1 };
  Box b3 { 11 };
  b3 = b2;
}
```



A BUGGY BOX

```
class Box {
public:
  int* ptr;
  Box(int value) : ptr {new int {value}} { }
  Box(const Box& b) : ptr {b.ptr} { }
 ~Box(void) { delete ptr; }
};
```

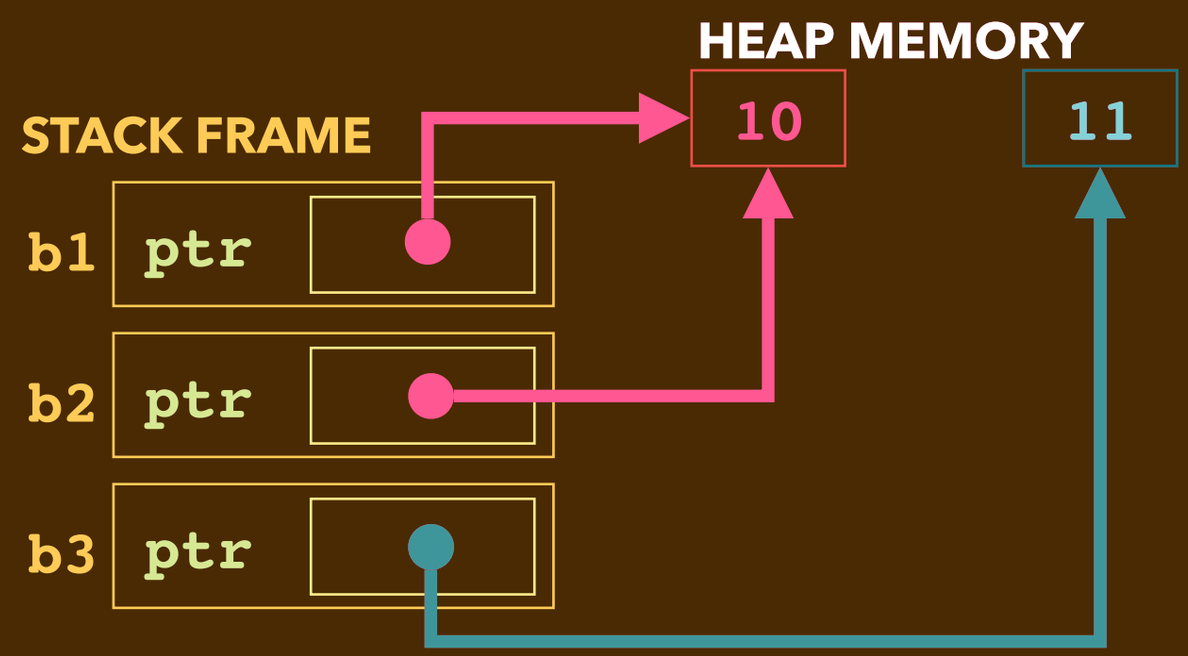
```
int main(void) {
  Box b1 { 10 };
  Box b2 { b1 };
  Box b3 { 11 };
  b3 = b2;
}
```



A BUGGY BOX

```
class Box {
public:
  int* ptr;
  Box(int value) : ptr {new int {value}} { }
  Box(const Box& b) : ptr {b.ptr} { }
 ~Box(void) { delete ptr; }
};
```

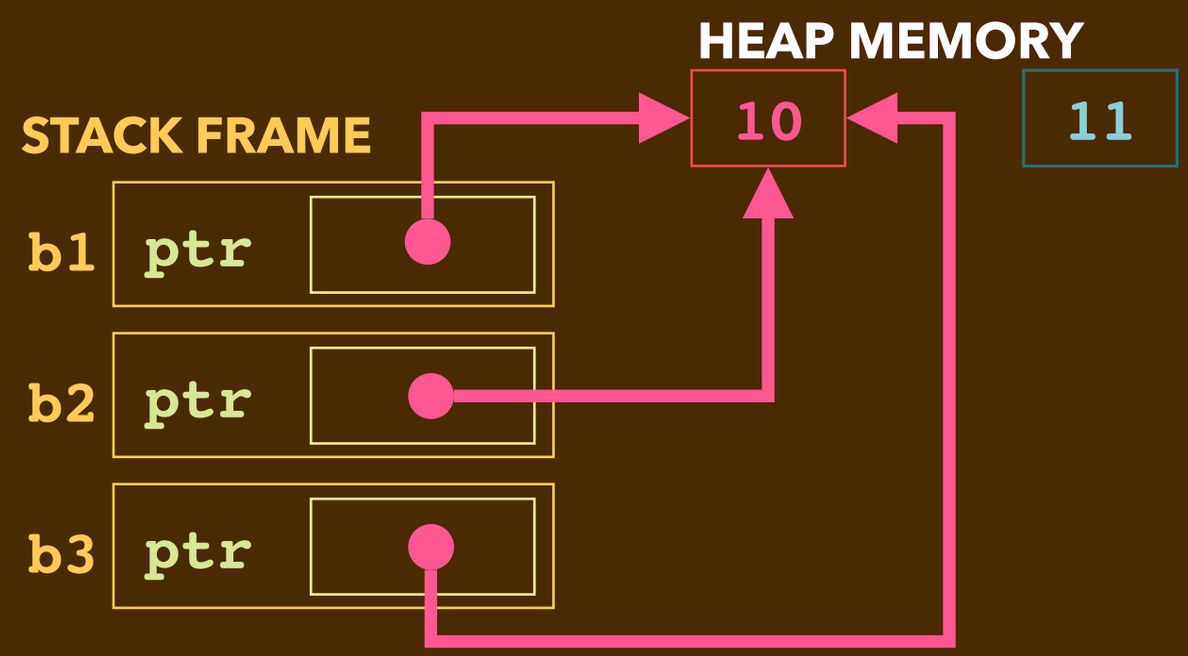
```
int main(void) {
  Box b1 { 10 };
  Box b2 { b1 };
  Box b3 { 11 };
  b3 = b2;
}
```



A BUGGY BOX

```
class Box {
public:
  int* ptr;
  Box(int value) : ptr {new int {value}} { }
  Box(const Box& b) : ptr {b.ptr} { }
 ~Box(void) { delete ptr; }
};
```

```
int main(void) {
  Box b1 { 10 };
  Box b2 { b1 };
  Box b3 { 11 };
  b3 = b2;
}
```



Changed reference in b3.

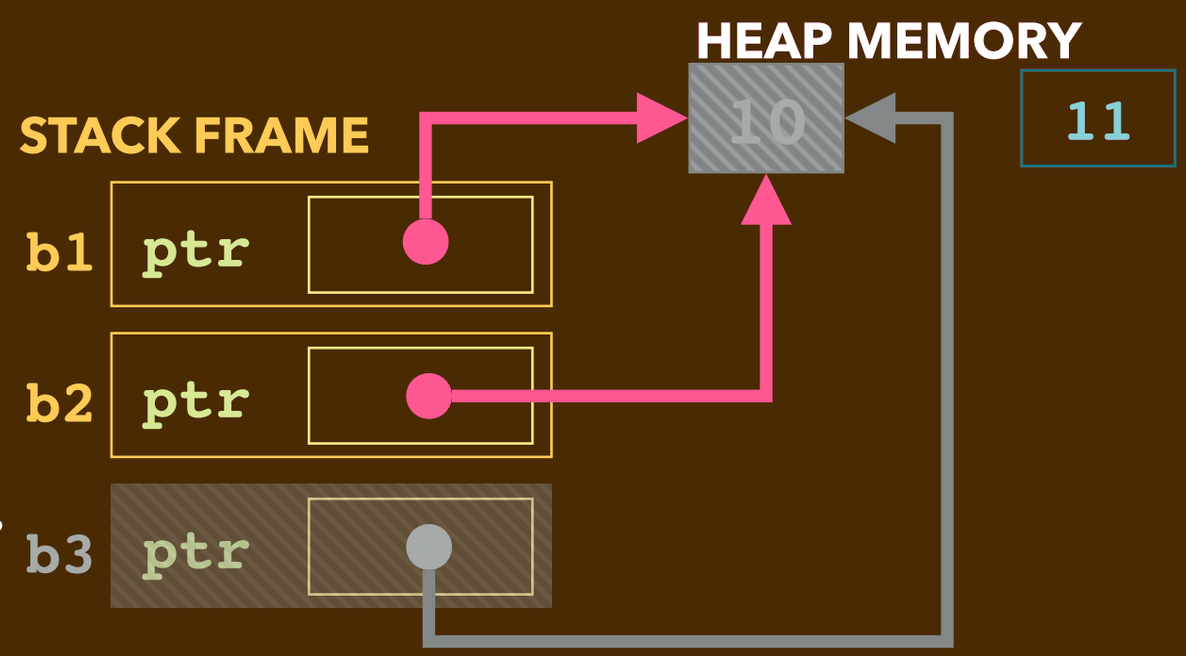
This is a "memory leak" error.

A BUGGY BOX

```
class Box {
public:
  int* ptr;
  Box(int value) : ptr {new int {value}} { }
  Box(const Box& b) : ptr {b.ptr} { }
 ~Box(void) { delete ptr; }
};
```

```
int main(void) {
  Box b1 { 10 };
  Box b2 { b1 };
  Box b3 { 11 };
  b3 = b2;
}
```

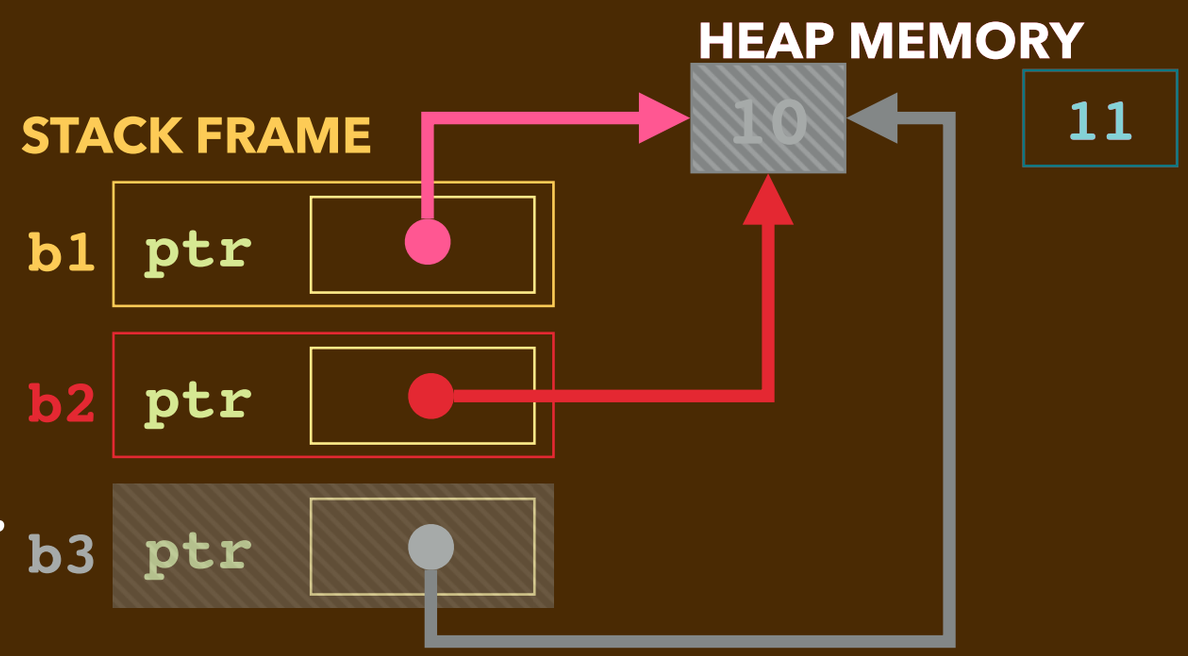
Destructor called on b3, b3.ptr deleted.



A BUGGY BOX

```
class Box {
public:
  int* ptr;
  Box(int value) : ptr {new int {value}} { }
  Box(const Box& b) : ptr {b.ptr} { }
 ~Box(void) { delete ptr; }
};
```

```
int main(void) {
  Box b1 { 10 };
  Box b2 { b1 };
  Box b3 { 11 };
  b3 = b2;
}
```



Destructor called on b3, b3.ptr deleted.

Destructor called on b2, b2.ptr delete.

This is a "double delete" error.

BUGS BUGS BUGS

- These kinds of bugs are so rampant in C++ code.
- Many of them go unnoticed.
- Current practice and research is devoted to finding/preventing these bugs.

BUGS BUGS BUGS

- These kinds of bugs are so rampant in C++ code.
- Many of them go unnoticed.
- Current practice and research is devoted to finding/preventing these bugs.
  - ➡Programming disciplines/idioms developed.
  - ➡Programming libraries developed.
  - ➡Programming languages developed.
  - ➡Program analysis tools developed.
  - ➡Runtime instruments developed.
  - ➡Testing strategies developed.

AUTOMATIC GARBAGE COLLECTION

- Some problems can be prevented/solved with automatic garbage collection.
  - ➡ A runtime component checks whether any part of the code can access an object.
  - ➡ If not, it reclaims that object's storage.
- **Question:** How does it do that?
- **Answers:** There are several ways.
  - **E.g.** a "stop-the-world mark-and-sweep" garbage collector halts the program, briefly, then scans through the program's stack frames and marks what objects are reachable by links. Unreachable objects are reclaimed,
  - **E.g.** in a "reference count" scheme, every object has a count of how many things point to it. When that count goes to 0, its storage is reclaimed.

MARK-AND-SWEEP ALGORITHM

- Behavior of a *"stop-the-world, mark-and-sweep"* garbage collector
  - halts the program
  - scans through the program's stack frames looking for links
  - marks what heap objects are reachable by pointers in the frame
  - from those, marks what other objects are reachable
  - from those, marks what additional objects are reachable
  - ...
    - ➡ Whatever is left unmarked is considered garbage
    - ➡ It is reclaimed by the heap.

MARK-AND-SWEEP ALGORITHM

- Behavior of a *"stop-the-world, mark-and-sweep"* garbage collector
  - halts the program
  - scans through the program's stack frames looking for links ✓
  - marks what heap objects are reachable by pointers in the frame
  - from those, marks what other objects are reachable
  - from those, marks what additional objects are reachable
  - ...
    - ➡ Whatever is left unmarked is considered garbage
    - ➡ It is reclaimed by the heap.

MARK-AND-SWEEP ALGORITHM

- Behavior of a *"stop-the-world, mark-and-sweep"* garbage collector
  - halts the program
  - scans through the program's stack frames looking for links ✓
  - marks what heap objects are reachable by pointers in the frame ✓
  - from those, marks what other objects are reachable
  - from those, marks what additional objects are reachable
  - ...
    - ➡ Whatever is left unmarked is considered garbage
    - ➡ It is reclaimed by the heap.

MARK-AND-SWEEP ALGORITHM

- Behavior of a *"stop-the-world, mark-and-sweep"* garbage collector
  - halts the program
  - scans through the program's stack frames looking for links ✓
  - marks what heap objects are reachable by pointers in the frame ✓
  - from those, marks what other objects are reachable ✓
  - from those, marks what additional objects are reachable
  - ...
    - ➡ Whatever is left unmarked is considered garbage
    - ➡ It is reclaimed by the heap.

MARK-AND-SWEEP ALGORITHM

- Behavior of a *"stop-the-world, mark-and-sweep"* garbage collector
  - halts the program
  - scans through the program's stack frames looking for links ✓
  - marks what heap objects are reachable by pointers in the frame ✓
  - from those, marks what other objects are reachable ✓
  - from those, marks what additional objects are reachable ✓
  - ...
    - ➡ Whatever is left unmarked is considered garbage
    - ➡ It is reclaimed by the heap.

MARK-AND-SWEEP ALGORITHM

- Behavior of a *"stop-the-world, mark-and-sweep"* garbage collector
  - halts the program
  - scans through the program's stack frames looking for links ✓
  - marks what heap objects are reachable by pointers in the frame ✓
  - from those, marks what other objects are reachable ✓
  - from those, marks what additional objects are reachable ✓
  - ...

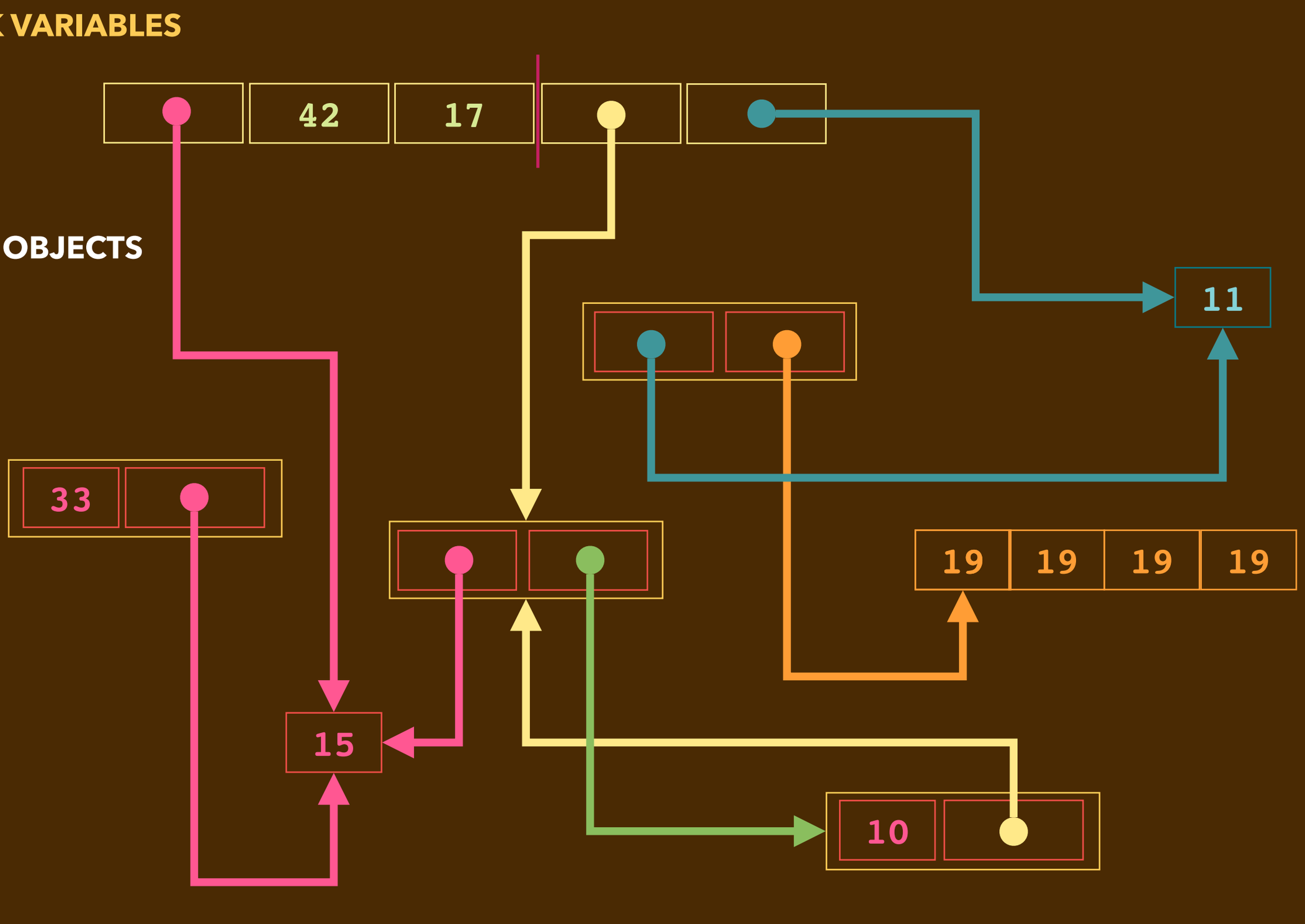
    ➡ Whatever is left unmarked is considered *garbage* ∅

    ➡ It is reclaimed by the heap.

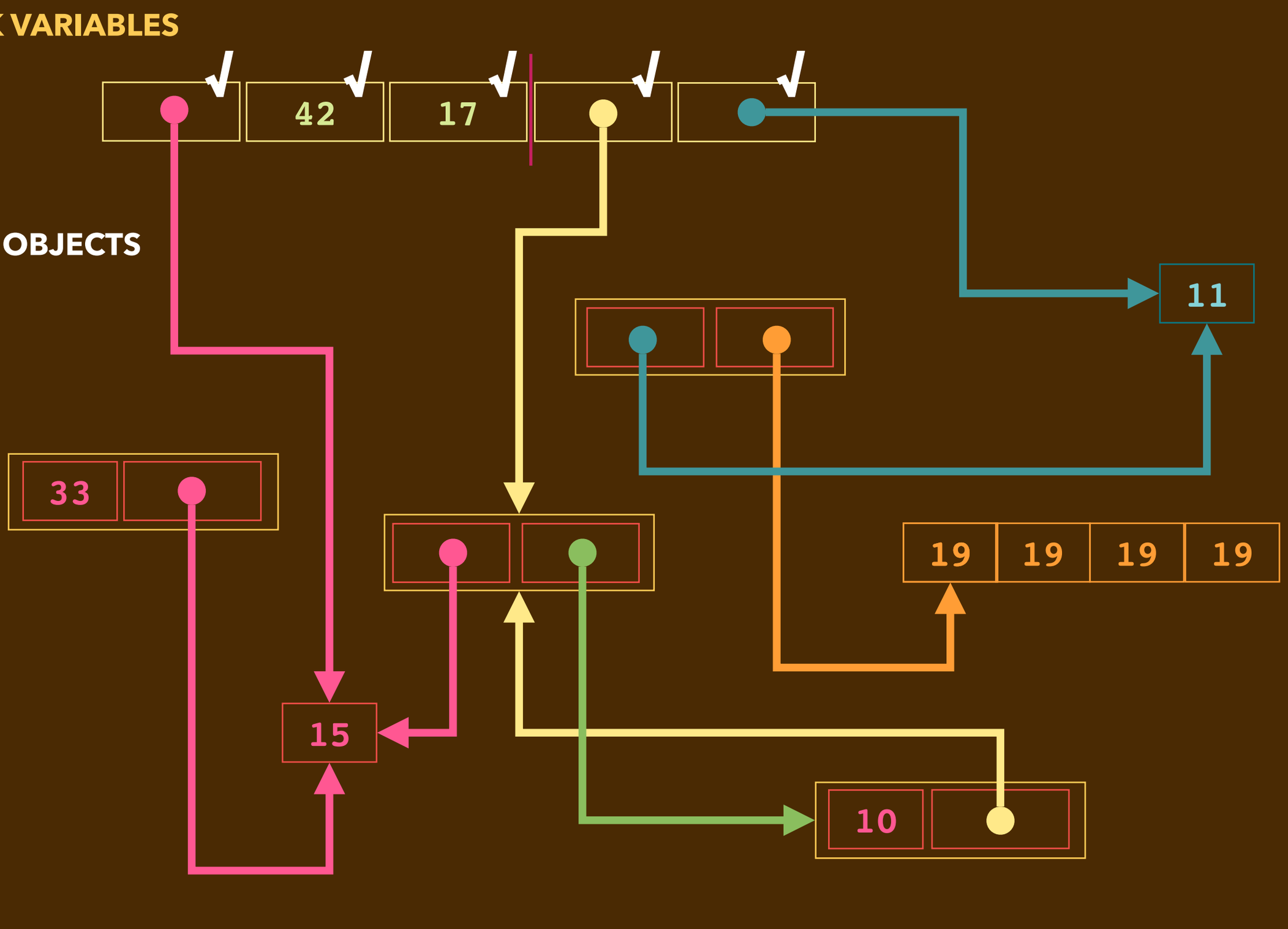
MARK-AND-SWEEP ALGORITHM

- Behavior of a *"stop-the-world, mark-and-sweep"* garbage collector
  - halts the program
  - scans through the program's stack frames looking for links ✓
  - marks what heap objects are reachable by pointers in the frame ✓
  - from those, marks what other objects are reachable ✓
  - from those, marks what additional objects are reachable ✓
  - ...
    - ➡ Whatever is left unmarked is considered *garbage* ø
    - ➡ It is ***reclaimed*** by the heap.

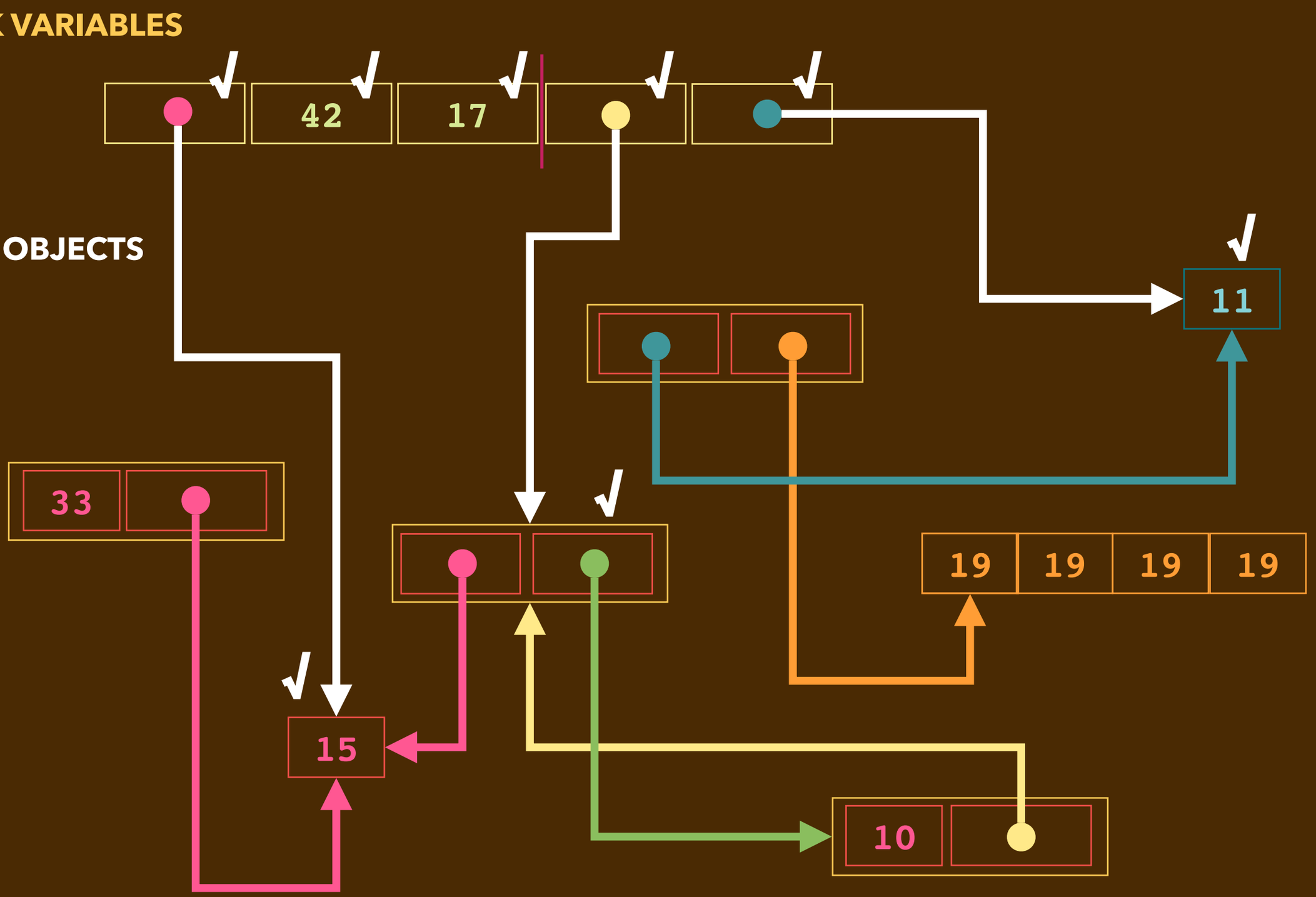
MEMORY DIAGRAM: MARK AND SWEEP



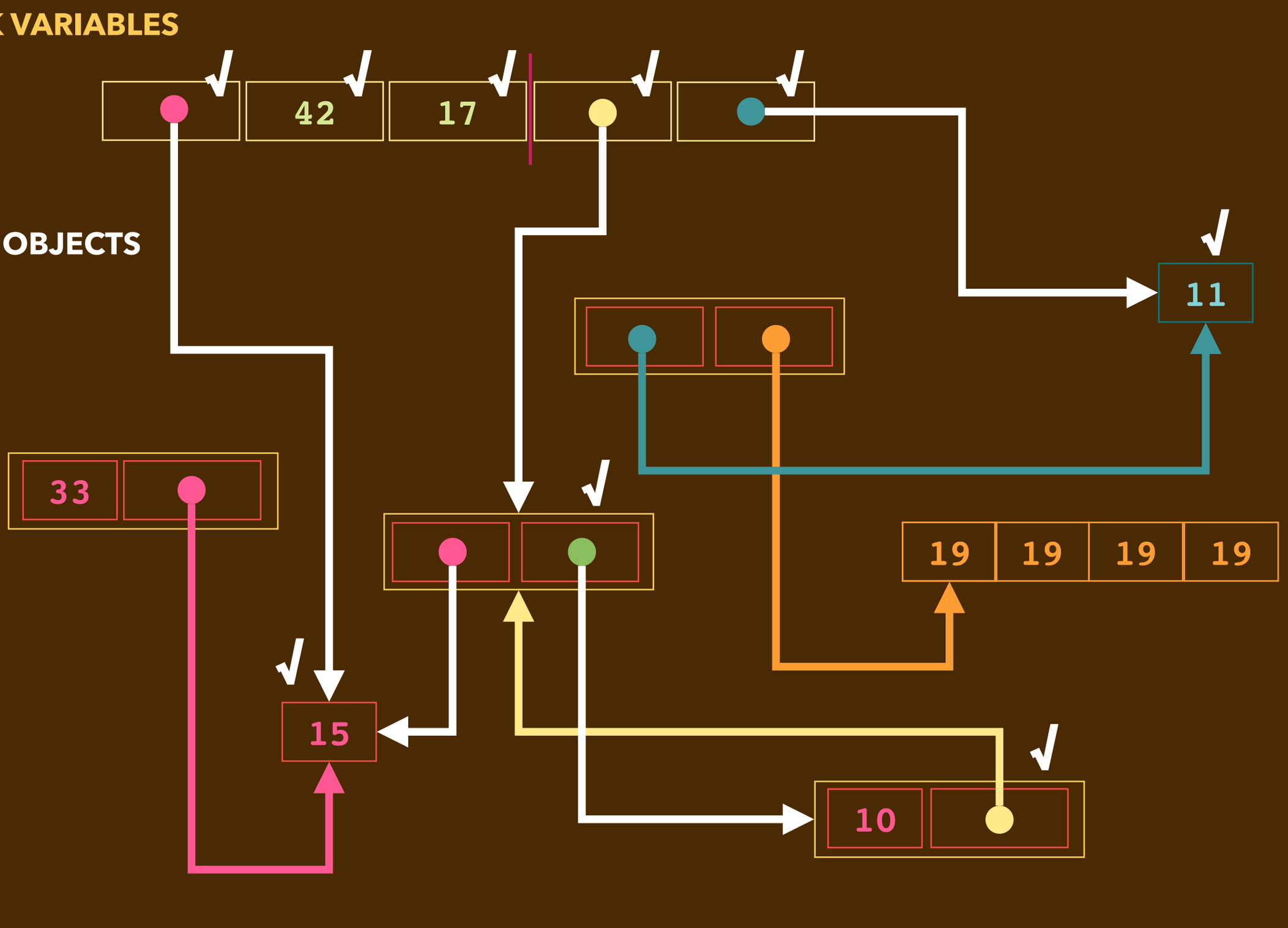
MEMORY DIAGRAM: MARK AND SWEEP



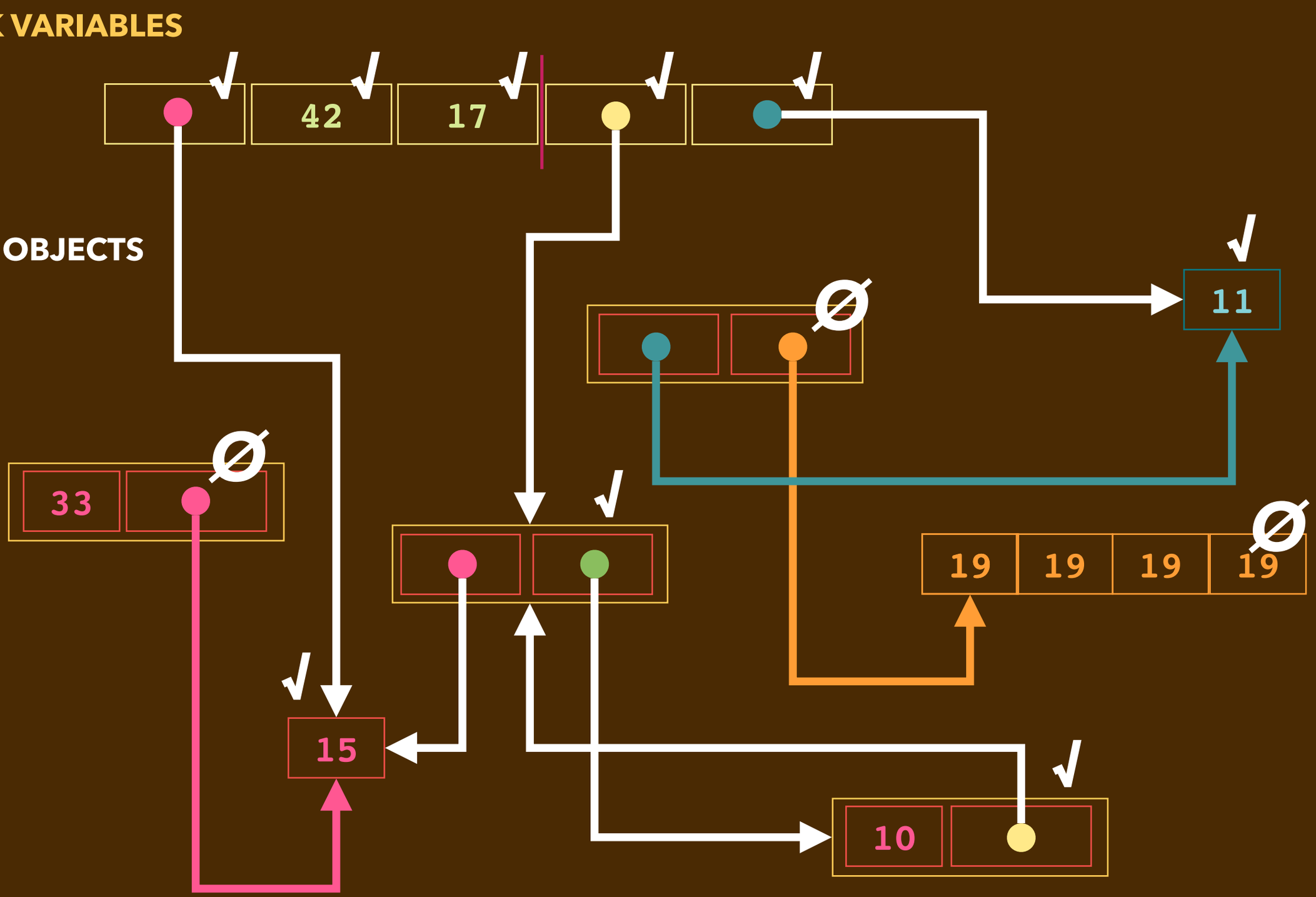
MEMORY DIAGRAM: MARK AND SWEEP



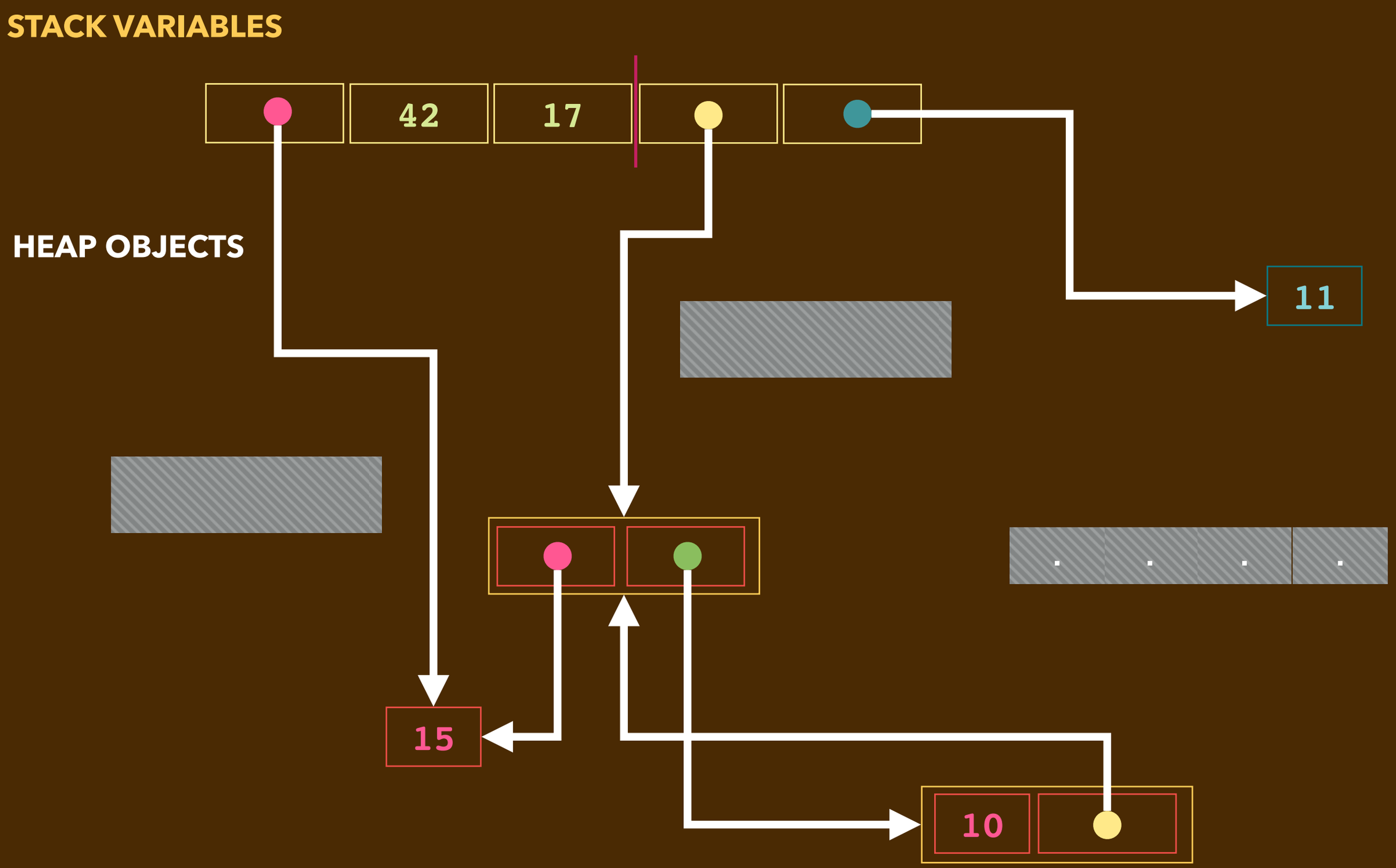
MEMORY DIAGRAM: MARK AND SWEEP



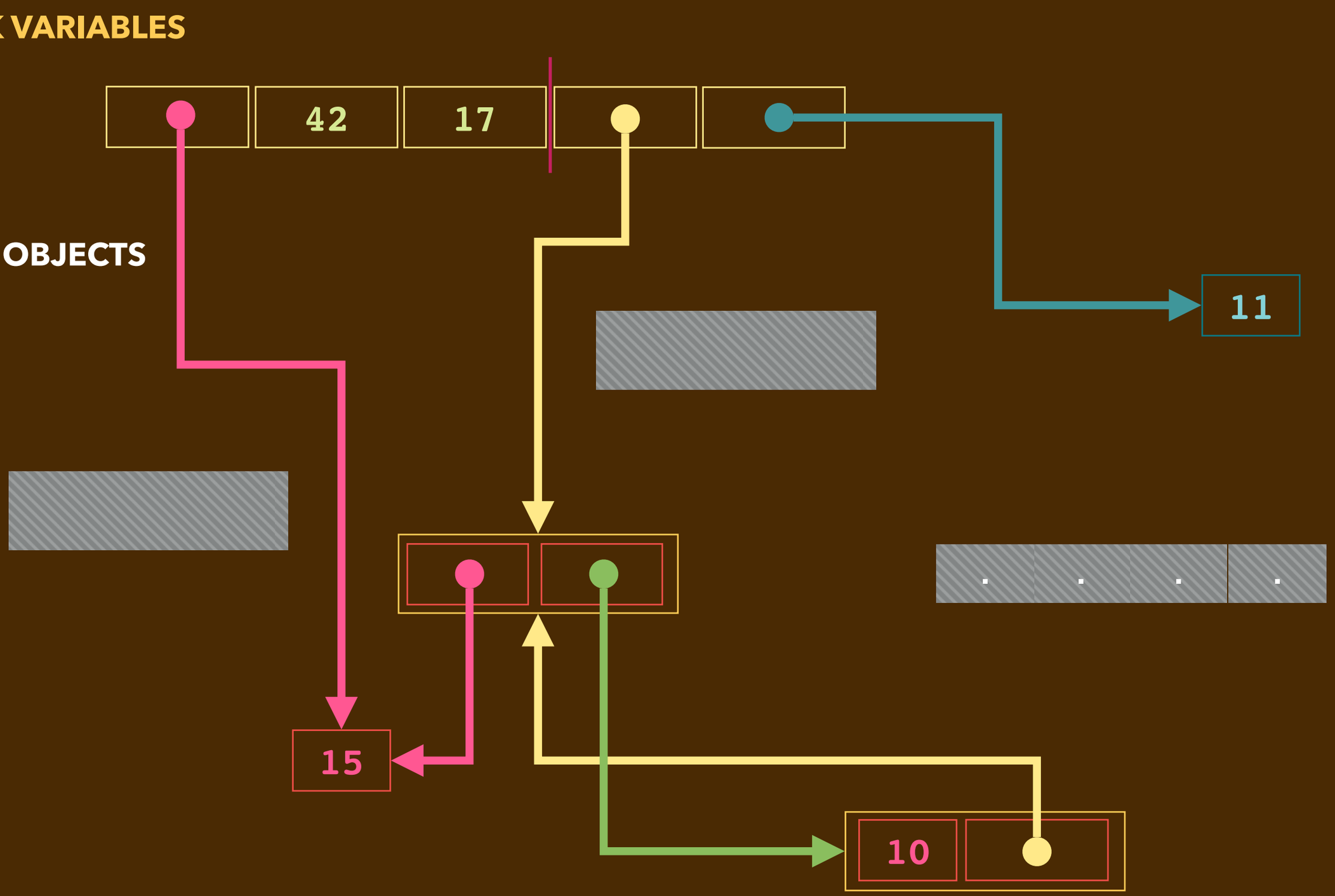
MEMORY DIAGRAM: MARK AND SWEEP



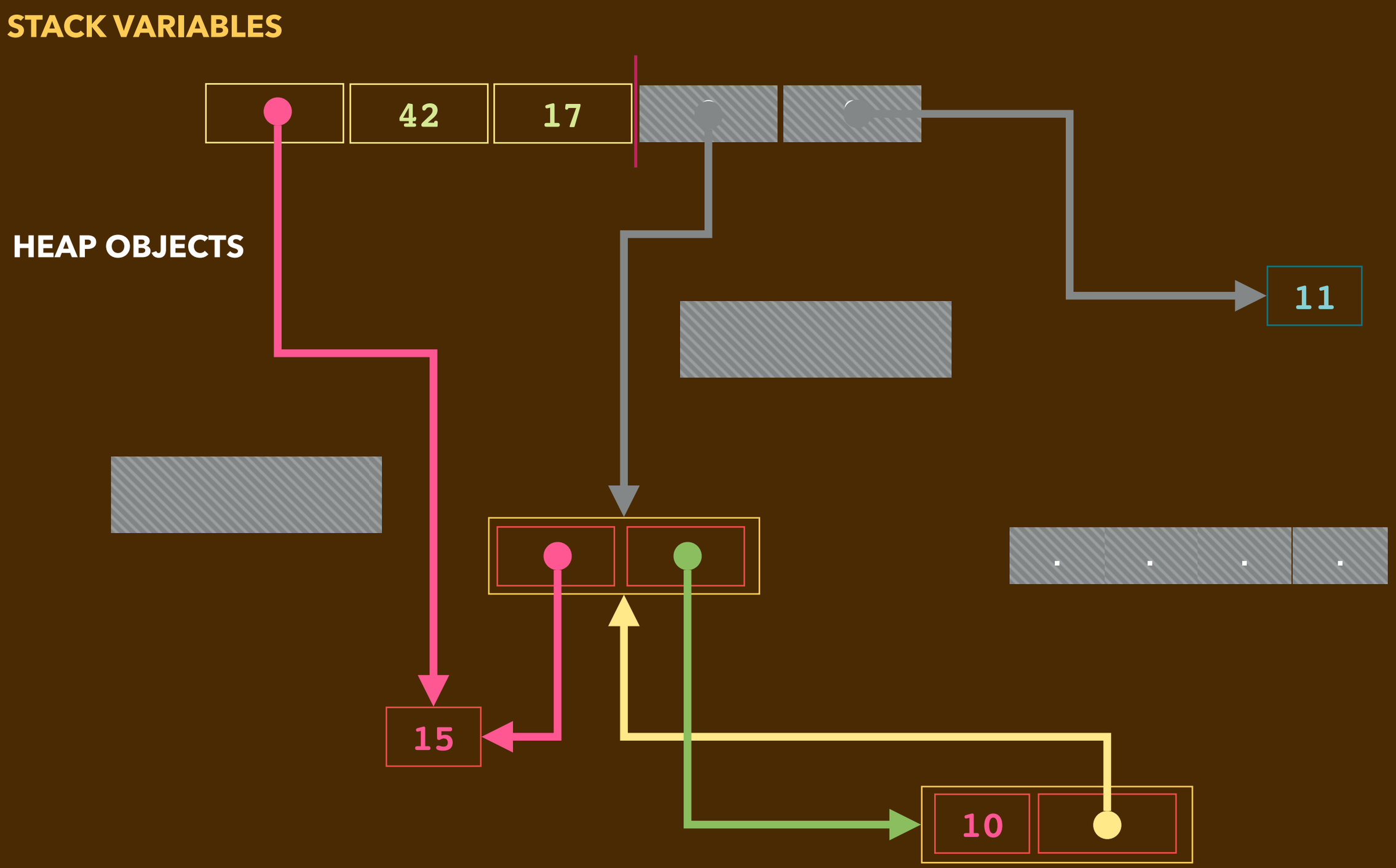
MEMORY DIAGRAM: MARK AND SWEEP



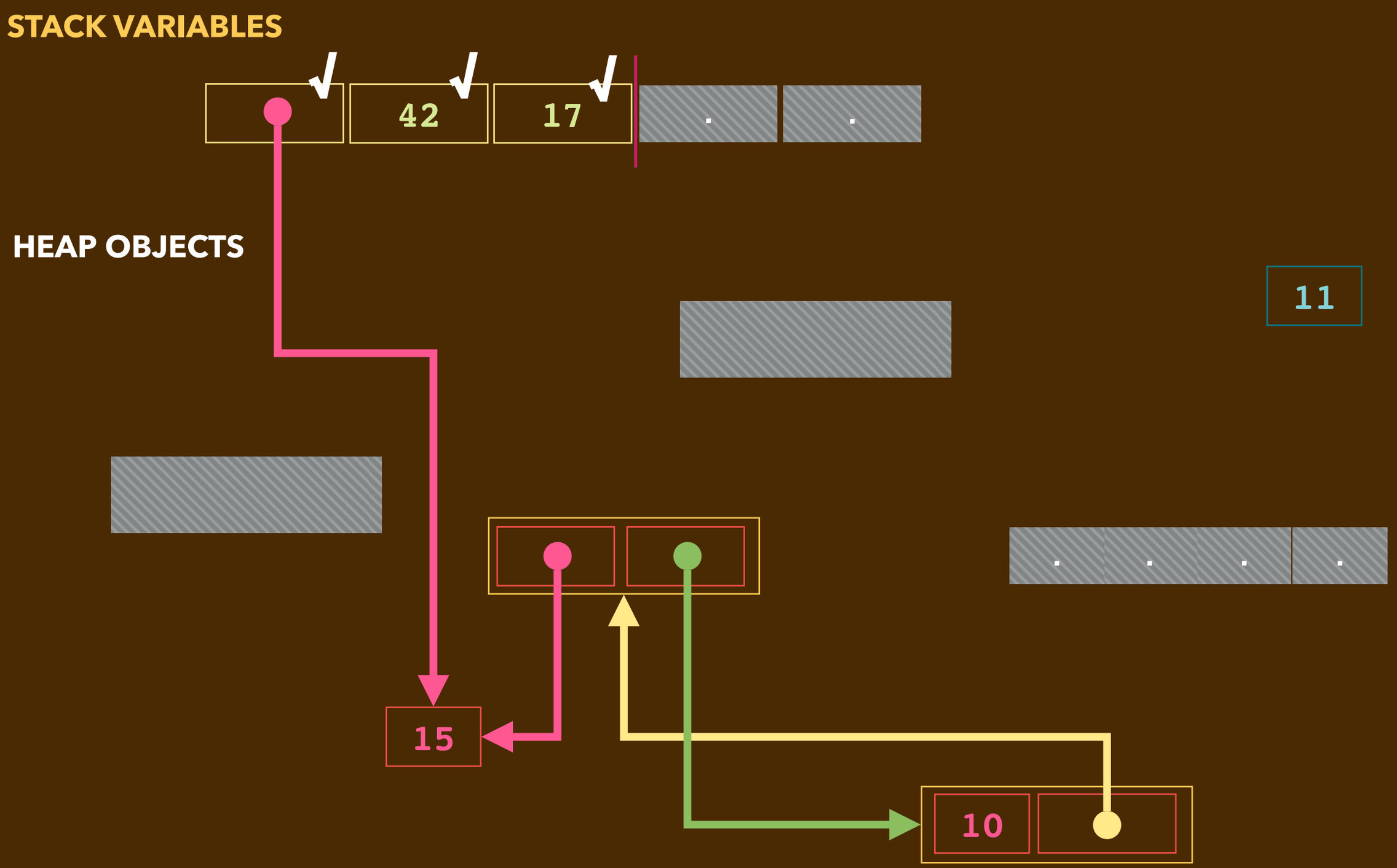
MEMORY DIAGRAM: MARK AND SWEEP



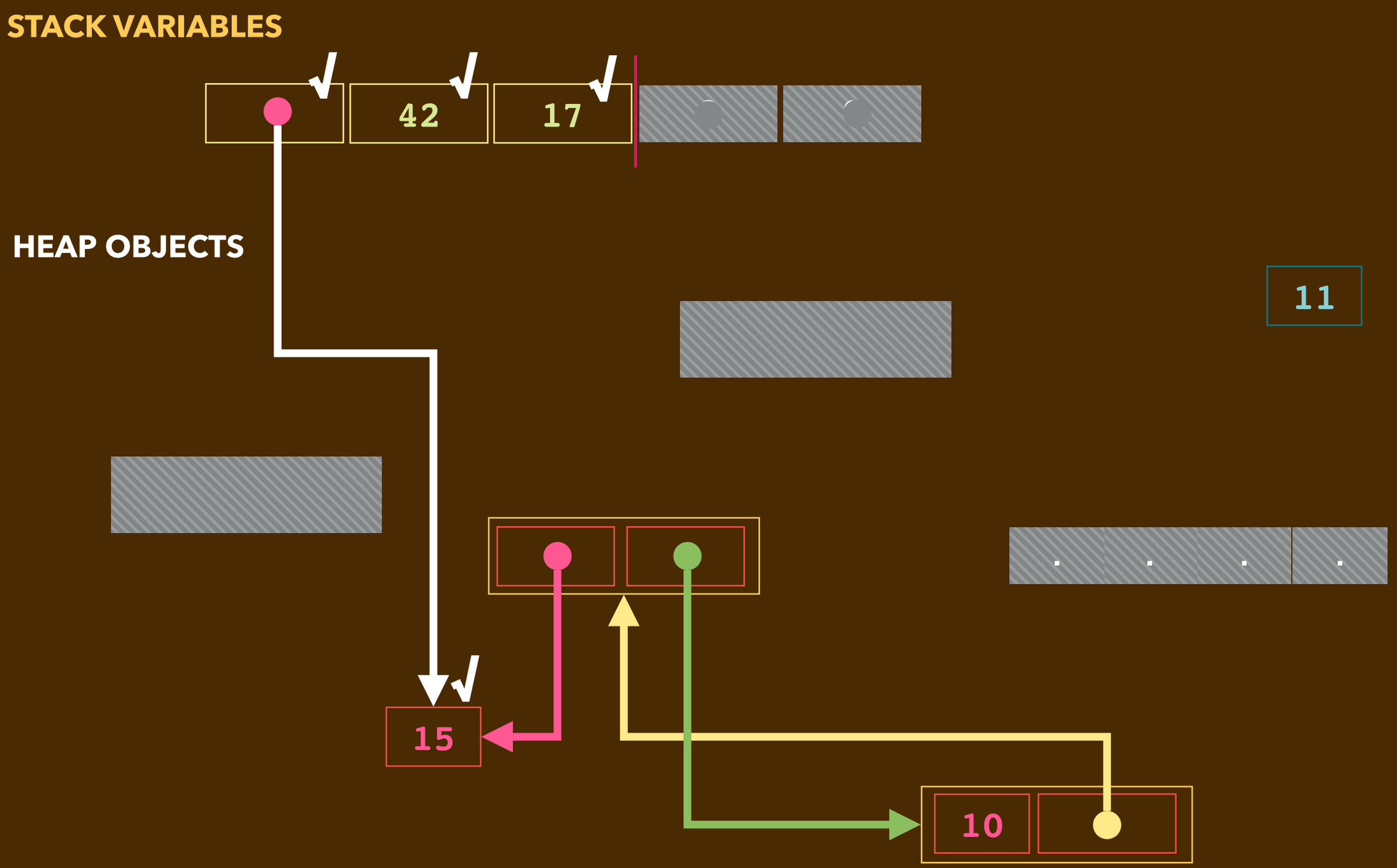
MEMORY DIAGRAM: MARK AND SWEEP



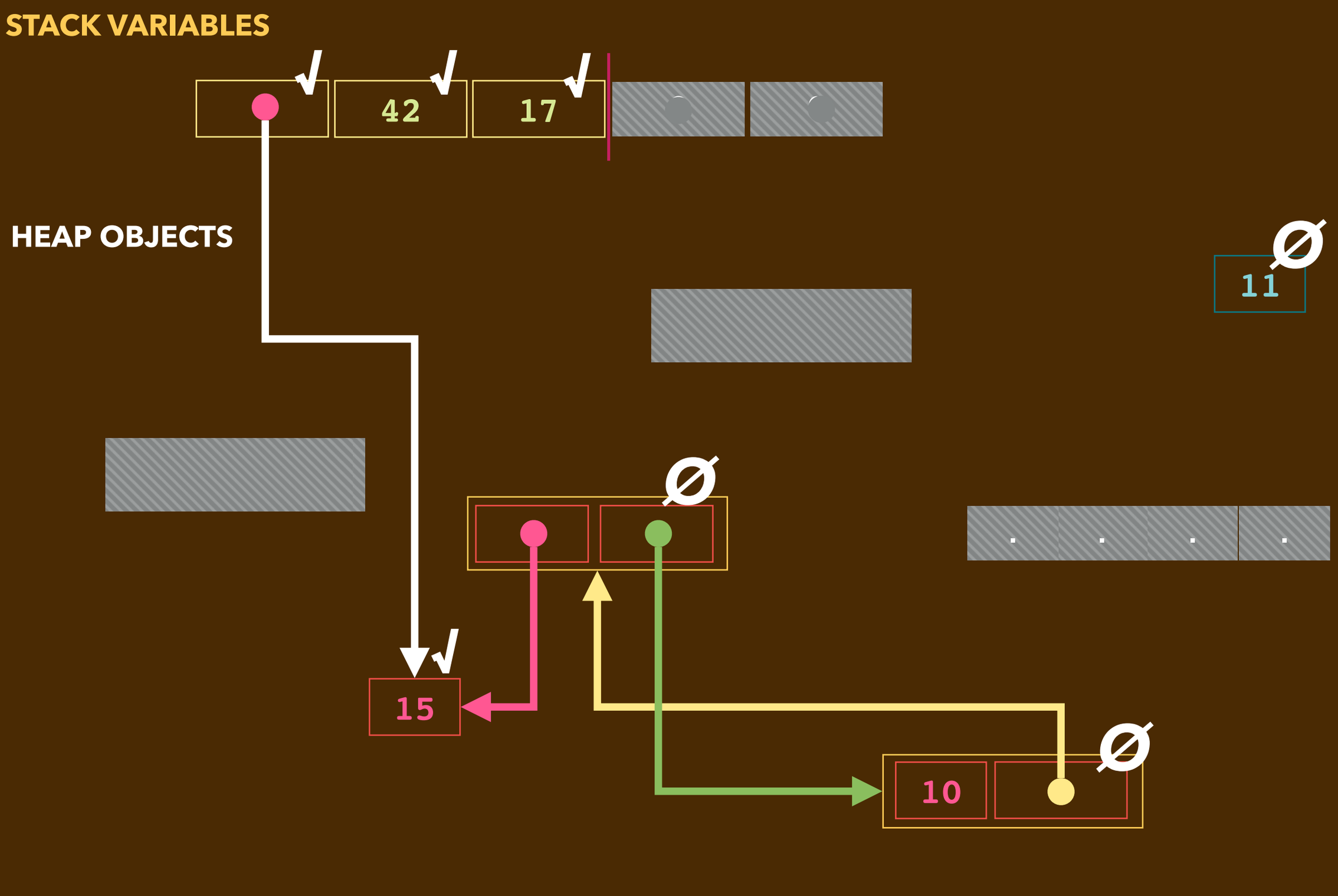
MEMORY DIAGRAM: MARK AND SWEEP



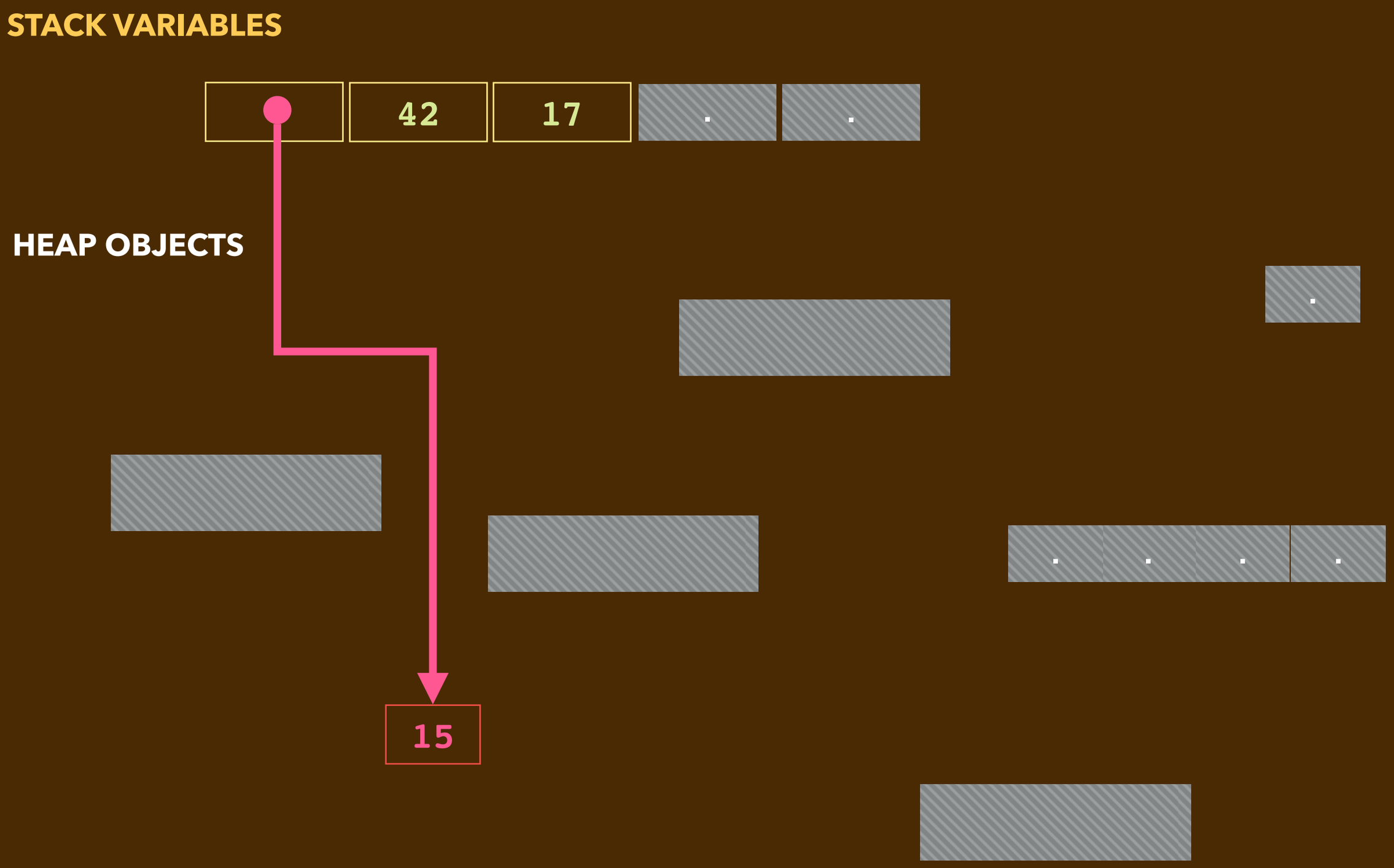
MEMORY DIAGRAM: MARK AND SWEEP



MEMORY DIAGRAM: MARK AND SWEEP



MEMORY DIAGRAM: MARK AND SWEEP



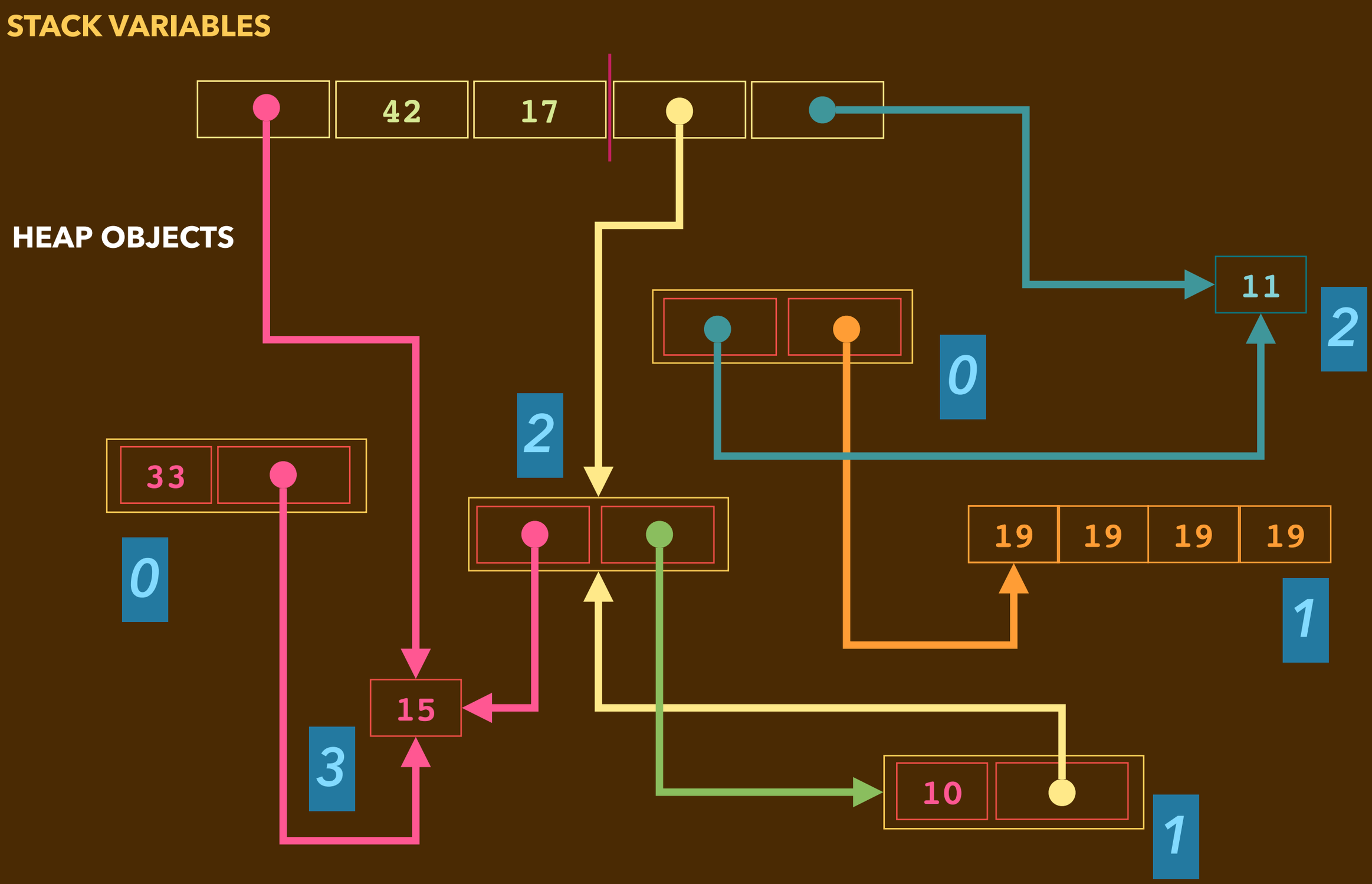
REFERENCE COUNTING

- As a program runs, we track *how many times each heap object is referenced*
  - A fresh heap object offered by `new` has a count of one. **COUNT := 1**
  - When program variables and objects link to that object, **COUNT ++**
  - When program variables and objects link to something else, **COUNT --**

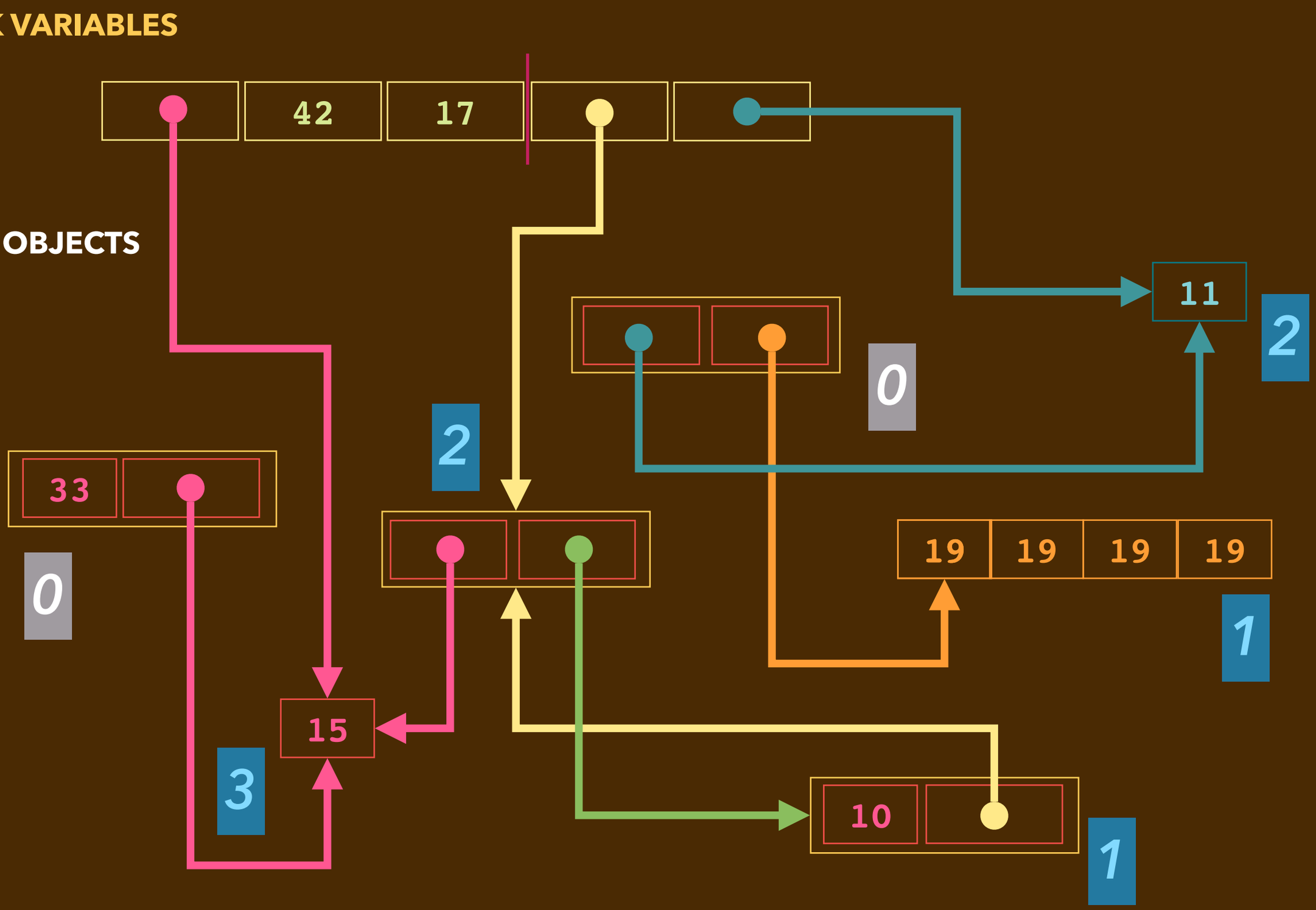
➡ Whatever has a **COUNT == 0** is considered *garbage*.

➡ It is ***reclaimed*** by the heap.

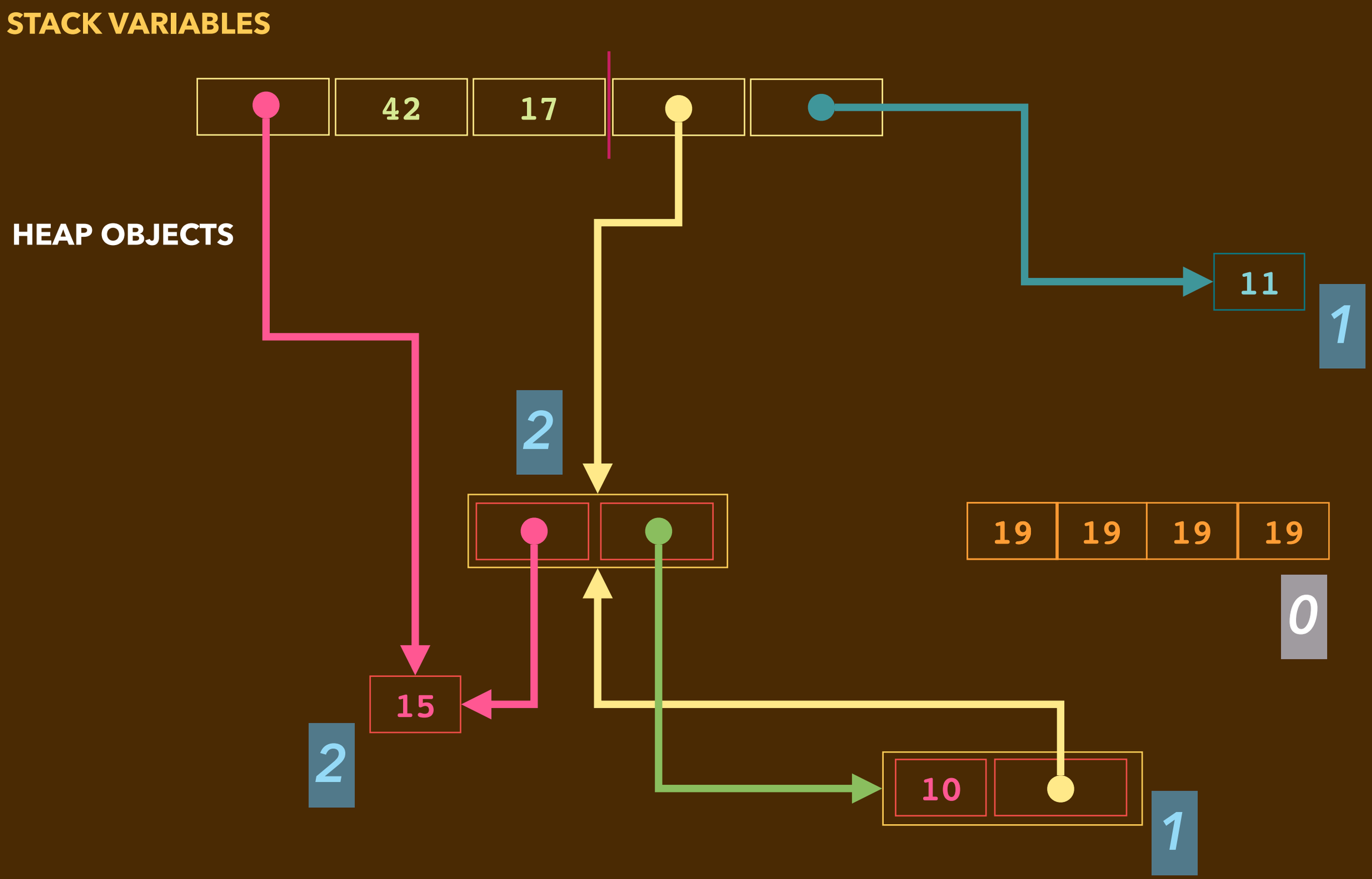
MEMORY DIAGRAM: REFERENCE COUNTS



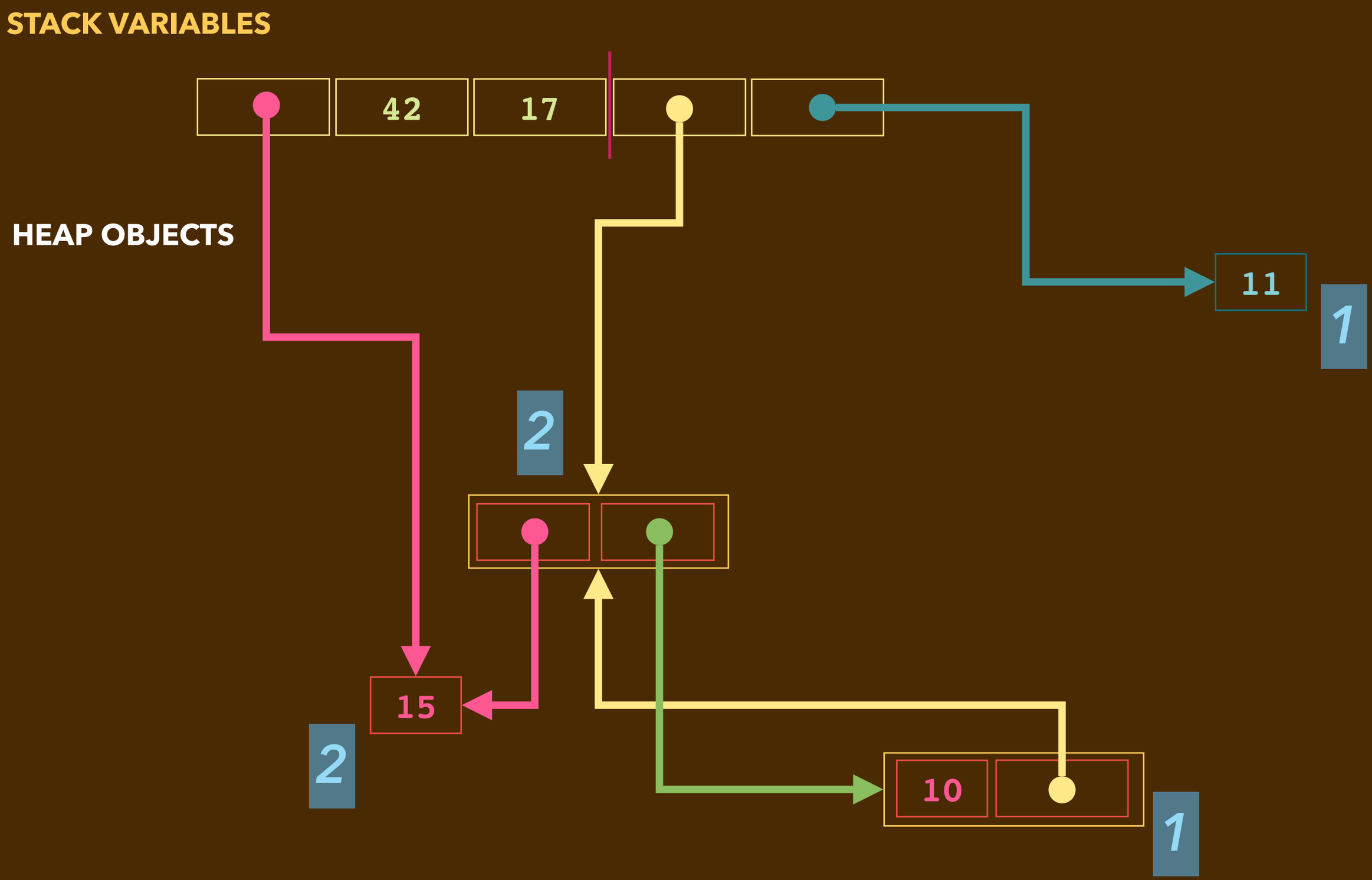
MEMORY DIAGRAM: REFERENCE COUNTS



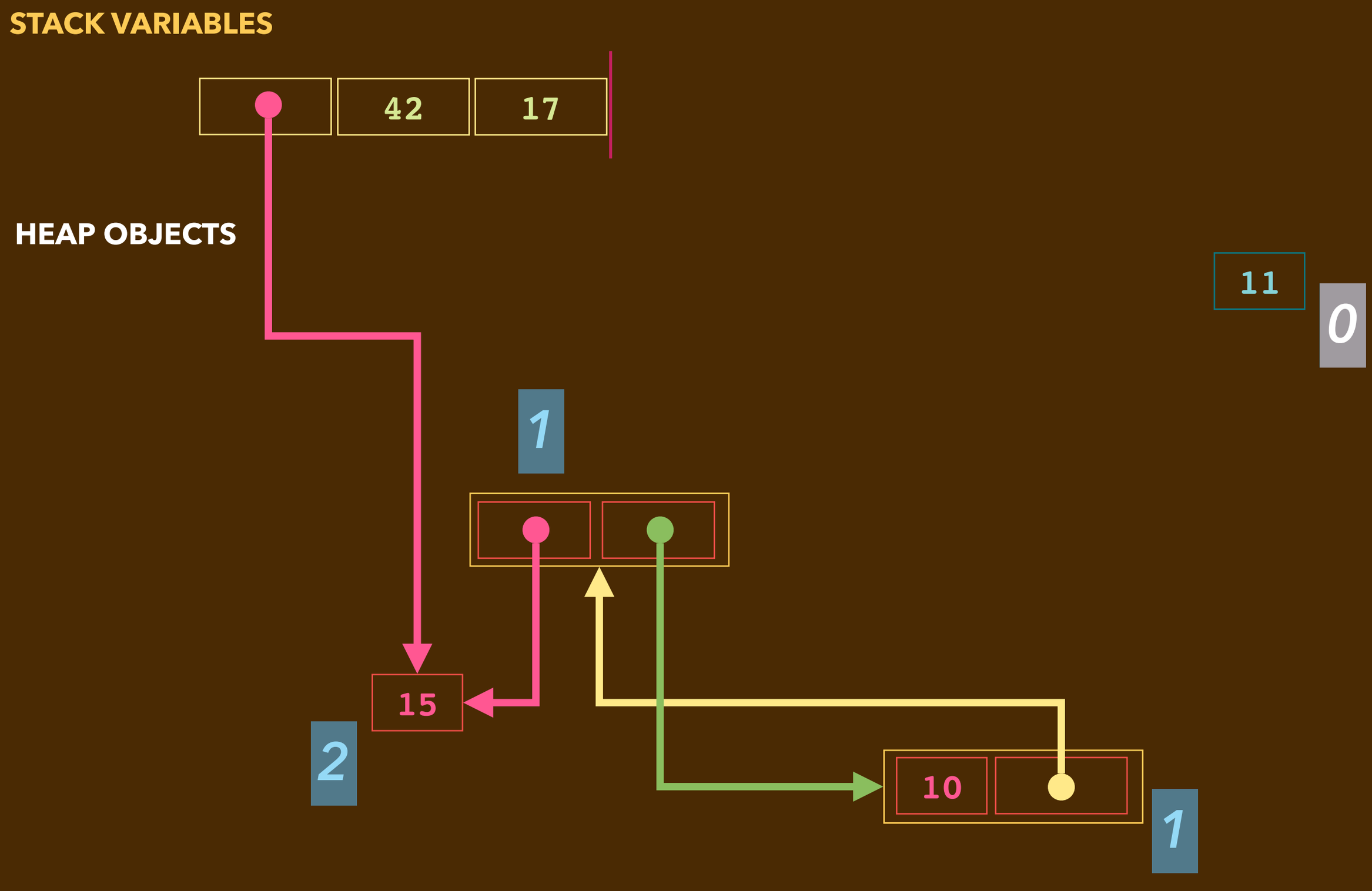
MEMORY DIAGRAM: REFERENCE COUNTS



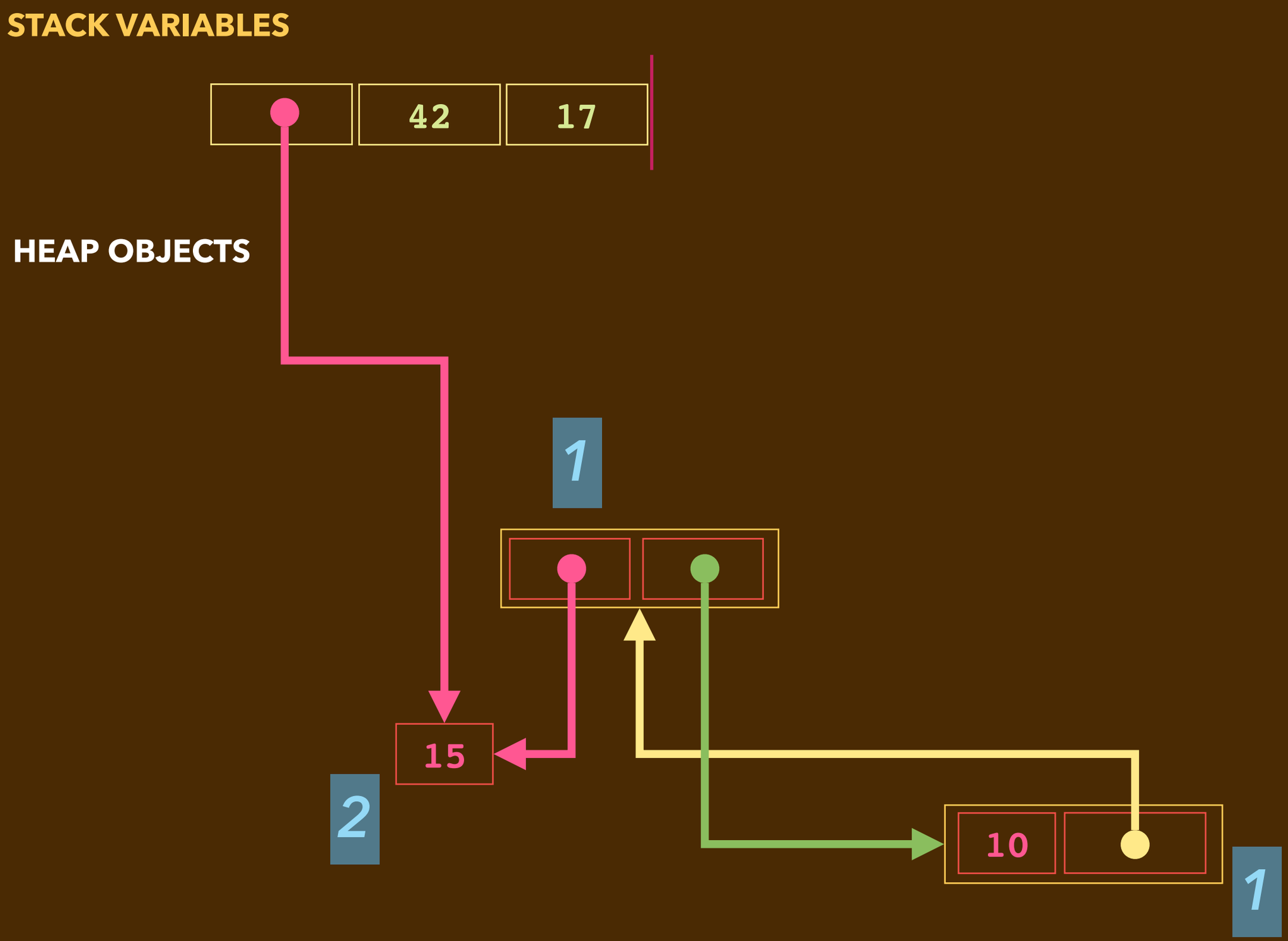
MEMORY DIAGRAM: REFERENCE COUNTS



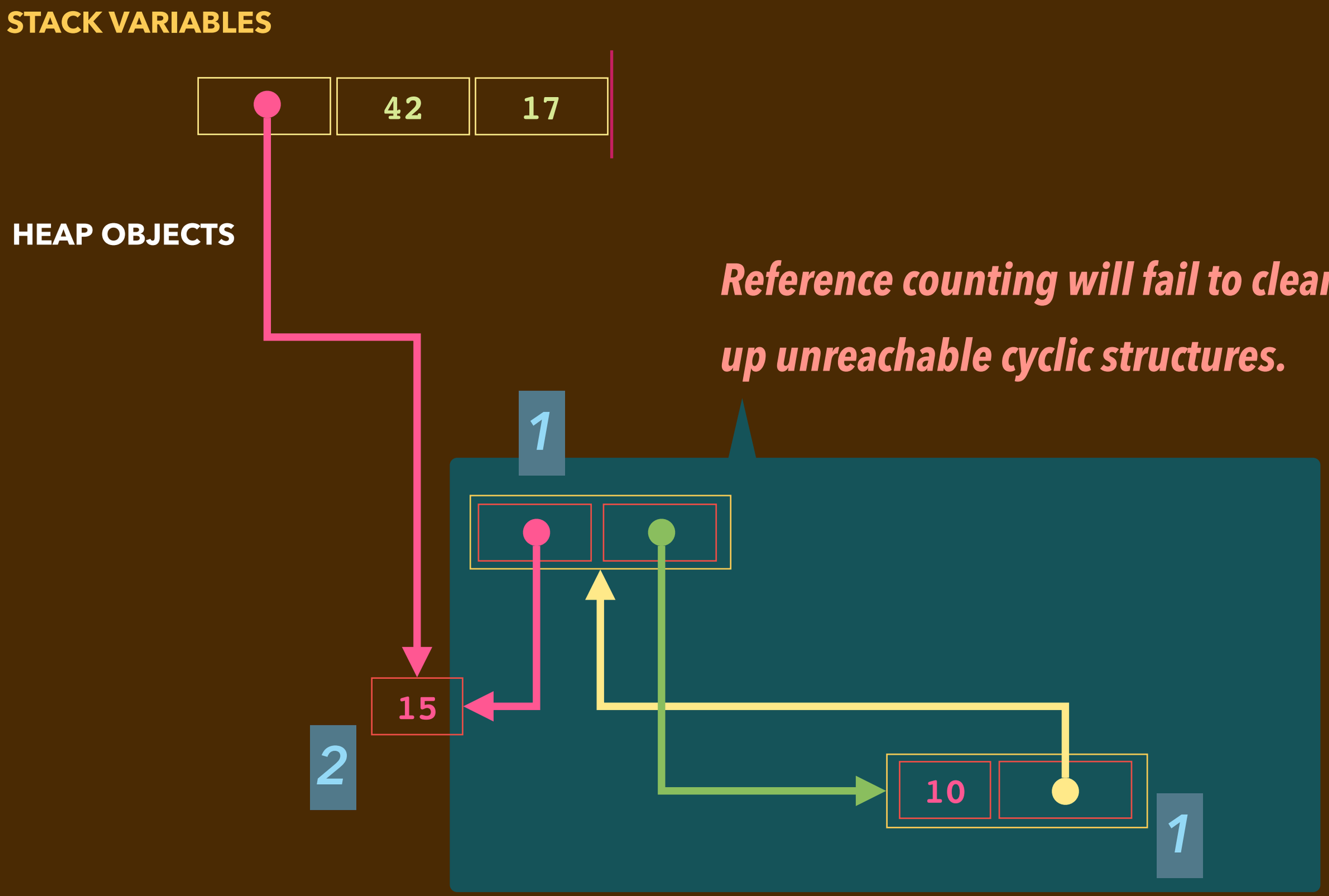
MEMORY DIAGRAM: REFERENCE COUNTS



MEMORY DIAGRAM: REFERENCE COUNTS



MEMORY DIAGRAM: REFERENCE COUNTS



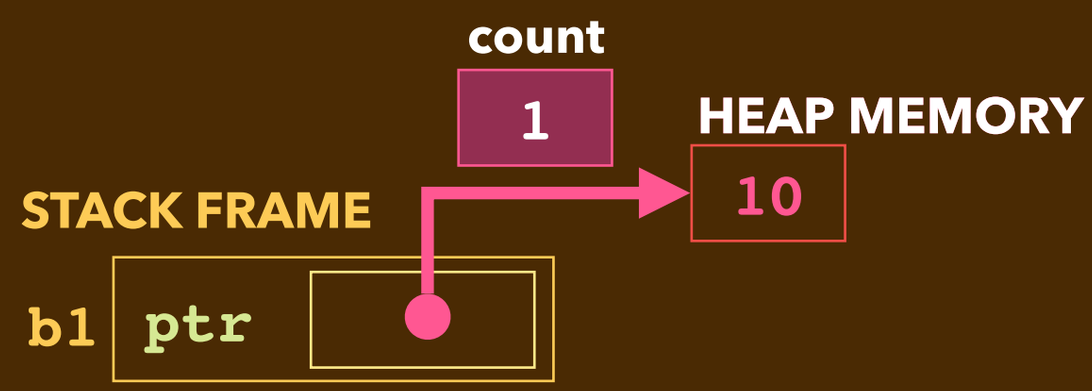
RECALL: SMART POINTERS IN THE C++ STL

- 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 `delete`d, 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 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 BOX

```
#include <memory>
class Box {
public:
  std::shared_ptr<int> ptr;
  Box(int value) : ptr {new int {value}} { }
  Box(const Box& b) : ptr {b.ptr} { }
};
```

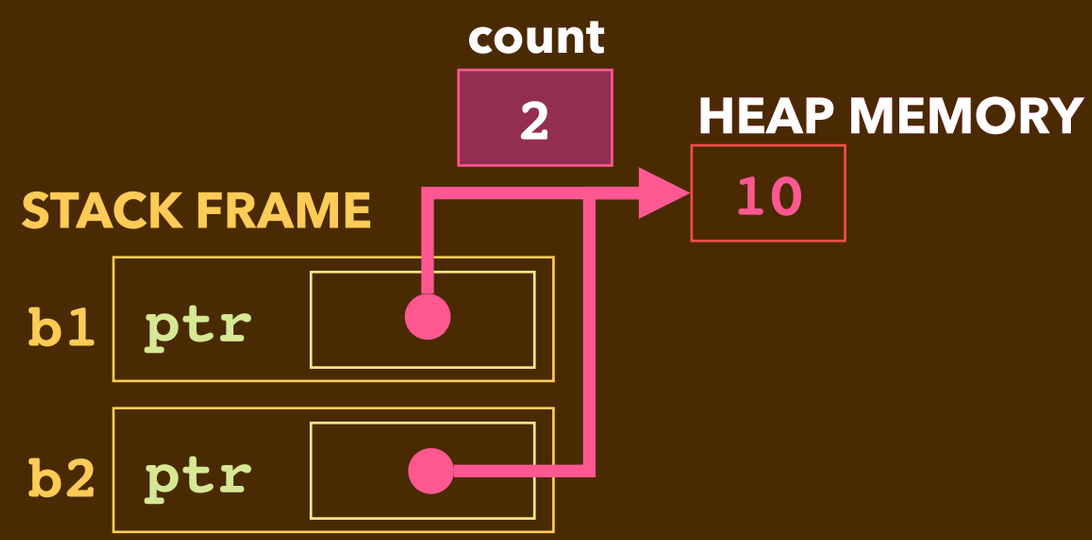
```
int main(void) {
  Box b1 { 10 };
  Box b2 { b1 };
  Box b3 { 11 };
  b3 = b2;
}
```



A SHARED_PTR BOX

```
#include <memory>
class Box {
public:
  std::shared_ptr<int> ptr;
  Box(int value) : ptr {new int {value}} { }
  Box(const Box& b) : ptr {b.ptr} { }
};
```

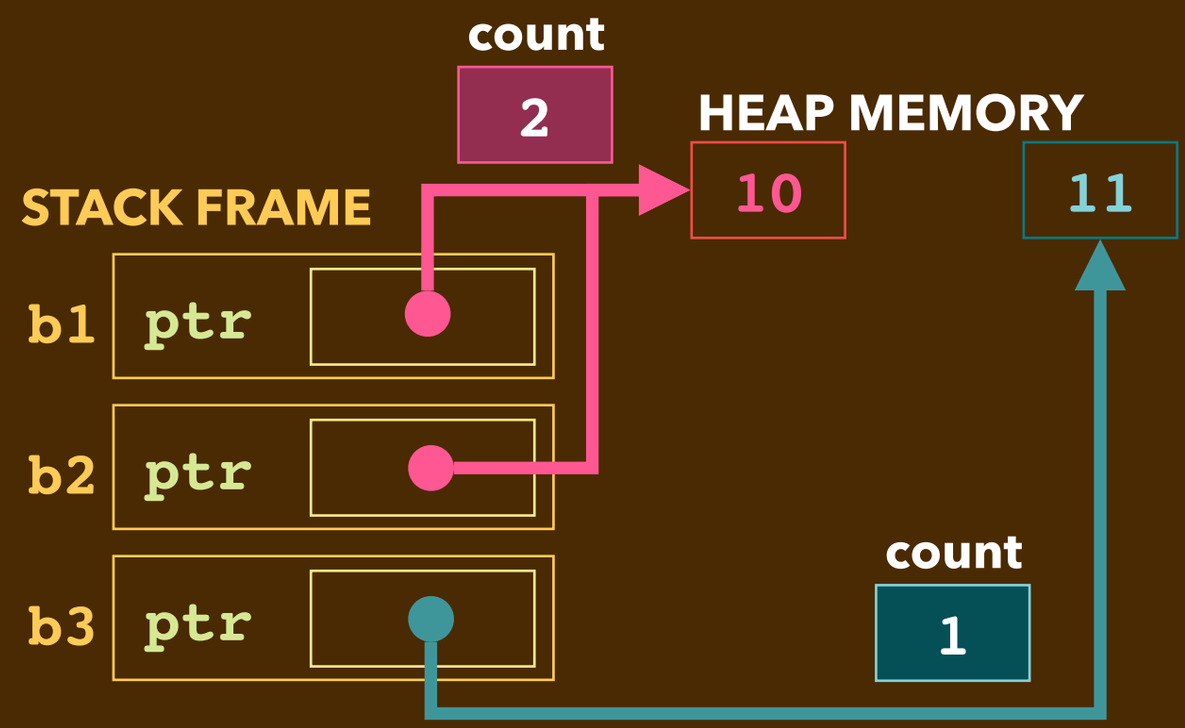
```
int main(void) {
  Box b1 { 10 };
  Box b2 { b1 };
  Box b3 { 11 };
  b3 = b2;
}
```



A SHARED_PTR BOX

```
#include <memory>
class Box {
public:
  std::shared_ptr<int> ptr;
  Box(int value) : ptr {new int {value}} { }
  Box(const Box& b) : ptr {b.ptr} { }
};
```

```
int main(void) {
  Box b1 { 10 };
  Box b2 { b1 };
  Box b3 { 11 };
  b3 = b2;
}
```



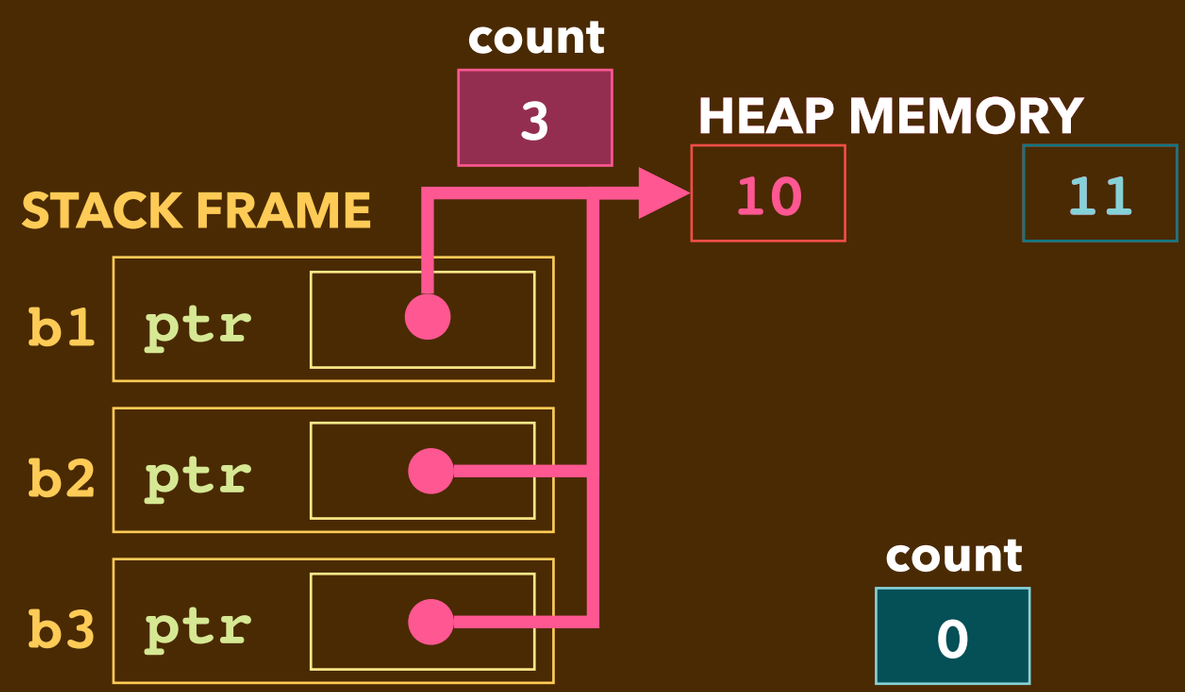
A SHARED_PTR BOX

```
#include <memory>
class Box {
public:
  std::shared_ptr<int> ptr;
  Box(int value) : ptr {new int {value}} { }
  Box(const Box& b) : ptr {b.ptr} { }
};
```

```
int main(void) {
  Box b1 { 10 };
  Box b2 { b1 };
  Box b3 { 11 };
  b3 = b2;
}
```

b1.ptr count increments to 3

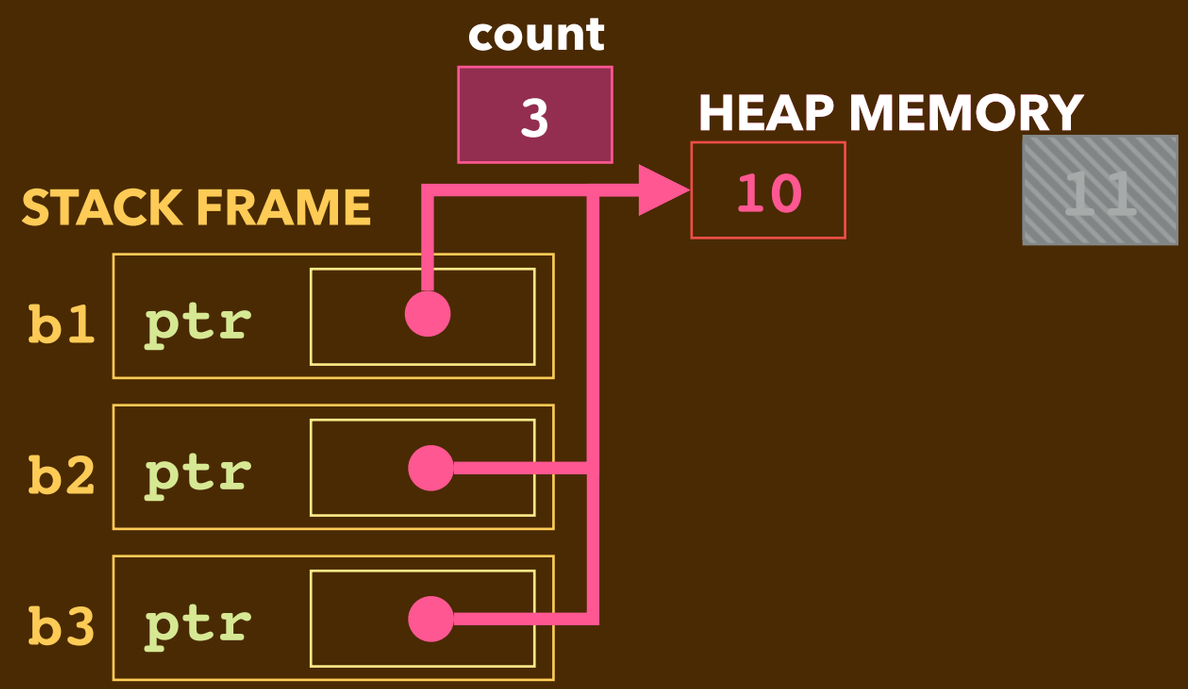
old b3.ptr decrements to 0



A SHARED_PTR BOX

```
#include <memory>
class Box {
public:
  std::shared_ptr<int> ptr;
  Box(int value) : ptr {new int {value}} { }
  Box(const Box& b) : ptr {b.ptr} { }
};
```

```
int main(void) {
  Box b1 { 10 };
  Box b2 { b1 };
  Box b3 { 11 };
  b3 = b2;
}
```



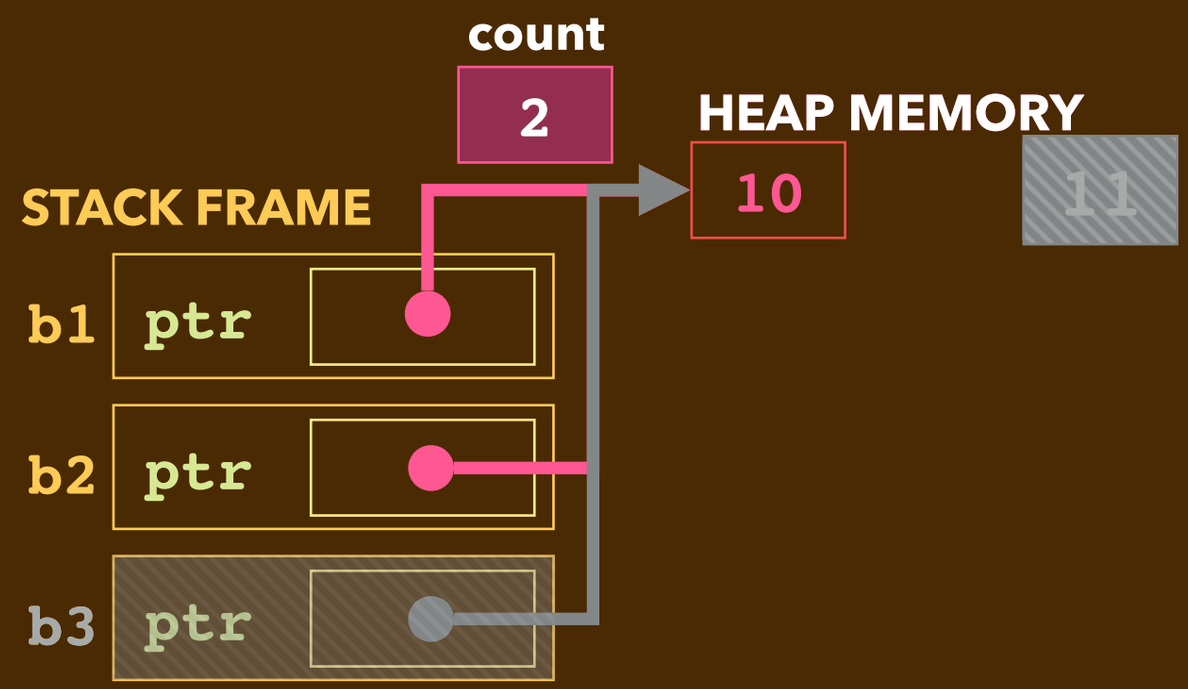
old b3.ptr 's raw pointer is deleted

A SHARED_PTR BOX

```
#include <memory>
class Box {
public:
  std::shared_ptr<int> ptr;
  Box(int value) : ptr {new int {value}} { }
  Box(const Box& b) : ptr {b.ptr} { }
};
```

```
int main(void) {
  Box b1 { 10 };
  Box b2 { b1 };
  Box b3 { 11 };
  b3 = b2;
}
```

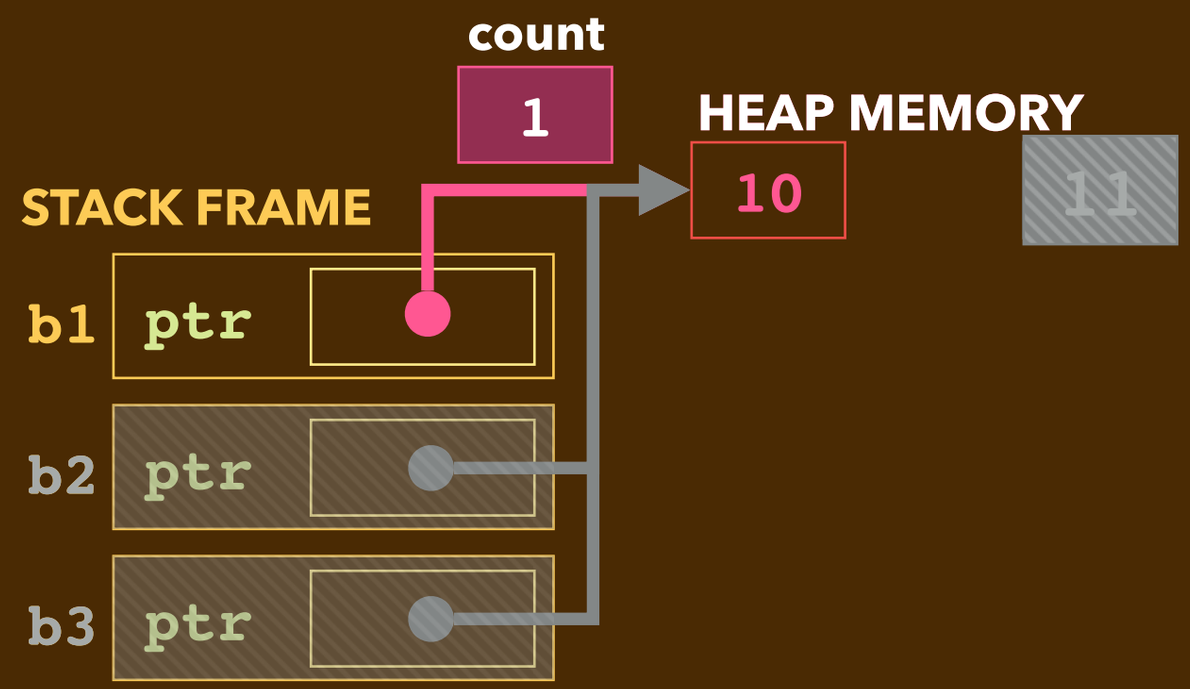
Box b3 loses scope; count decrements.



A SHARED_PTR BOX

```
#include <memory>
class Box {
public:
  std::shared_ptr<int> ptr;
  Box(int value) : ptr {new int {value}} { }
  Box(const Box& b) : ptr {b.ptr} { }
};
```

```
int main(void) {
  Box b1 { 10 };
  Box b2 { b1 };
  Box b3 { 11 };
  b3 = b2;
}
```



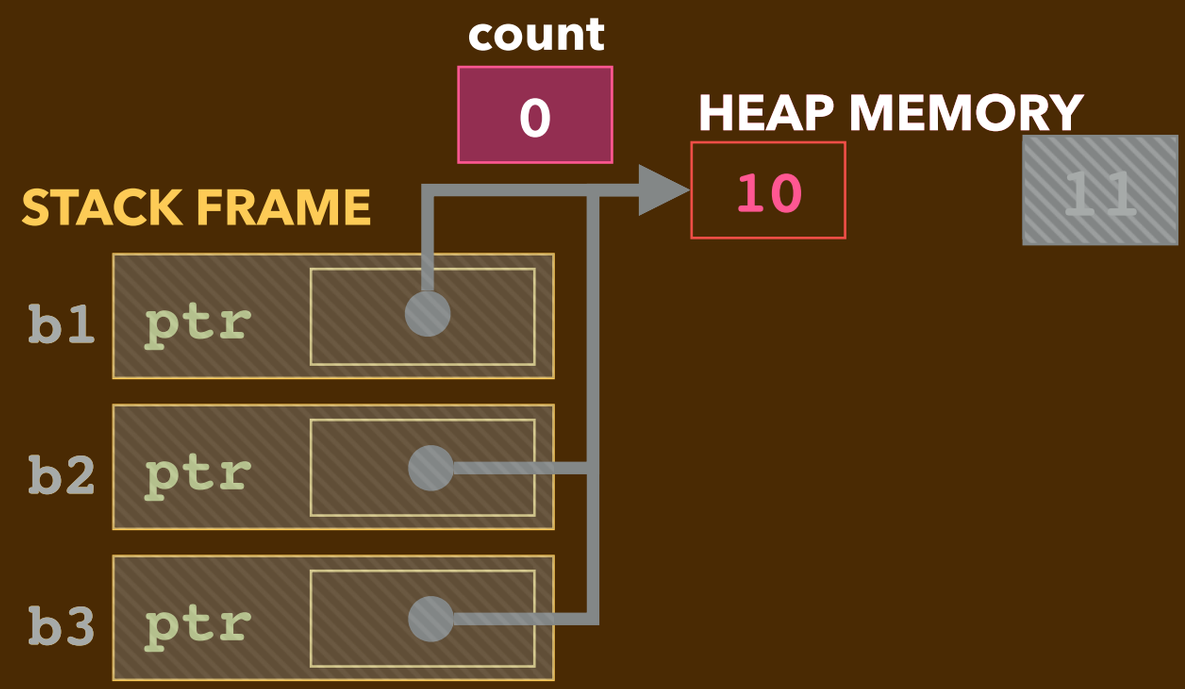
Box b3 loses scope; count decrements.

Box b2 loses scope; count decrements.

A SHARED_PTR BOX

```
#include <memory>
class Box {
public:
  std::shared_ptr<int> ptr;
  Box(int value) : ptr {new int {value}} { }
  Box(const Box& b) : ptr {b.ptr} { }
};
```

```
int main(void) {
  Box b1 { 10 };
  Box b2 { b1 };
  Box b3 { 11 };
  b3 = b2;
}
```

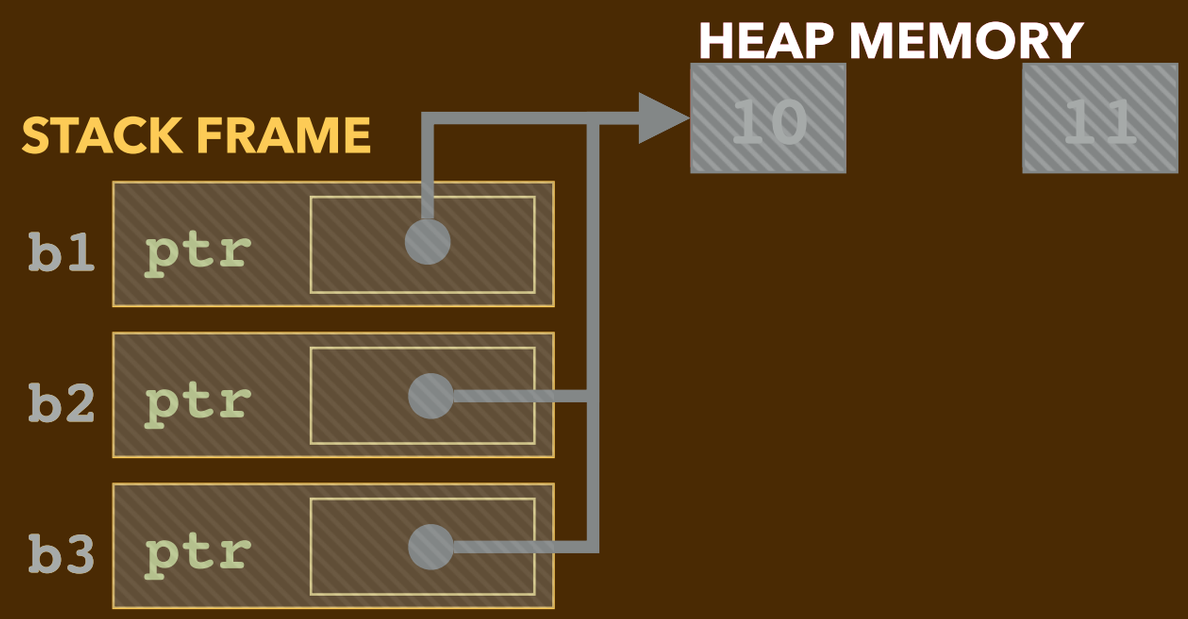


Box b3 loses scope; count decrements.
Box b2 loses scope; count decrements.
Box b1 loses scope; count decrements.

A SHARED_PTR BOX

```
#include <memory>
class Box {
public:
  std::shared_ptr<int> ptr;
  Box(int value) : ptr {new int {value}} { }
  Box(const Box& b) : ptr {b.ptr} { }
};
```

```
int main(void) {
  Box b1 { 10 };
  Box b2 { b1 };
  Box b3 { 11 };
  b3 = b2;
}
```



Box b3 loses scope; count decrements.
Box b2 loses scope; count decrements.
Box b1 loses scope; count decrements. The raw pointer b1.ptr is deleted.

LINKED LISTS USING SMART POINTERS

- This lecture's **samples** folder has three working implementations, each using `shared_ptr`:
  - A singly-linked list where `delete` is never explicitly called.
  - A doubly-linked list whose destructor unlinks the "back" pointers `prev`.
  - A doubly-linked list whose back pointers are of type `weak_ptr`.

A SINGLY LINKED LIST WITH RAW POINTERS

```
#include <memory>

class node {
public:
  int data;
  node* next;
  node(int value) : data {value}, next {nullptr} { }
 ~node(void) { }
};
```

```
class llist {
private:
  node* first;
  node* last;
public:
  llist(void) : first {nullptr}, last {nullptr} { }
 ~llist(void) { ... // traversal with delete of each    }
  void prepend(int value) { ... // new node as first    }
  void append(int value)  { ... // new node as last     }
  void remove(int value)  { ... // extract; delete node }
};
```

A SINGLY LINKED LIST WITH RAW POINTERS

```
#include <memory>

class node {
public:
  int data;
  node* next;
  node(int value) : data {value}, next {nullptr} { }
 ~node(void) { }
};
```

```
class llist {
private:
  node* first;
  node* last;
public:
  llist(void) : first {nullptr}, last {nullptr} { }
 ~llist(void) { ... // traversal with delete of each    }
  void prepend(int value) { ... // new node as first    }
  void append(int value)  { ... // new node as last     }
  void remove(int value)  { ... // extract; delete node }
};
```

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;
  }
}

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 each initialize their `shared_ptr` *count to 1.*

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;
  }
}

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 each initialize their `shared_ptr` *count to 1.*

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's 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's 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;
      }
    }
  }
}
```

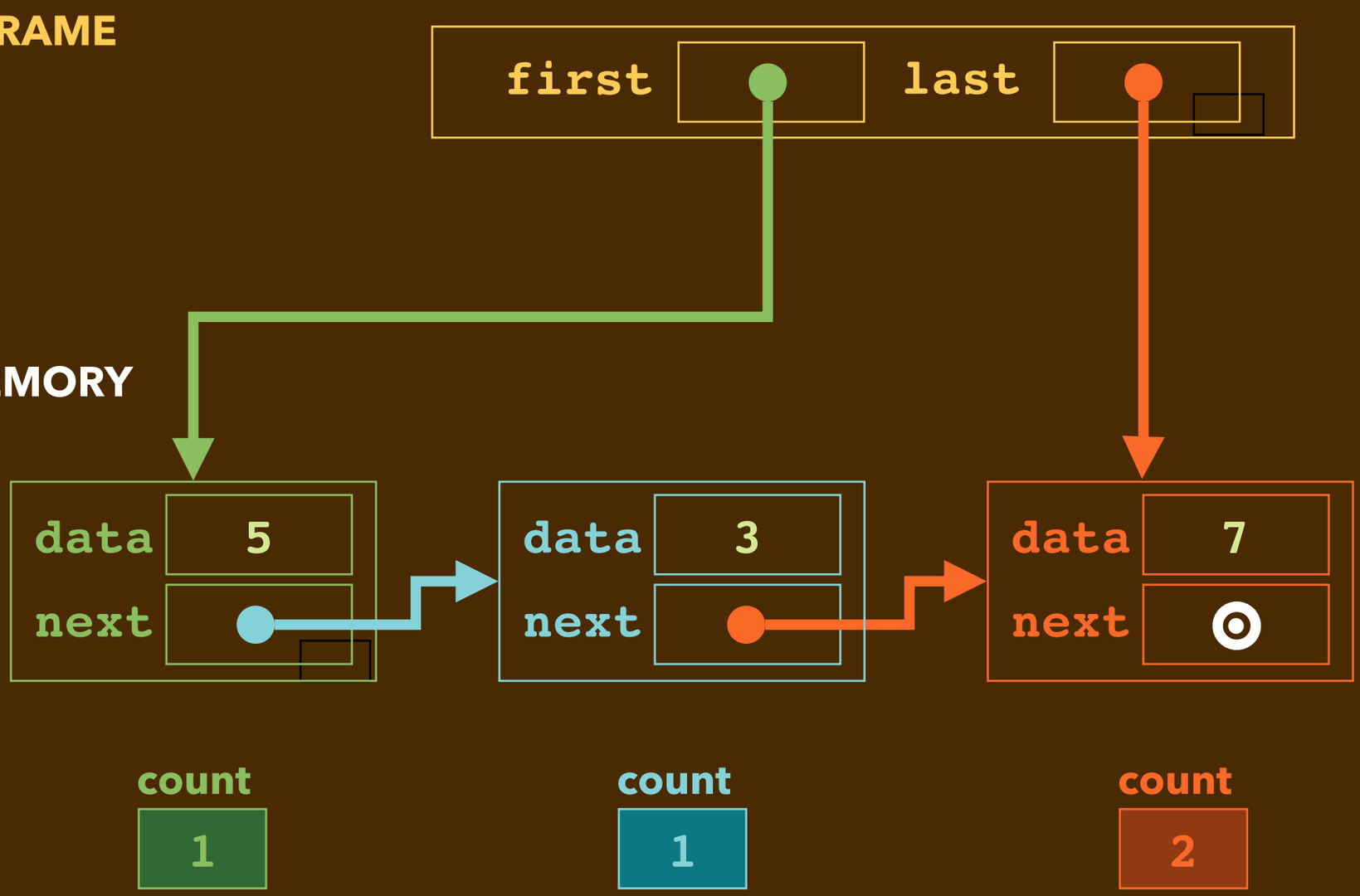
Here `current` loses scope; removed node's reference count goes to 0 and is reclaimed.

NO DESTRUCTOR CODE NEEDED

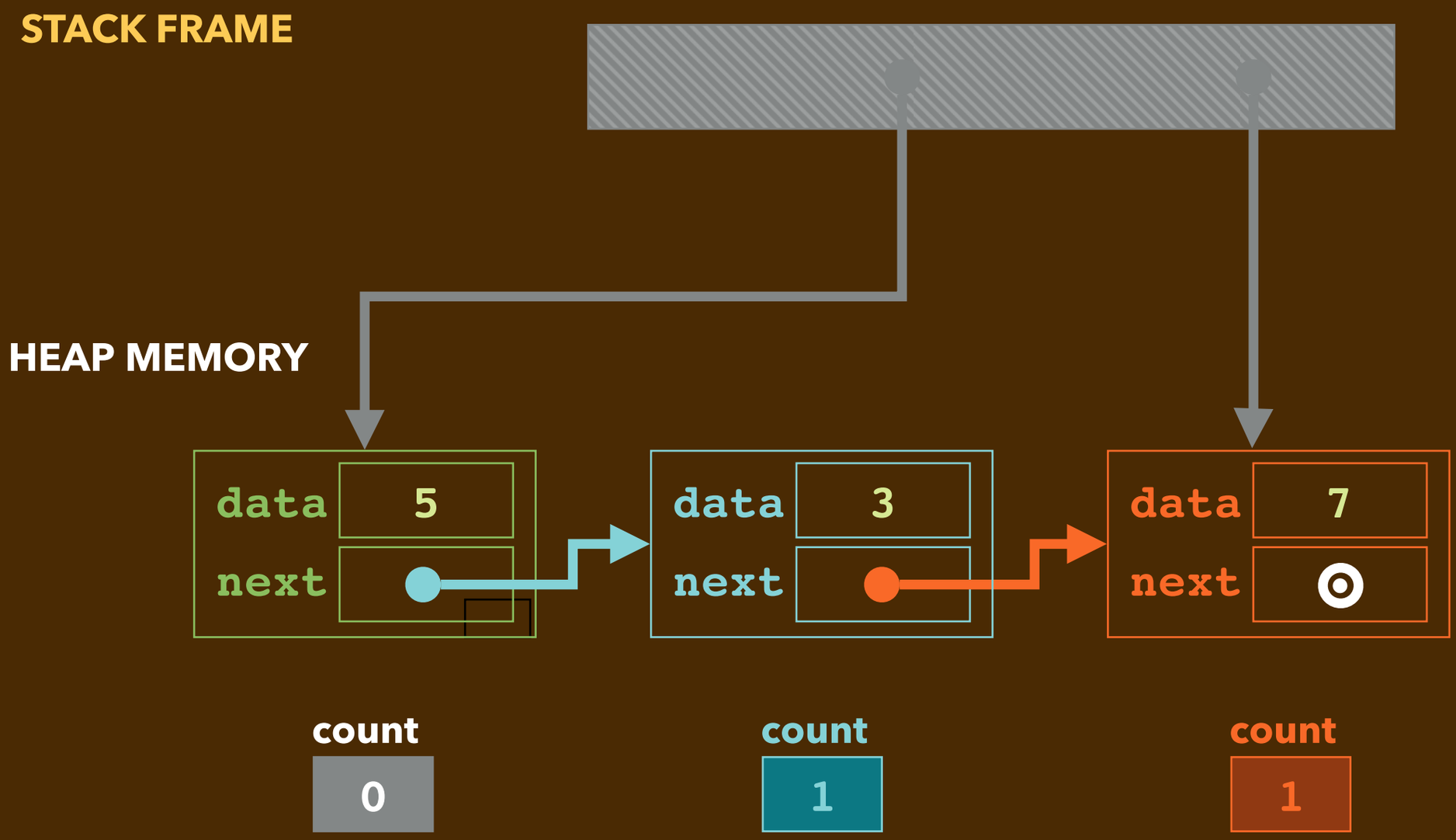
```
class node {
 int data;
  std::shared_ptr<node> next;
  node(int value) : data {value}, next {nullptr} { }
 ~node(void) { }
};
class llist {
  std::shared_ptr<node> first;
  std::shared_ptr<node> last;
  llist(void) : first {nullptr}, last {nullptr} { }
 ~llist(void) { // NOTHING HERE!! }
  void 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 **`node`**s.

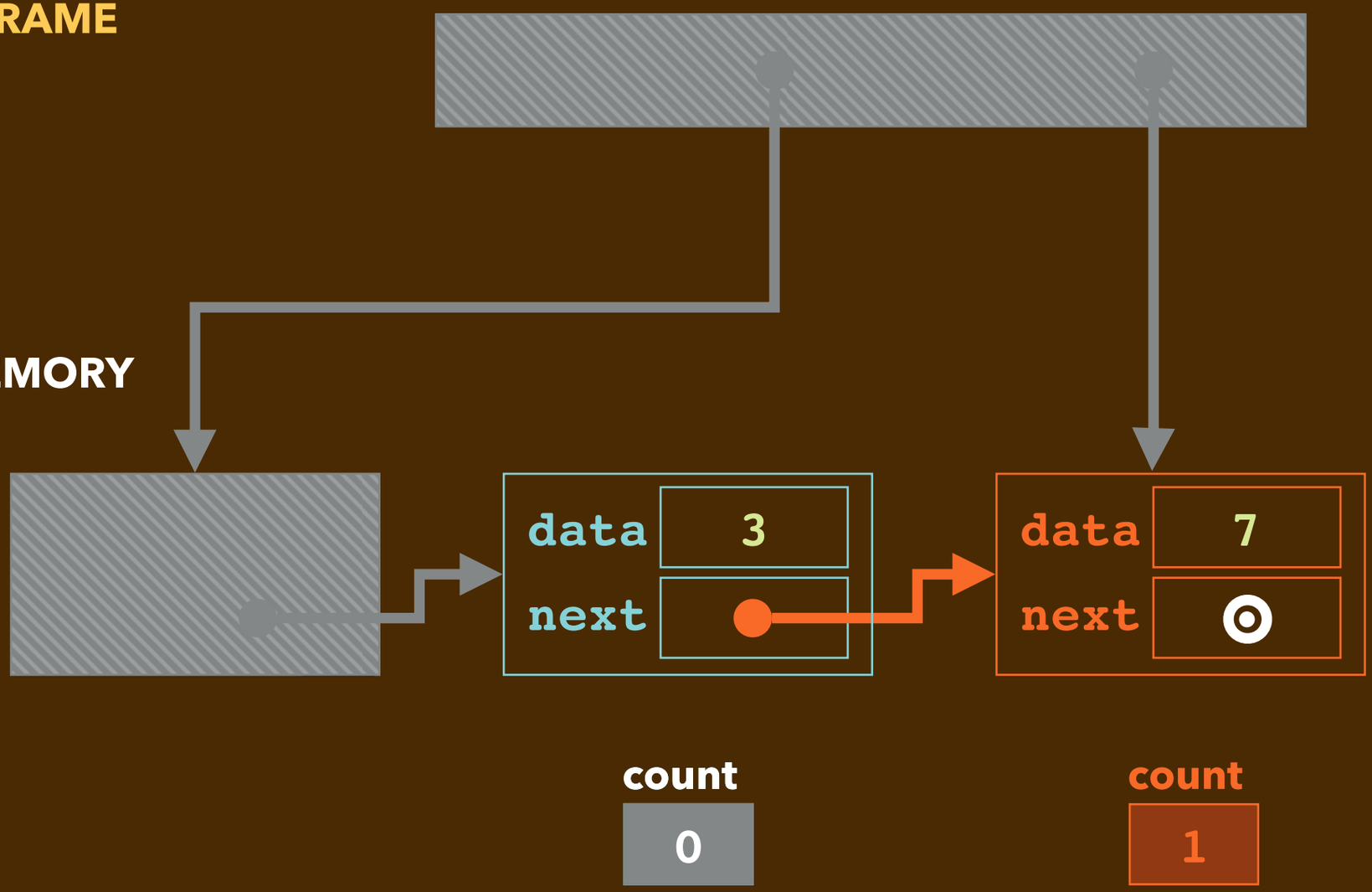
WHY IT WORKS: SINGLY LINKED LIST



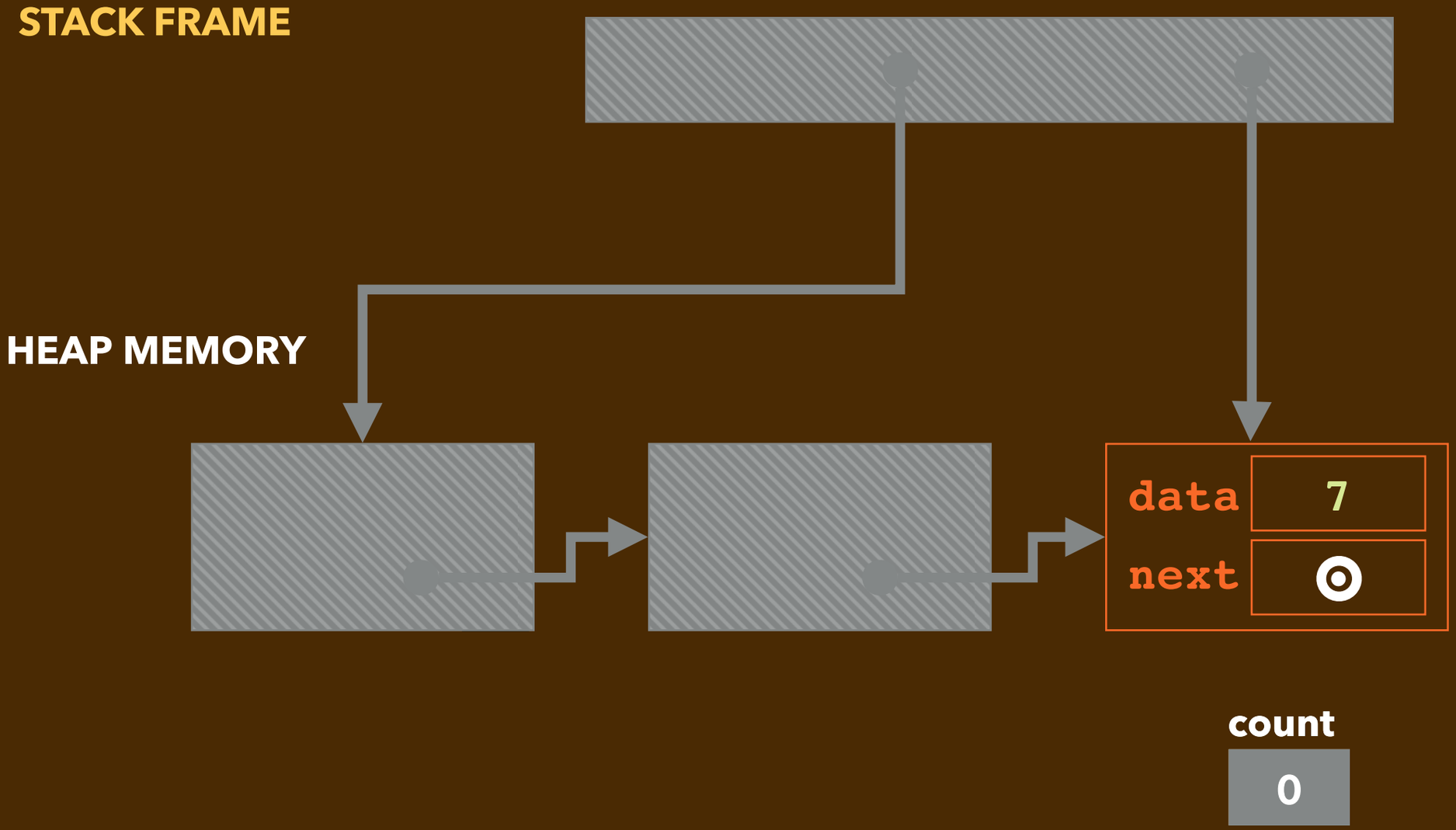
WHY IT WORKS: SINGLY LINKED LIST



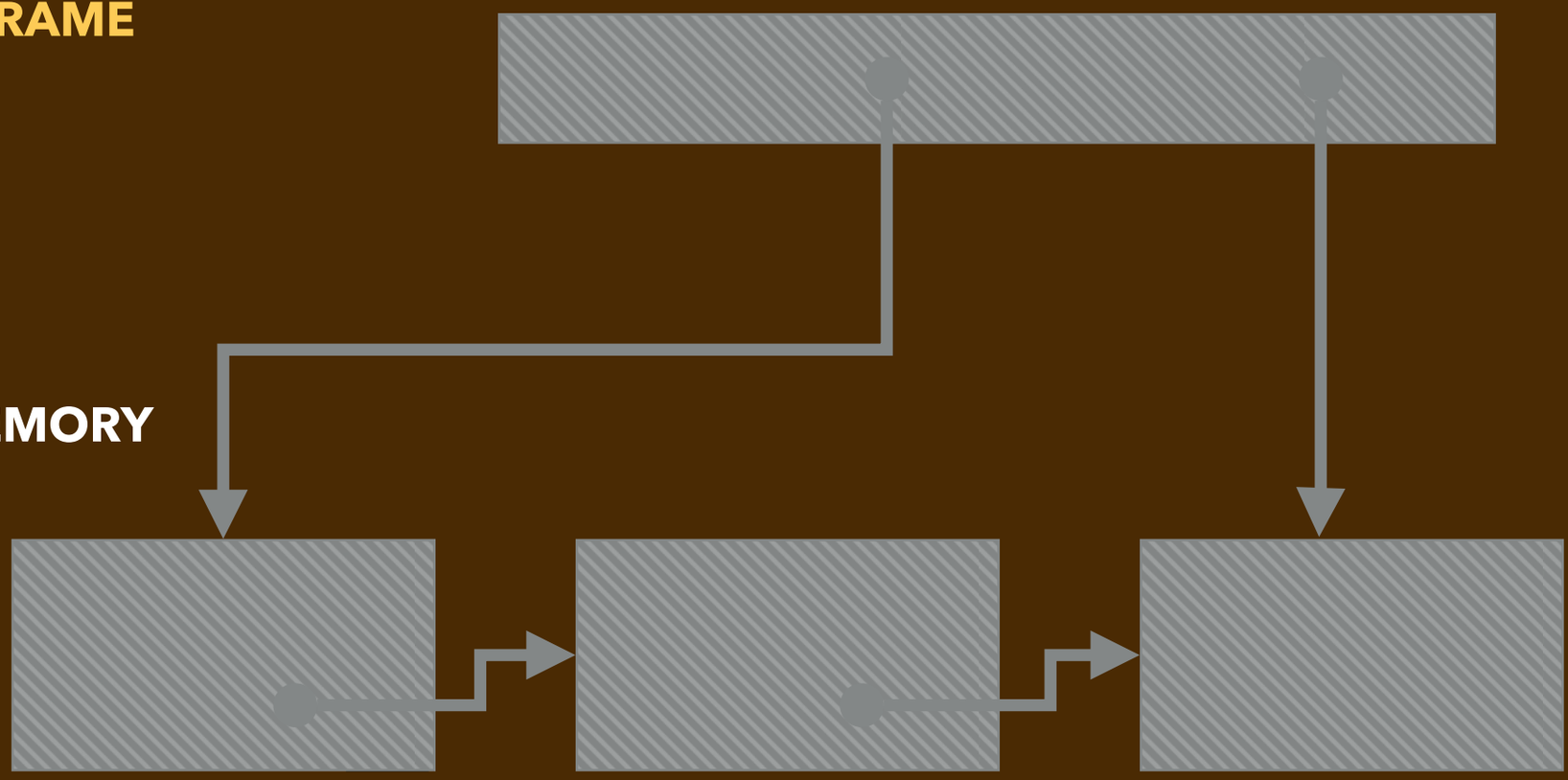
WHY IT WORKS: SINGLY LINKED LIST



WHY IT WORKS: SINGLY LINKED LIST



WHY IT WORKS: SINGLY LINKED LIST



A SHARED_PTR SINGLY LINKED LIST SUMMARY

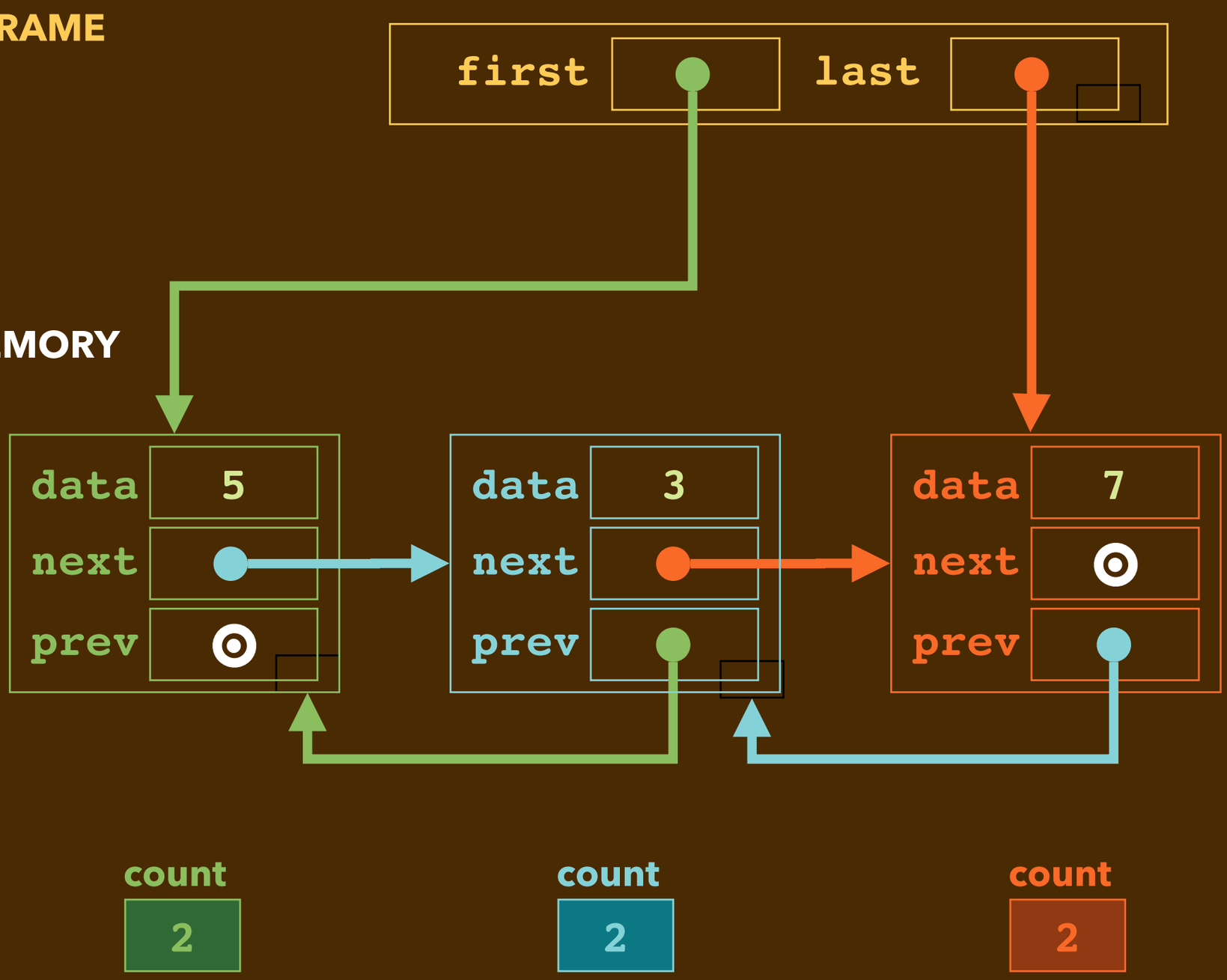
- By using **`shared_ptr`**, every reference to a node is counted.
- When a new node is made, a **`shared_ptr`** is invented with a count of 1.
  - It has an underlying raw pointer obtained from **`new`**.
- When a relink happens:
  - ➡ A non-null reference's count decrements.
  - ➡ Another reference's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

A SHARED_PTR DOUBLY LINKED LIST

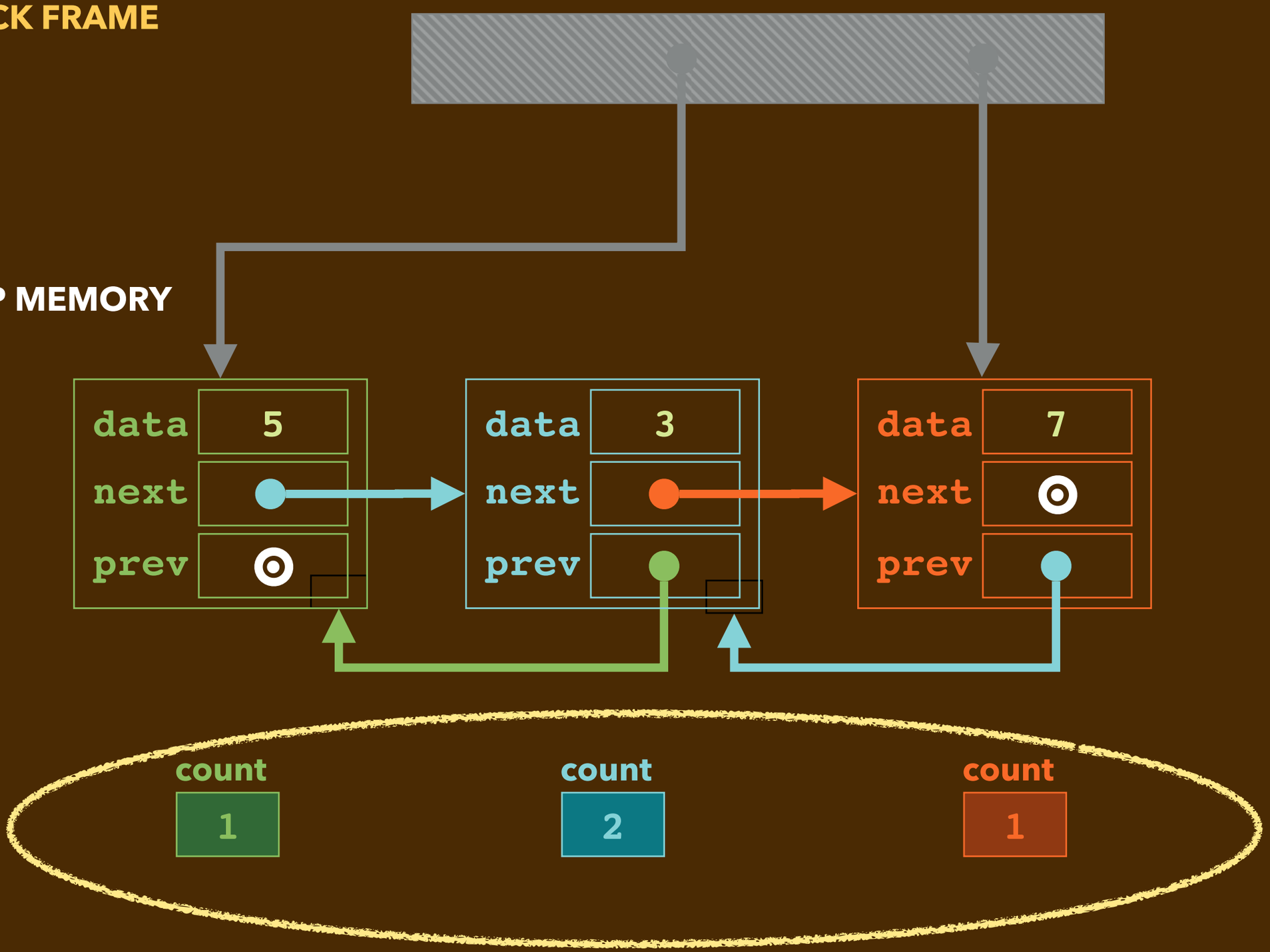
```
#include <memory>

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



WHY IT FAILS: DOUBLY LINKED LIST

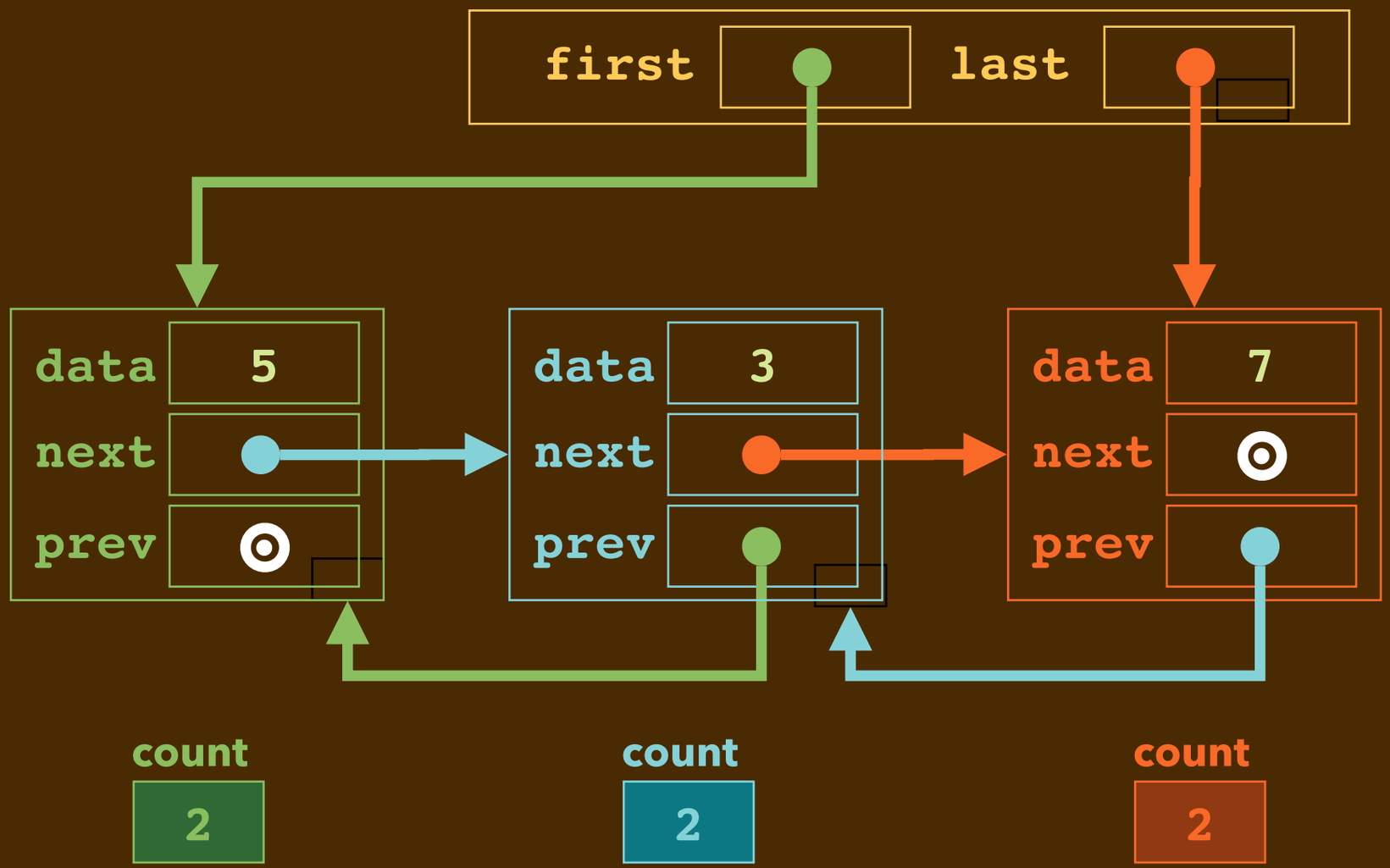


FIX #1: A DESTRUCTOR THAT UNLINKS PREV POINTERS

```
class dbllist {
private:
  std::shared_ptr<dnode> first;
  std::shared_ptr<dnode> 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);
}
```

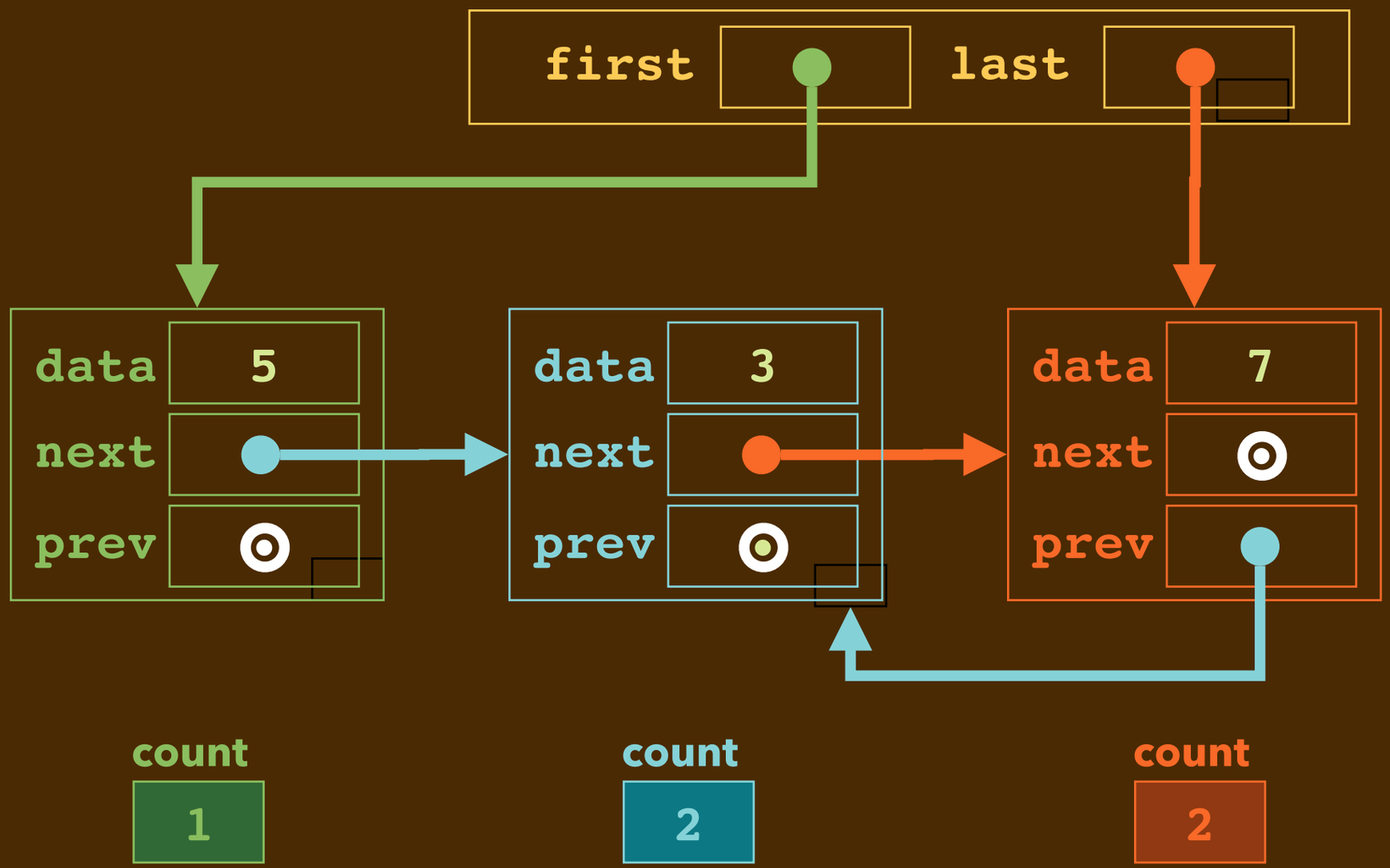
FIX #1: A DESTRUCTOR THAT UNLINKS EACH PREV

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



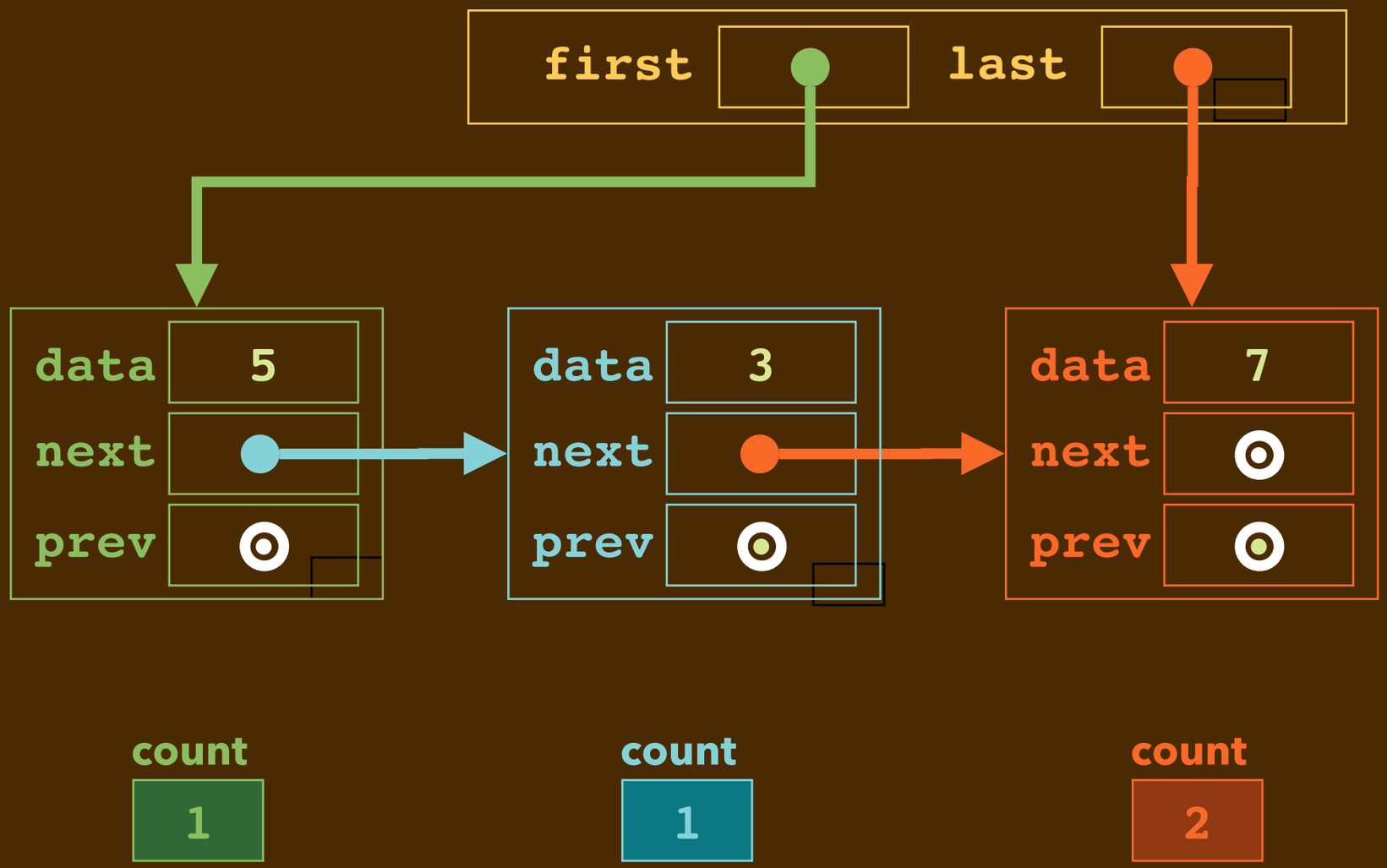
FIX #1: A DESTRUCTOR THAT UNLINKS EACH PREV

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



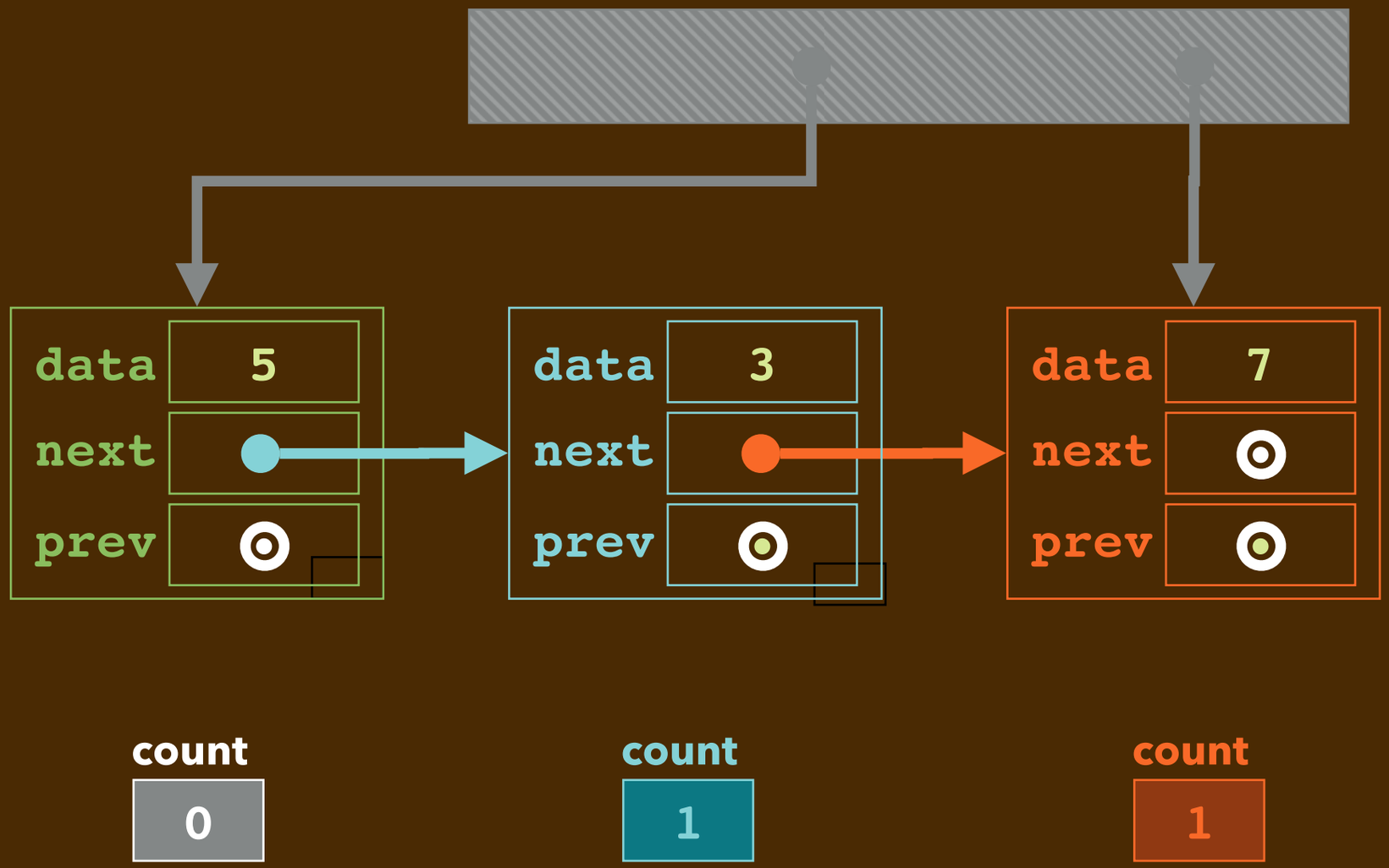
FIX #1: A DESTRUCTOR THAT UNLINKS EACH PREV

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



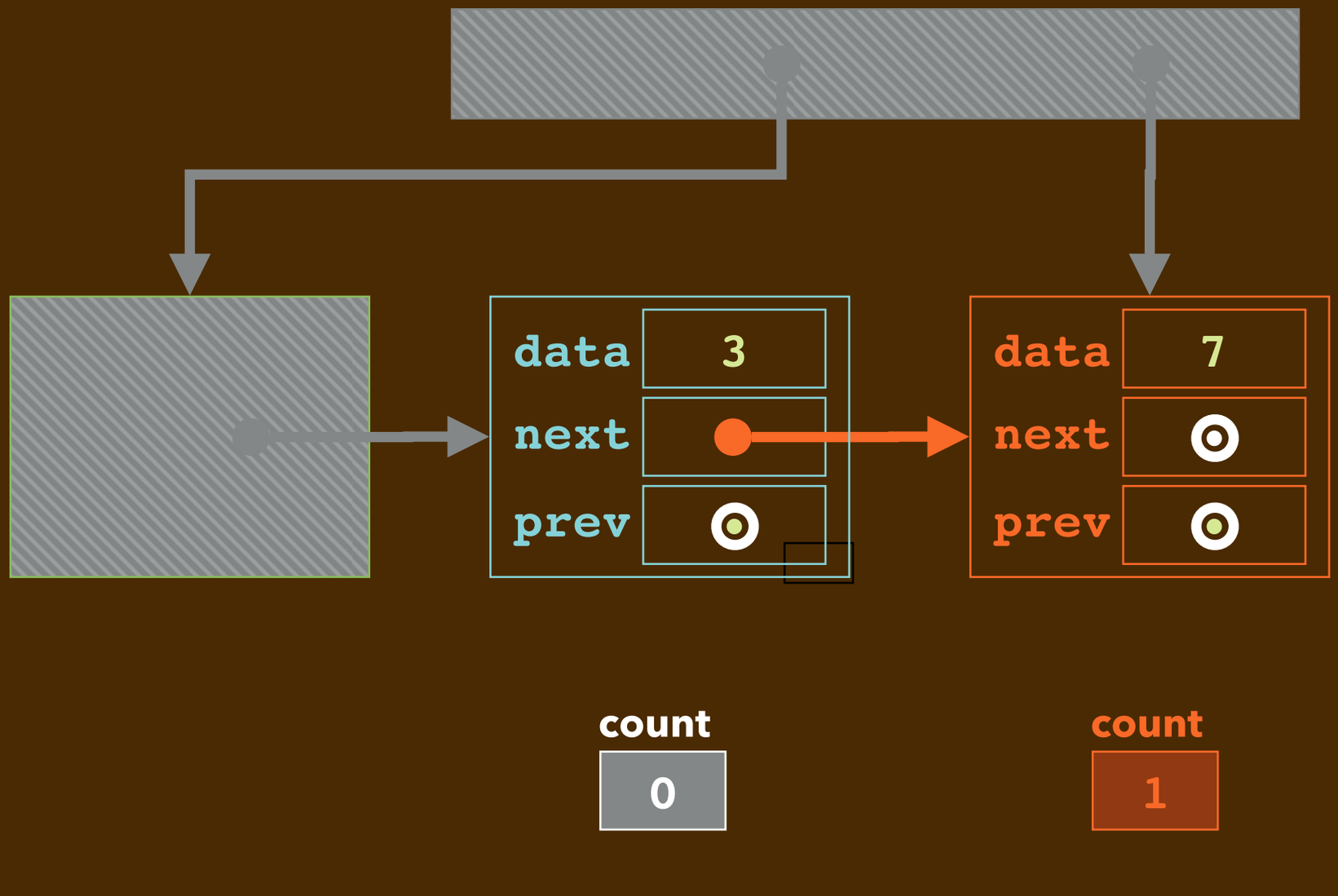
FIX #1: A DESTRUCTOR THAT UNLINKS EACH PREV

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



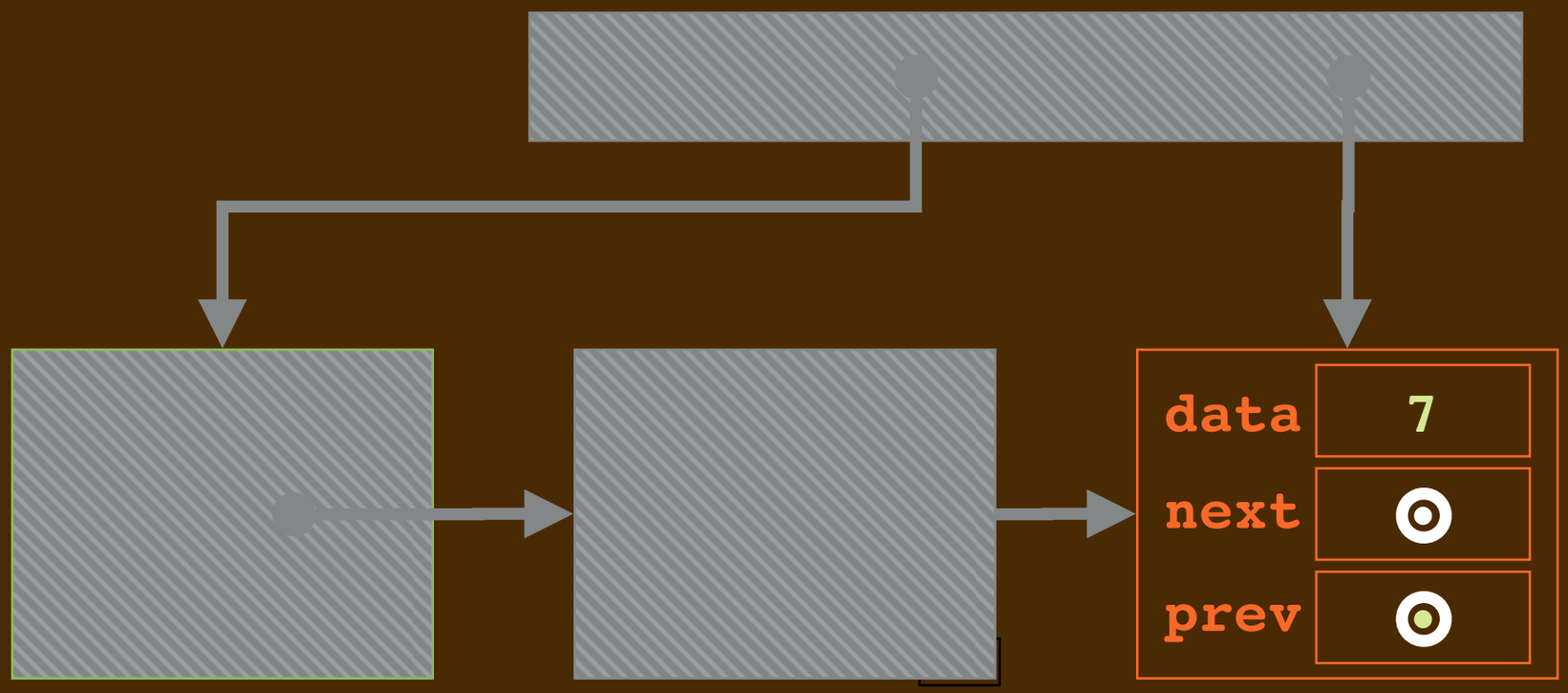
FIX #1: A DESTRUCTOR THAT UNLINKS EACH PREV

```
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;
  }
}
```

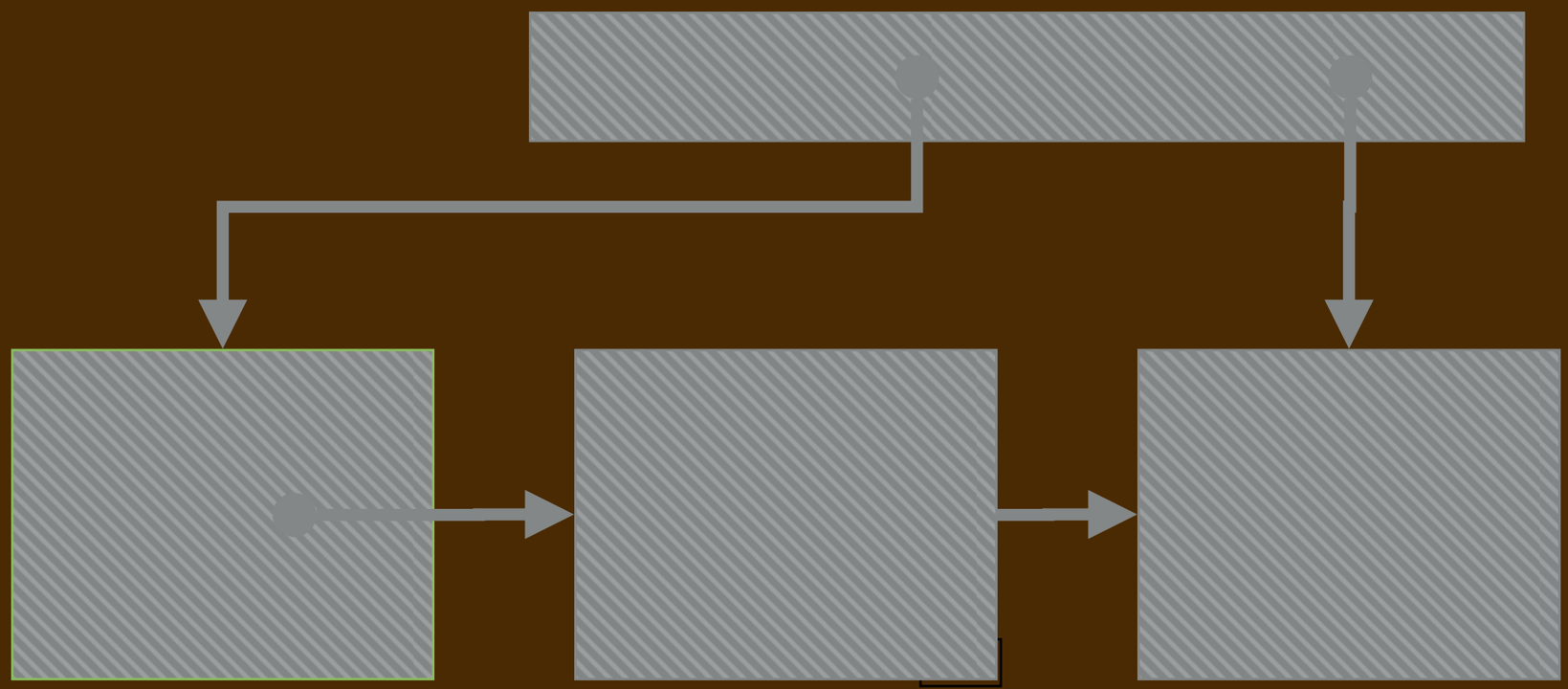


count

0

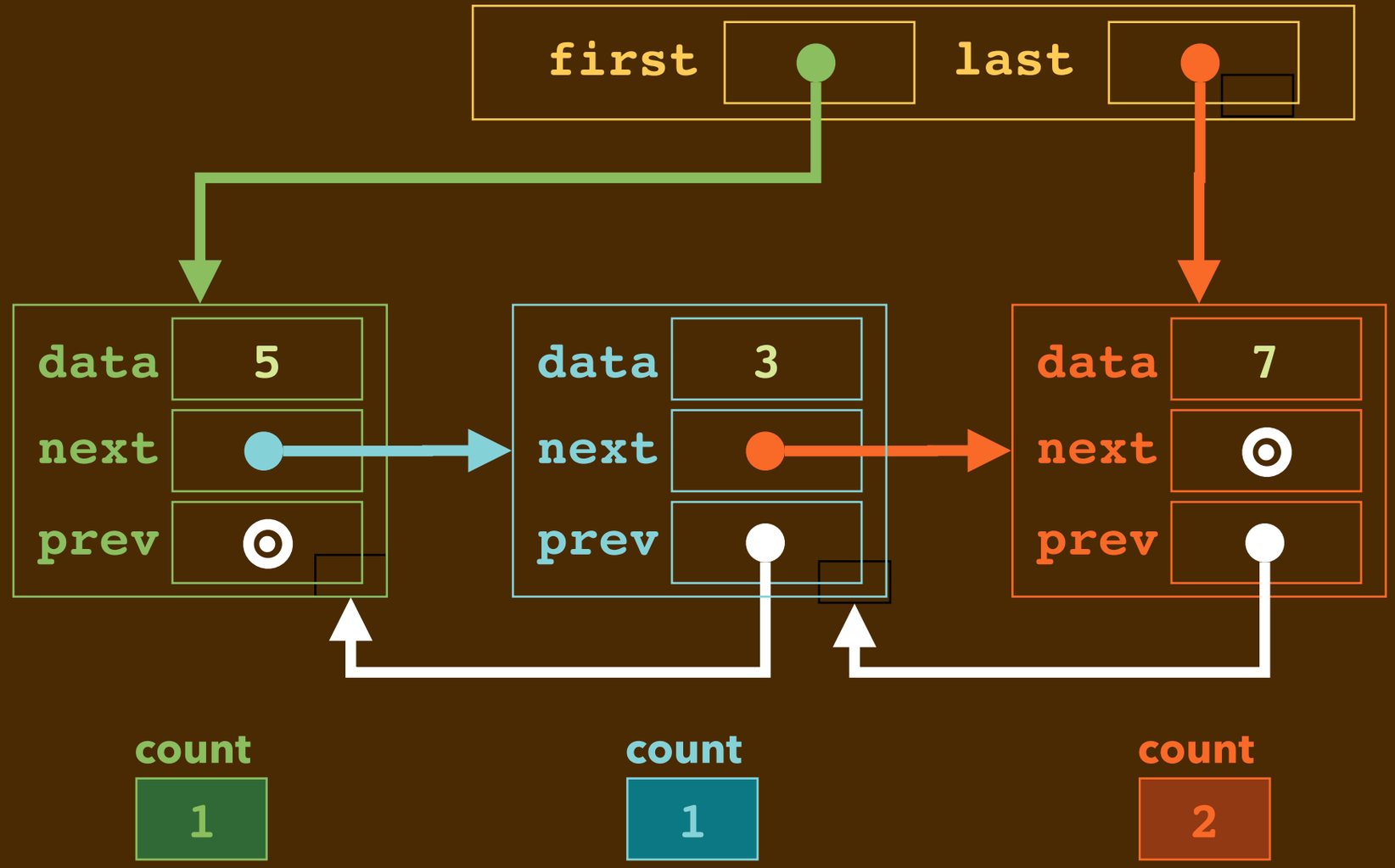
FIX #1: A DESTRUCTOR THAT UNLINKS EACH PREV

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



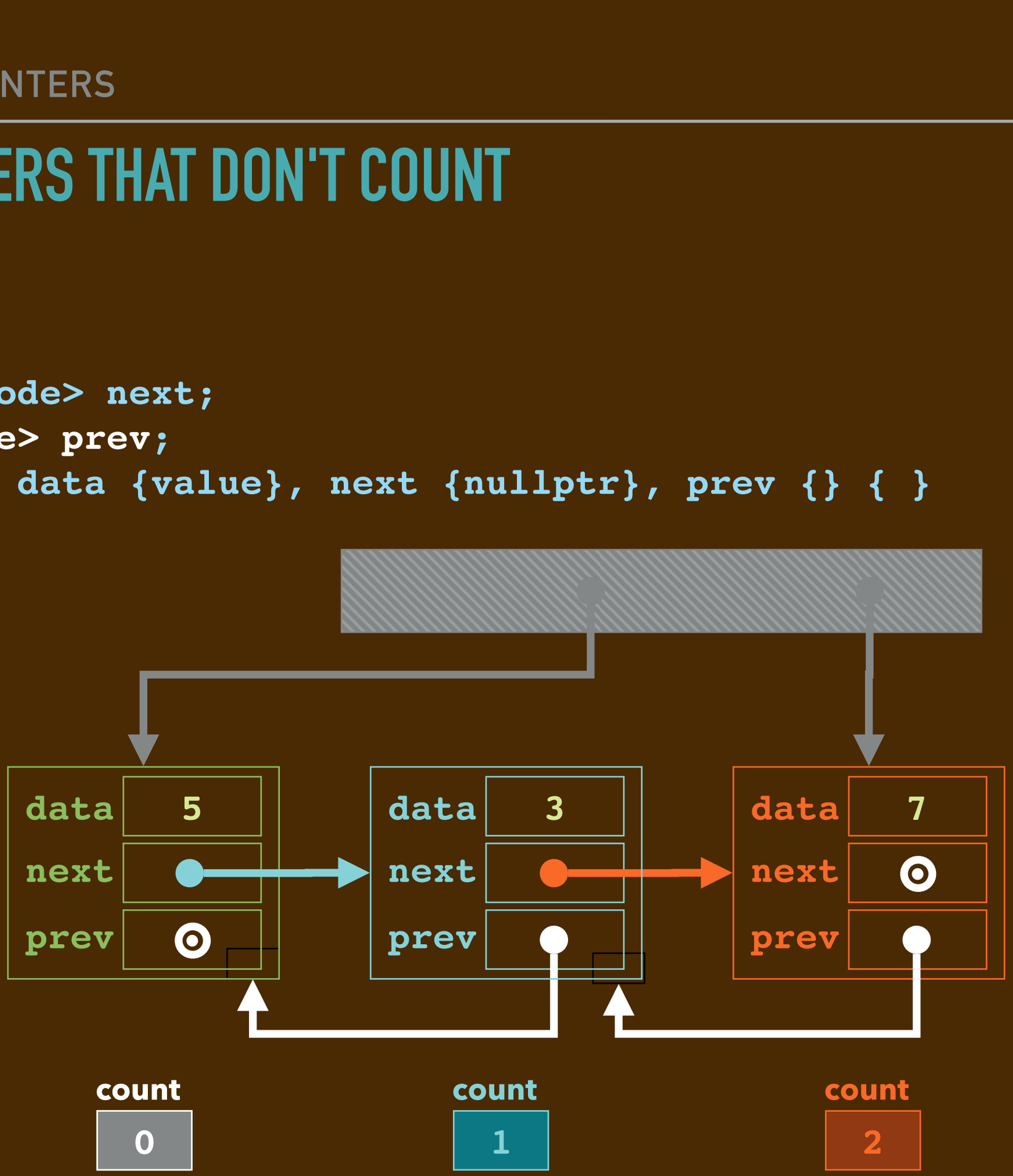
FIX #2: PREV POINTERS THAT DON'T COUNT

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



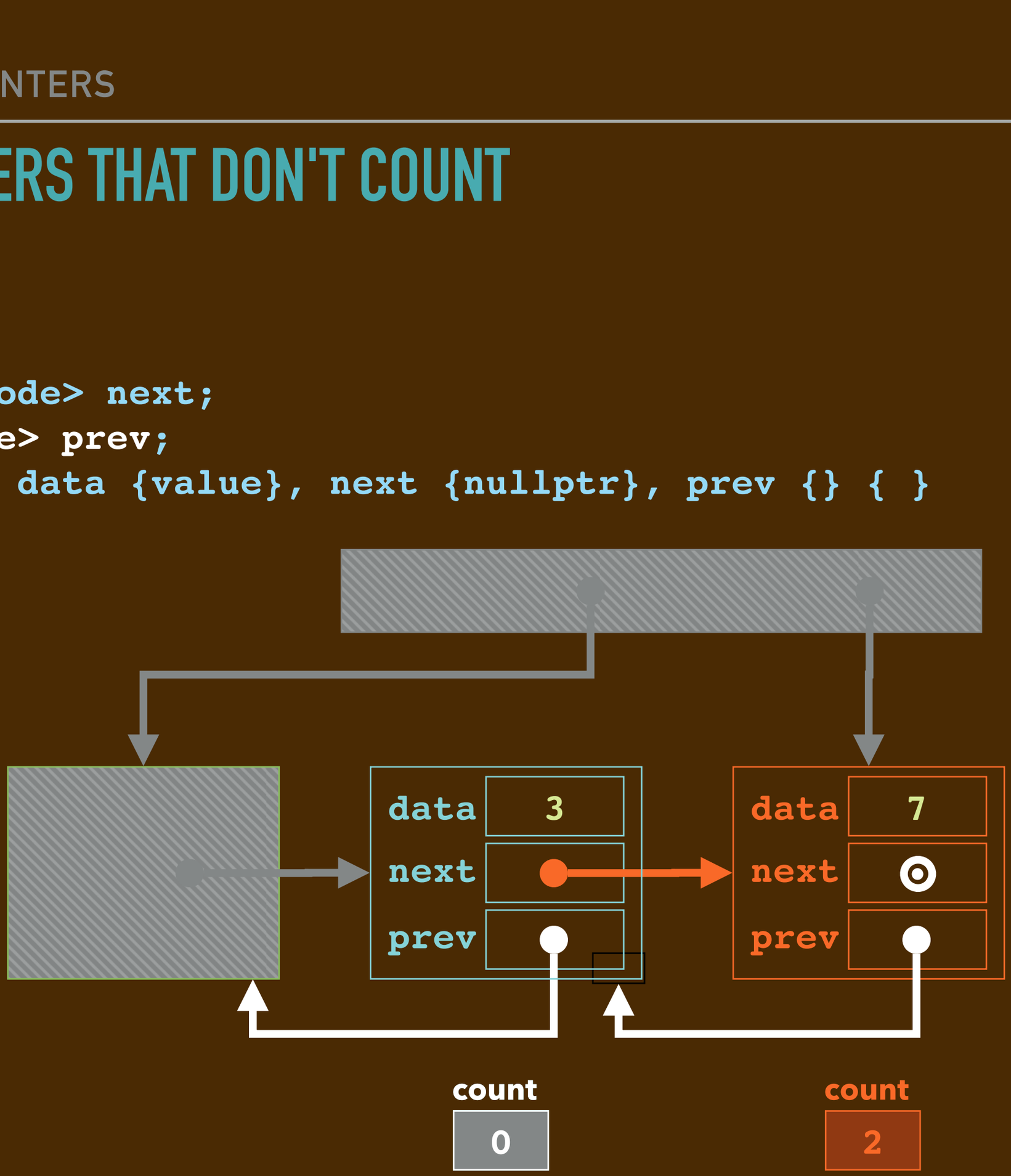
FIX #2: PREV POINTERS THAT DON'T COUNT

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



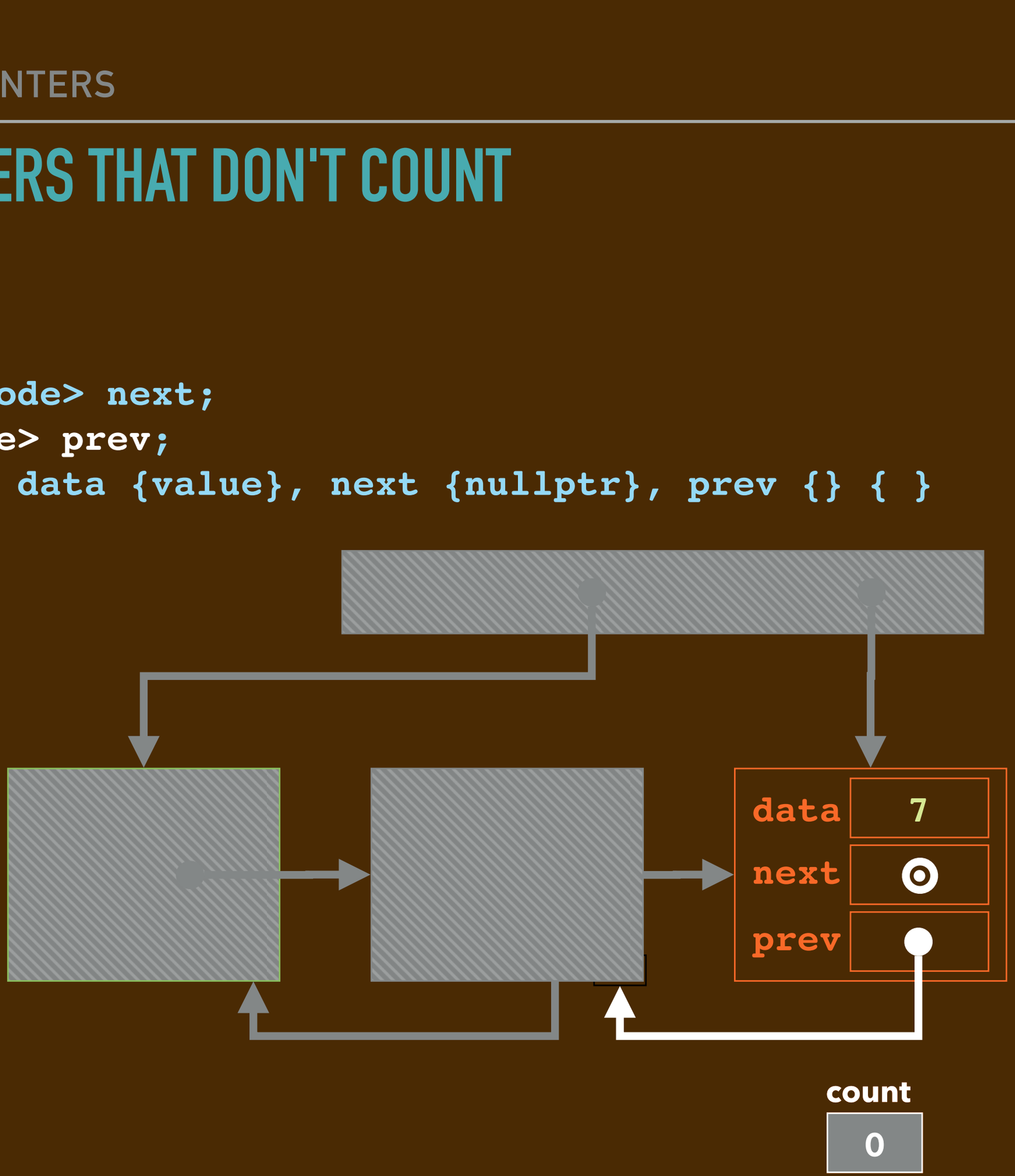
FIX #2: PREV POINTERS THAT DON'T COUNT

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



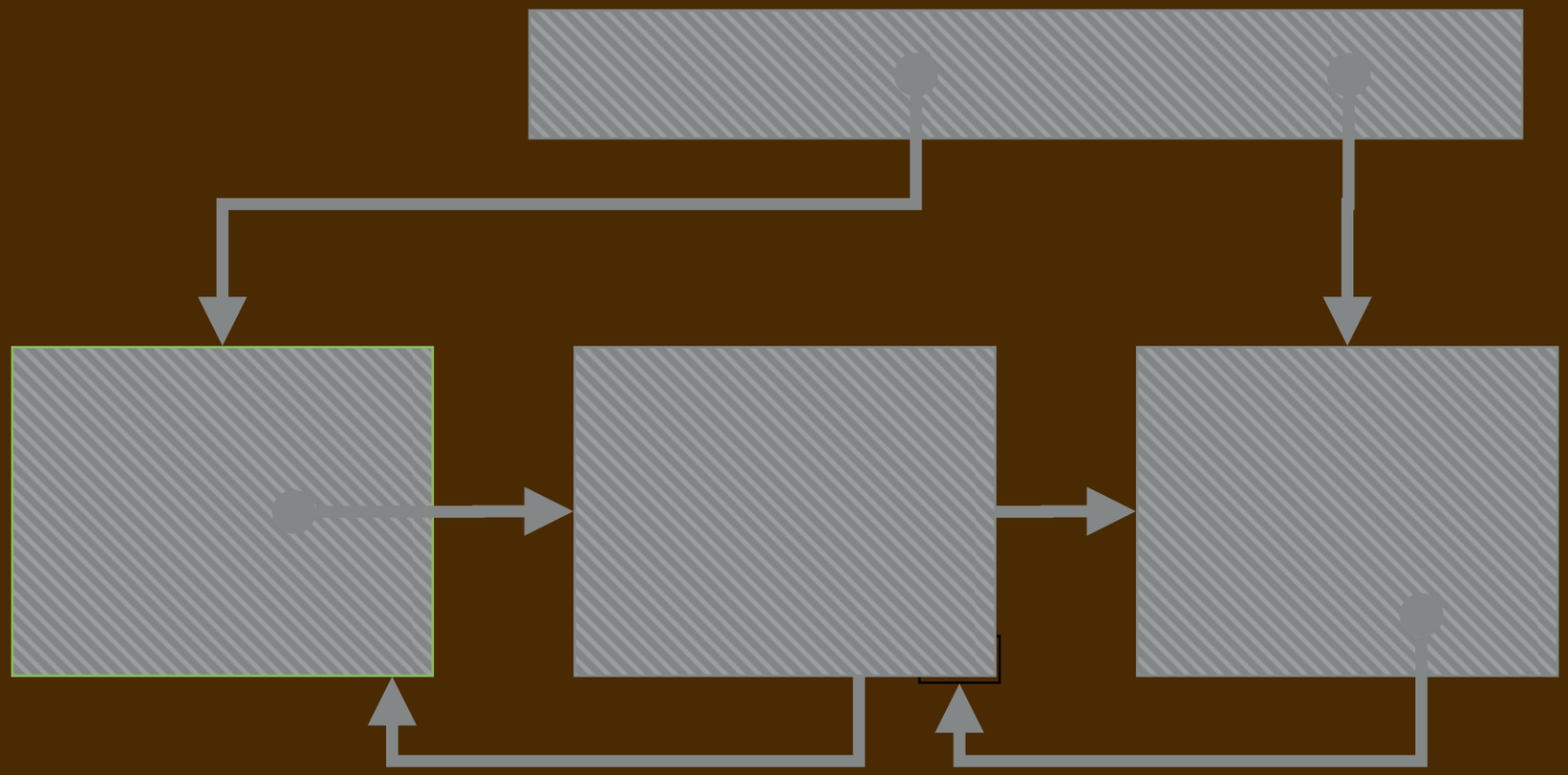
FIX #2: PREV POINTERS THAT DON'T COUNT

```
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`**:
  - **llist.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