SMART POINTERS

LECTURE 11-3

JIM FIX, REED COLLEGE CS2-S20

TODAY'S PLAN

- CLARIFICATION ABOUT `mutable` KEYWORD in *lambdas*
- LOOK AT MEMORY MANAGEMENT BUGS
- AUTOMATIC MEMORY MANAGEMENT *ANIMATED*
  - MARK-AND-SWEEP GARBAGE COLLECTION
  - REFERENCE COUNTS
- LOOK AT C++ STL's ***SMART POINTERS***
  - BOX EXAMPLE WITH `shared_ptr`
  - LINKED LIST EXAMPLE WITH `shared_ptr`

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
- (C++ makes the **`operator()`** method **`const`** otherwise.)
- 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
- (C++ makes the **operator()** method **const** otherwise.)
- 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
- (C++ makes the **operator()** method **const** otherwise.)
- 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
- (C++ makes the **`operator()`** method **`const`** otherwise.)
- 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
- (C++ makes the **`operator()`** method **`const`** otherwise.)
- 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
- (C++ makes the **operator()** method **const** otherwise.)
- 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
- (C++ makes the **operator()** method **const** otherwise.)
- 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
```

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
- (C++ makes the **operator()** method **const** otherwise.)
- 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*.

SMART POINTERS

LECTURE 11-3

JIM FIX, REED COLLEGE CS2-S20

SOME WAYS OF HAVING POINTER-BASED CODE BREAK

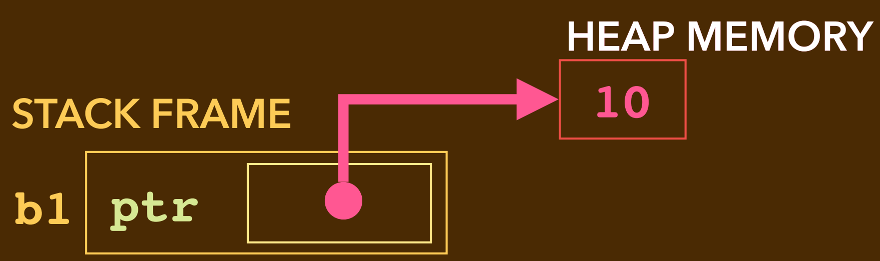
- You use **&** on a stack variable/object in a function and that pointer gets exported outside a function. You try to access the component referenced by that pointer outside that function.
- You forget to initialize a pointer component. You access that component.
- You try to access a component referenced 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***

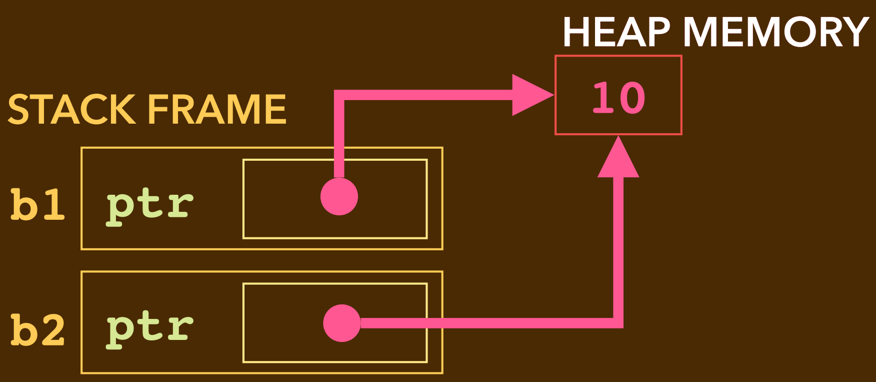
A BUGGY BOX

```
class Box {
public:
  int* ptr;
  Box(int value) : {new int[value]} { }
  Box(const Box& b) : {b.ptr} { }
  Box& operator=(const Box& b) { ptr = b.ptr; return *this; }
 ~Box(void) { delete ptr; }
};

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



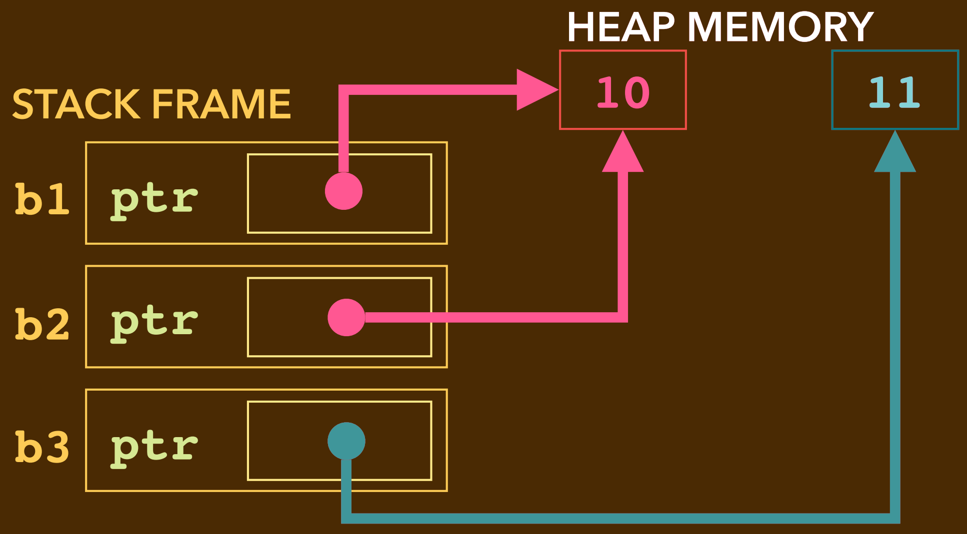
A BUGGY BOX

```
class Box {
public:
  int* ptr;
  Box(int value) : {new int[value]} { }
  Box(const Box& b) : {b.ptr} { }
  Box& operator=(const Box& b) { ptr = b.ptr; return *this; }
 ~Box(void) { delete ptr; }
};

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



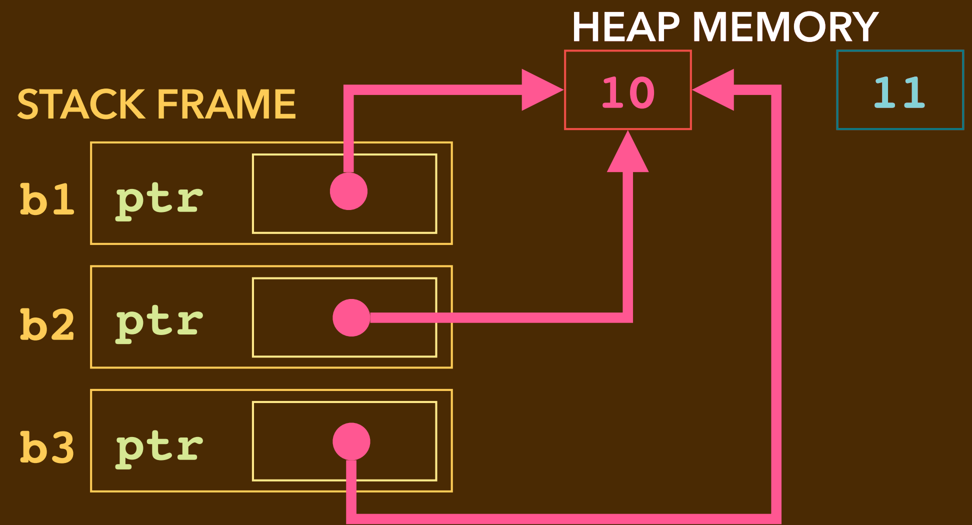
A BUGGY BOX

```
class Box {
public:
  int* ptr;
  Box(int value) : {new int[value]} { }
  Box(const Box& b) : {b.ptr} { }
  Box& operator=(const Box& b) { ptr = b.ptr; return *this; }
 ~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) : {new int[value]} { }
  Box(const Box& b) : {b.ptr} { }
  Box& operator=(const Box& b) { ptr = b.ptr; return *this; }
 ~Box(void) { delete ptr; }
};

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



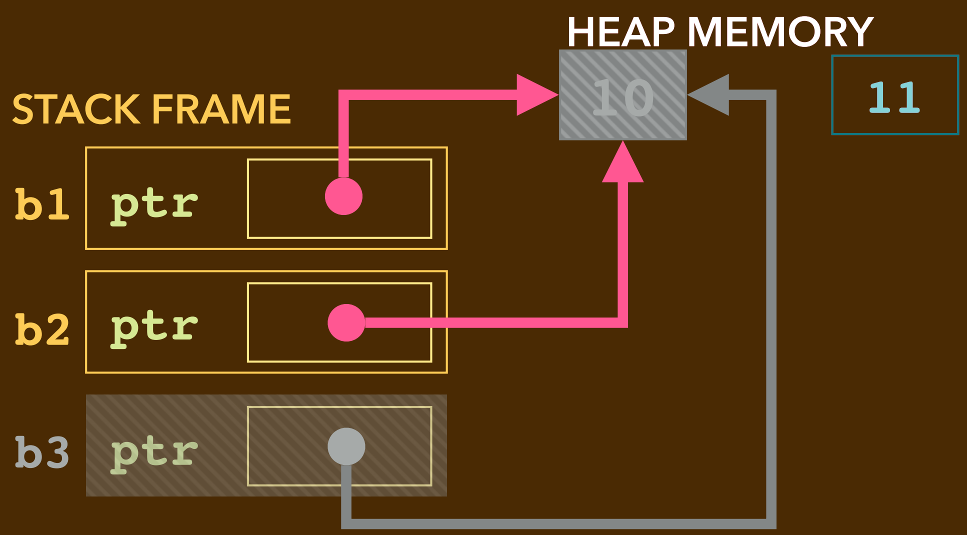
Changed reference in b3.

This is a "memory leak" error.

A BUGGY BOX

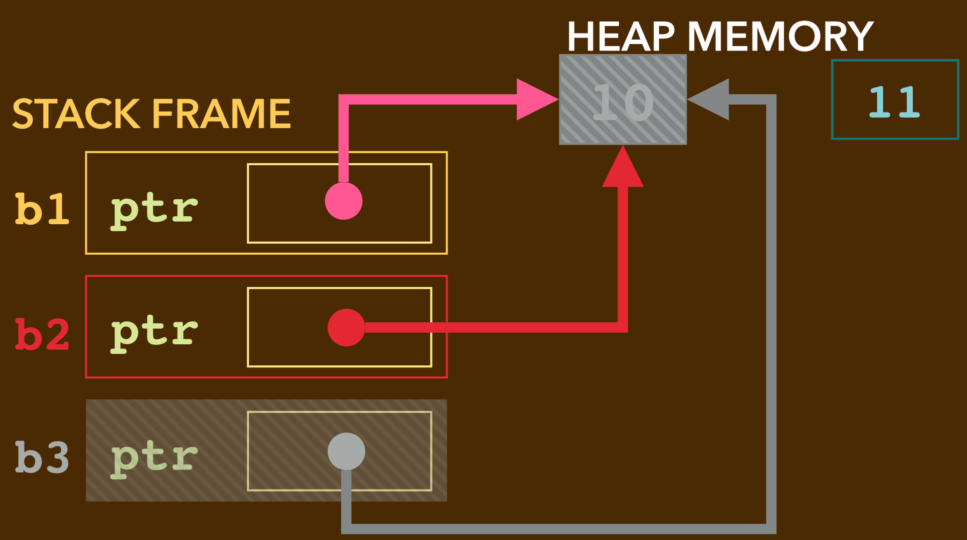
```
class Box {
public:
  int* ptr;
  Box(int value) : {new int[value]} { }
  Box(const Box& b) : {b.ptr} { }
  Box& operator=(const Box& b) { ptr = b.ptr; return *this; }
 ~Box(void) { delete ptr; }
};

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) : {new int[value]} { }
   Box(const Box& b) : {b.ptr} { }
   Box& operator=(const Box& b) { ptr = b.ptr; return *this; }
  ~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.
- Practitioners and researchers are heavily devoted to finding and preventing these bugs.

BUGS BUGS BUGS

- These kinds of bugs are so rampant in C++ code.
- Many of them go unnoticed.
- Practitioners and researchers are heavily devoted to finding and preventing these bugs.
  - ➡Programming disciplines/idioms developed

BUGS BUGS BUGS

- These kinds of bugs are so rampant in C++ code.
- Many of them go unnoticed.
- Practictioners and researchers are heavily devoted to finding and preventing these bugs.
  - Programming disciplines/idioms developed
  - Programming libraries developed

BUGS BUGS BUGS

- These kinds of bugs are so rampant in C++ code.
- Many of them go unnoticed.
- Practictioners and researchers are heavily devoted to finding and preventing these bugs.
  - ➡Programming disciplines/idioms developed
  - ➡Programming libraries developed
  - ➡Programming languages developed

BUGS BUGS BUGS

- These kinds of bugs are so rampant in C++ code.
- Many of them go unnoticed.
- Practictioners and researchers are heavily devoted to finding and preventing these bugs.
  - ➡Programming disciplines/idioms developed
  - ➡Programming libraries developed
  - ➡Programming languages developed
  - ➡Program analysis tools developed

BUGS BUGS BUGS

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

BUGS BUGS BUGS BUGS BUGS BUGS BUGS BUGS BUGS BUGS

- These kinds of bugs are so rampant in C++ code.
- Many of them go unnoticed.
- Practictioners and researchers are heavily devoted to finding and preventing these bugs.

Even messier with concurrency and sharing.

  - Programming disciplines/idioms developed
  - Programming libraries developed
  - Programming languages developed
  - Program analysis tools developed
  - Runtime instruments developed
  - Testing strategies developed.

2016 GÖDEL PRIZE

- From the *European Association for Thheoretical 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."

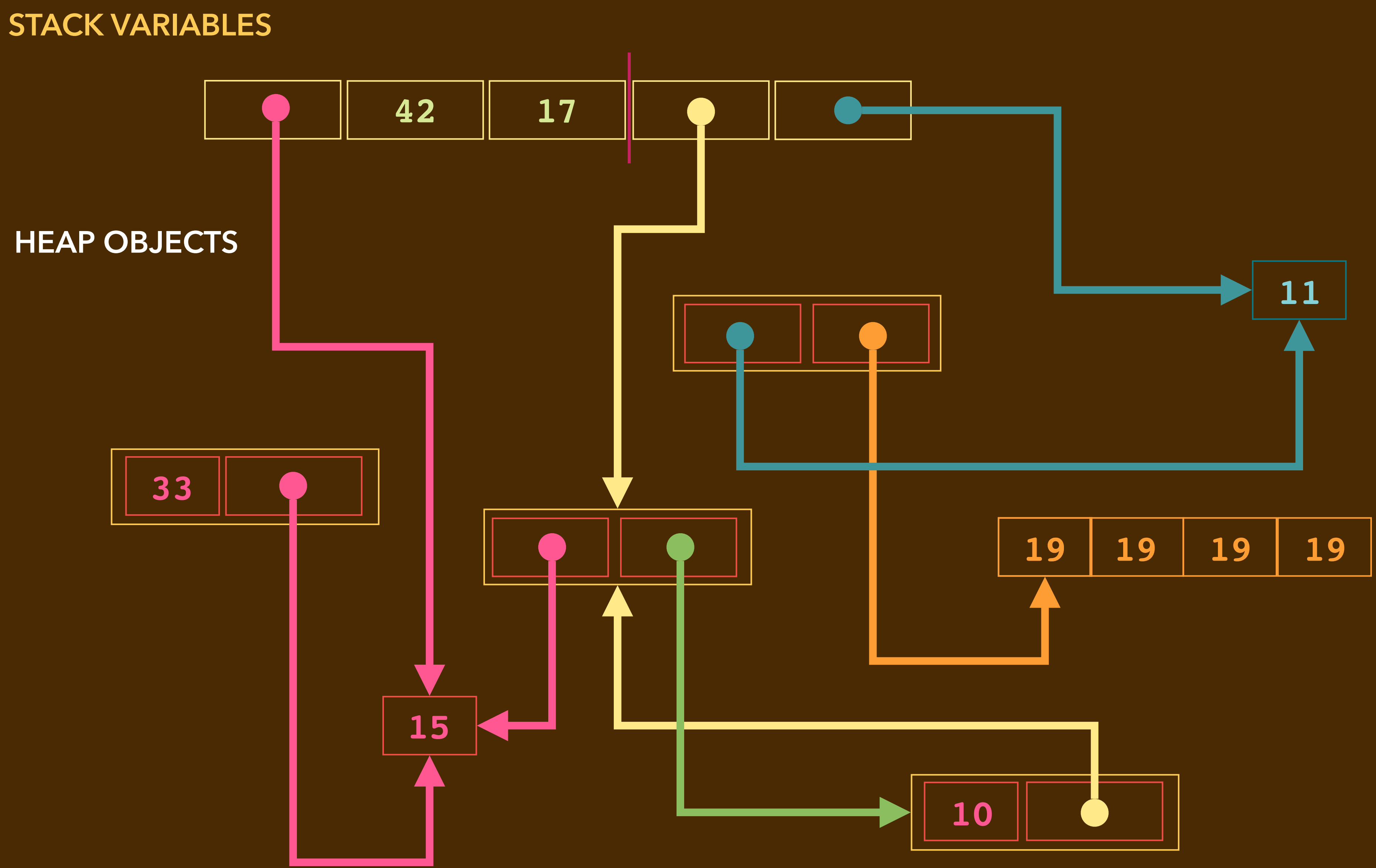
RESOURCE MANAGEMENT VIEWPOINT

- 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

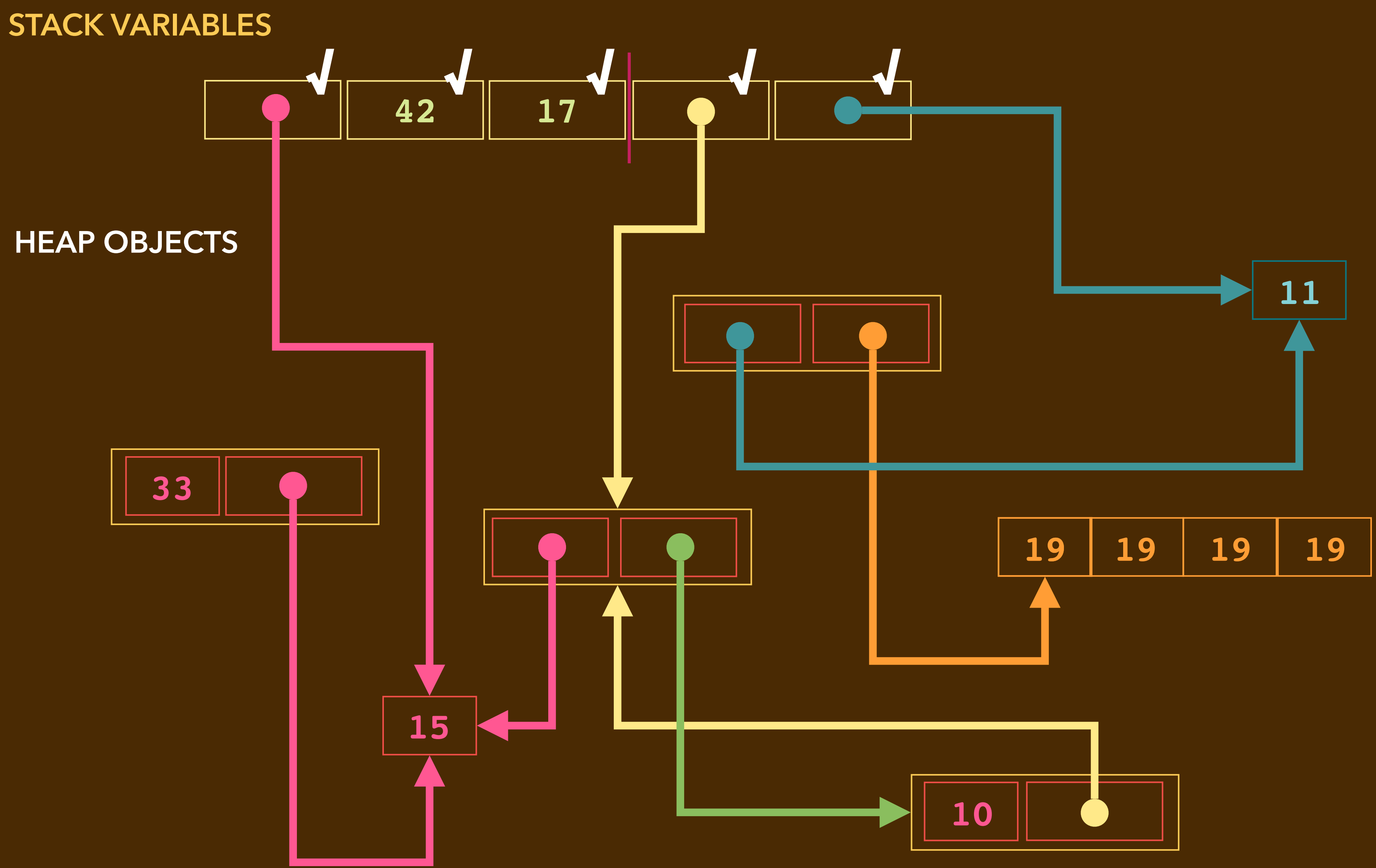
AUTOMATIC GARBAGE COLLECTION

- 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.

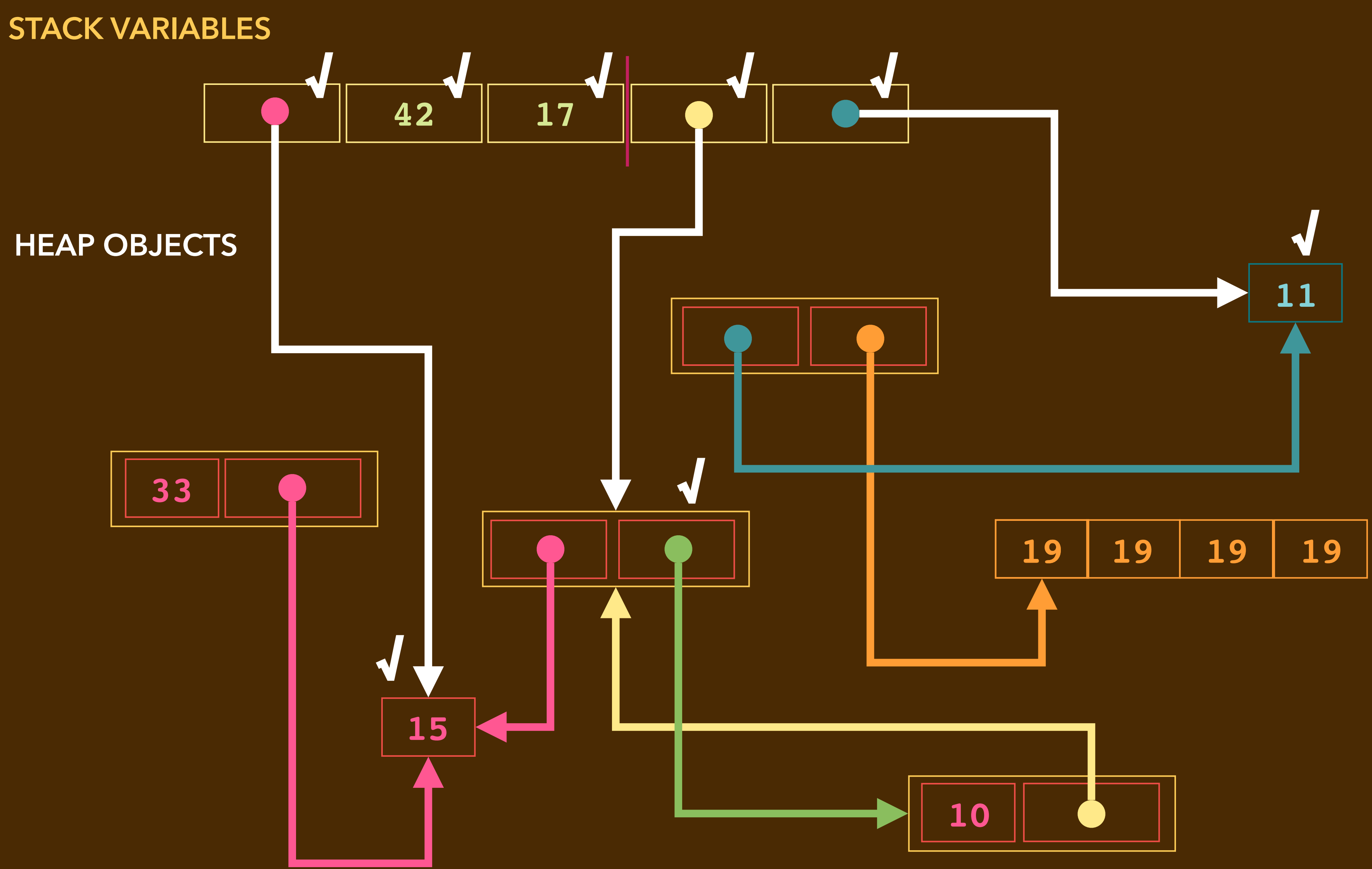
MEMORY DIAGRAM: MARK AND SWEEP



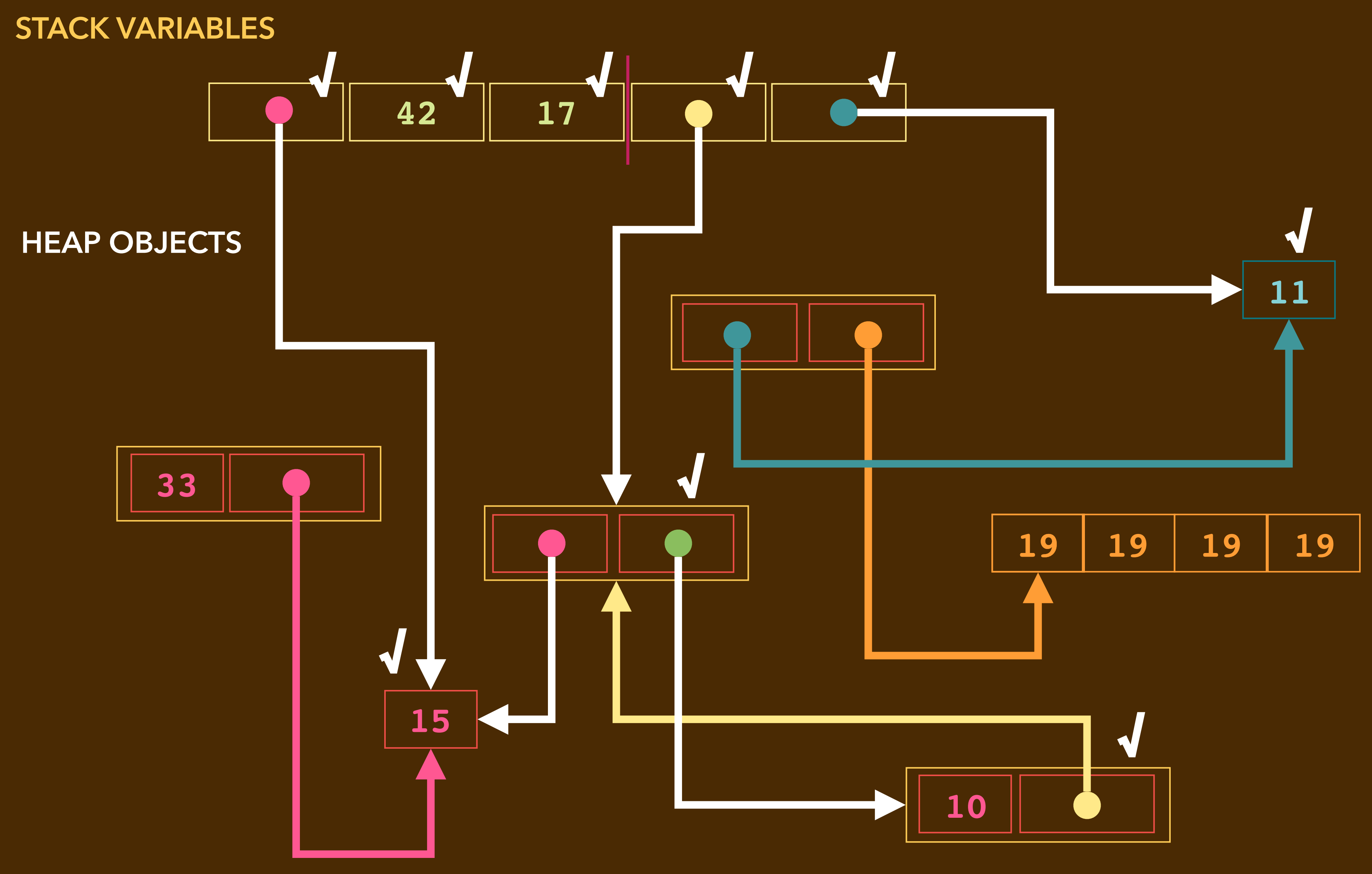
MEMORY DIAGRAM: MARK AND SWEEP



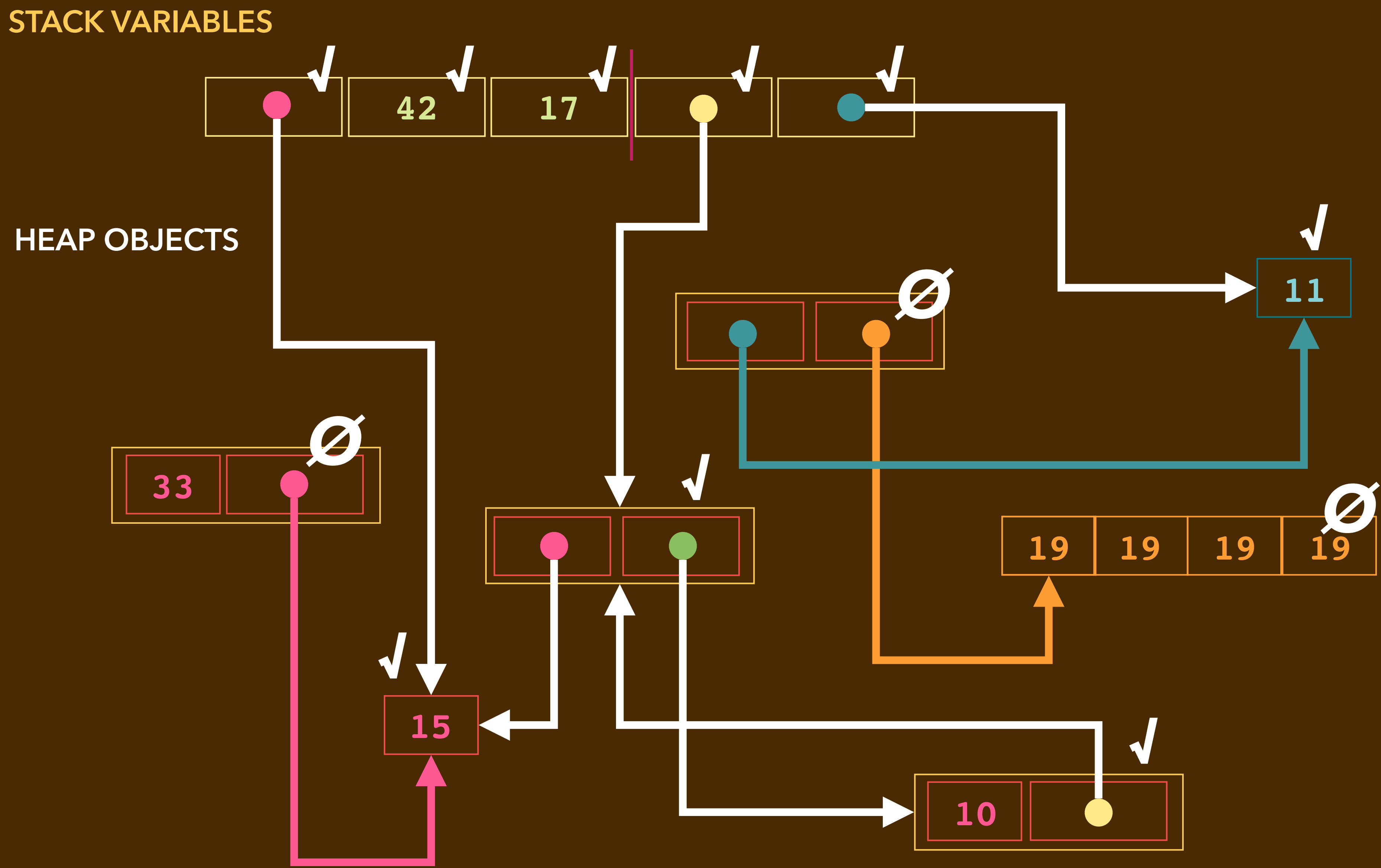
MEMORY DIAGRAM: MARK AND SWEEP



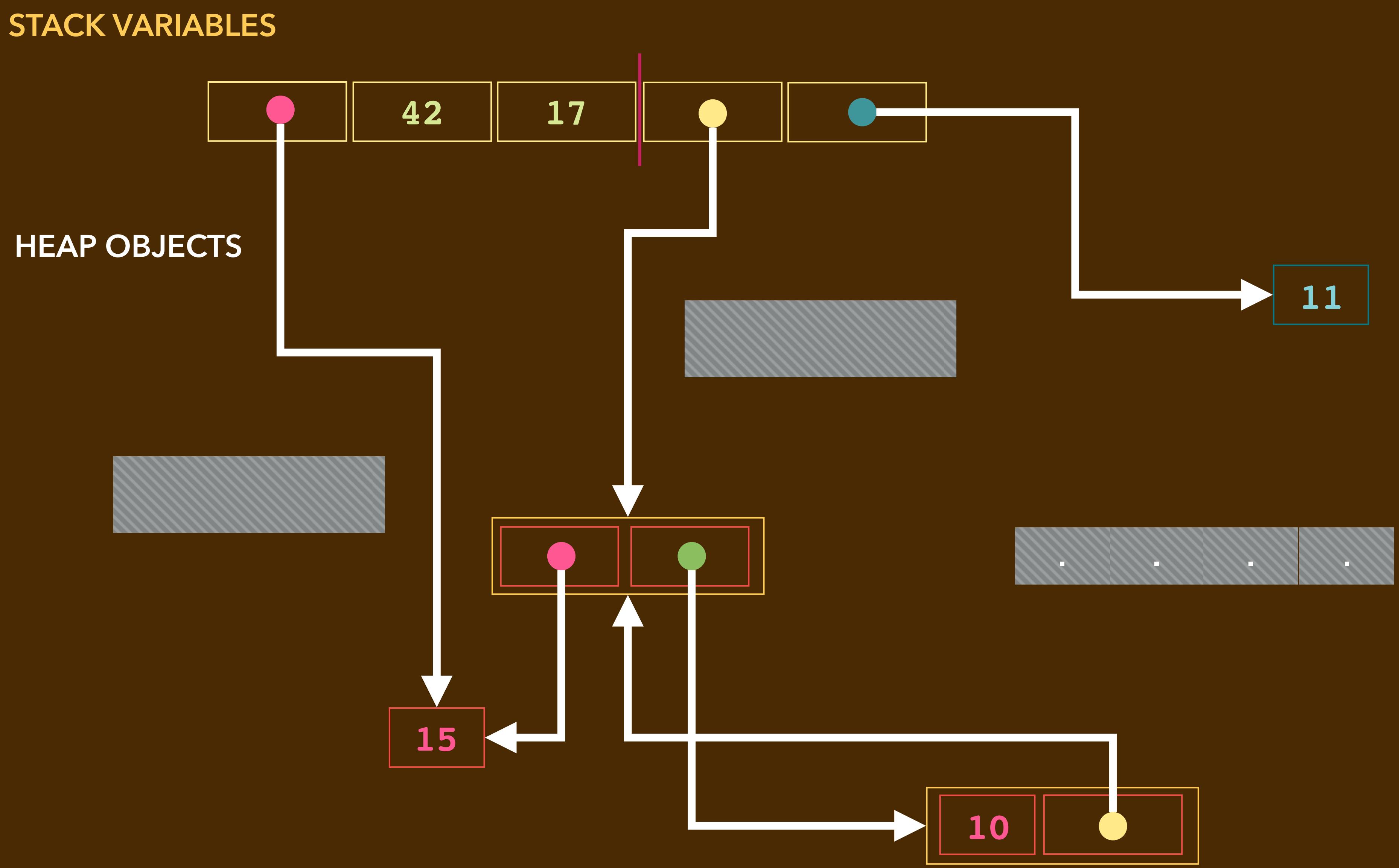
MEMORY DIAGRAM: MARK AND SWEEP



MEMORY DIAGRAM: MARK AND SWEEP



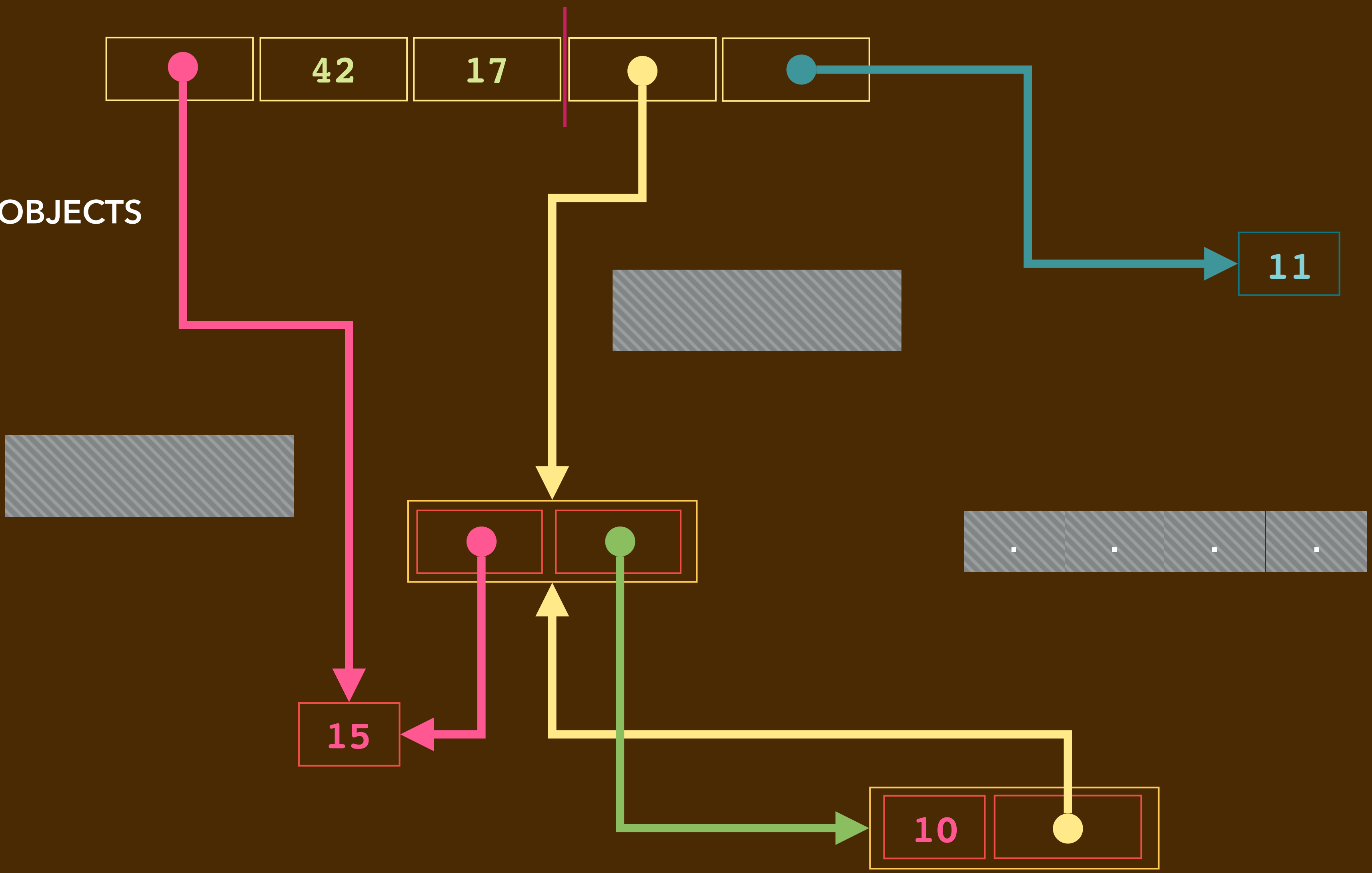
MEMORY DIAGRAM: MARK AND SWEEP



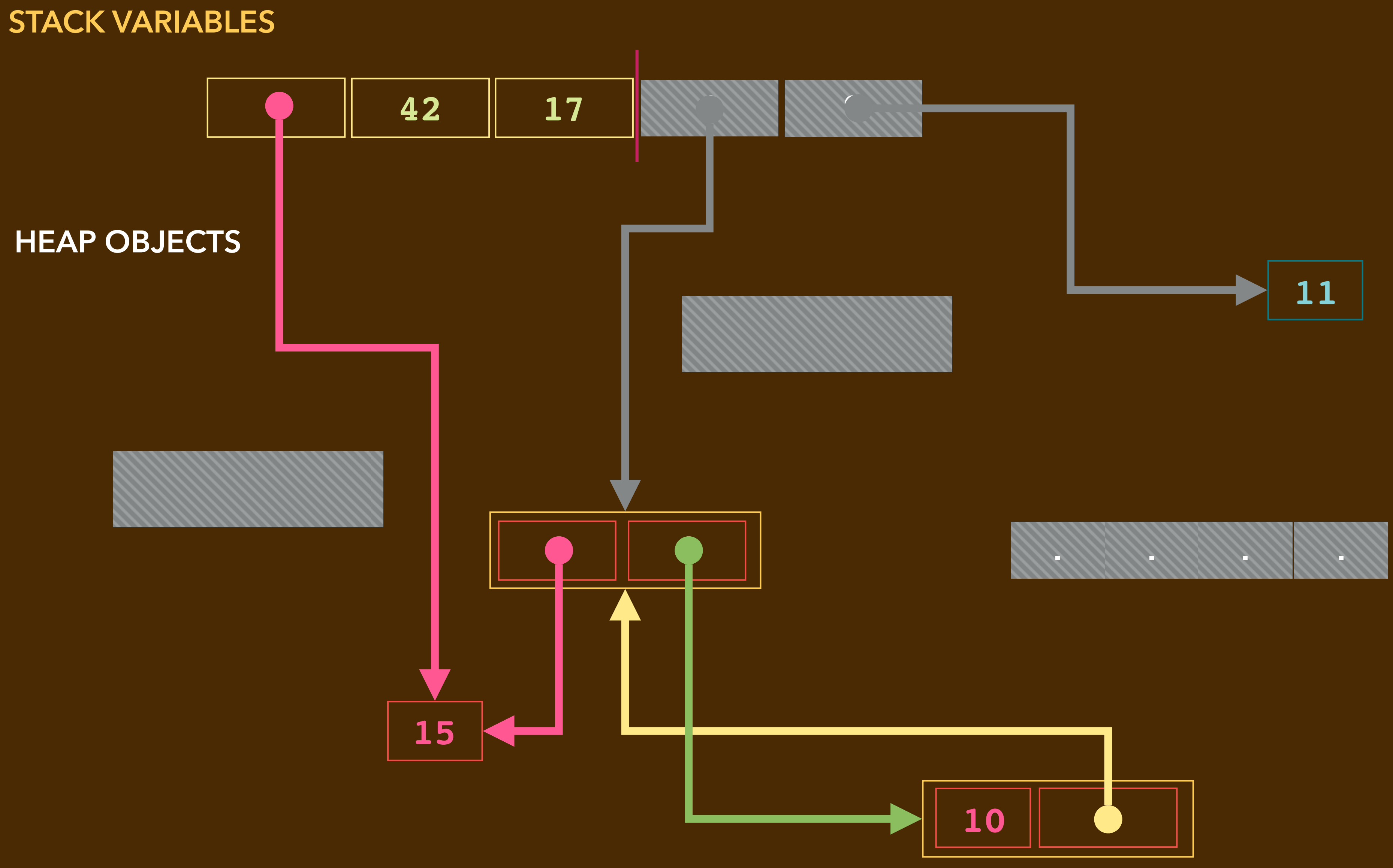
MEMORY DIAGRAM: MARK AND SWEEP

STACK VARIABLES

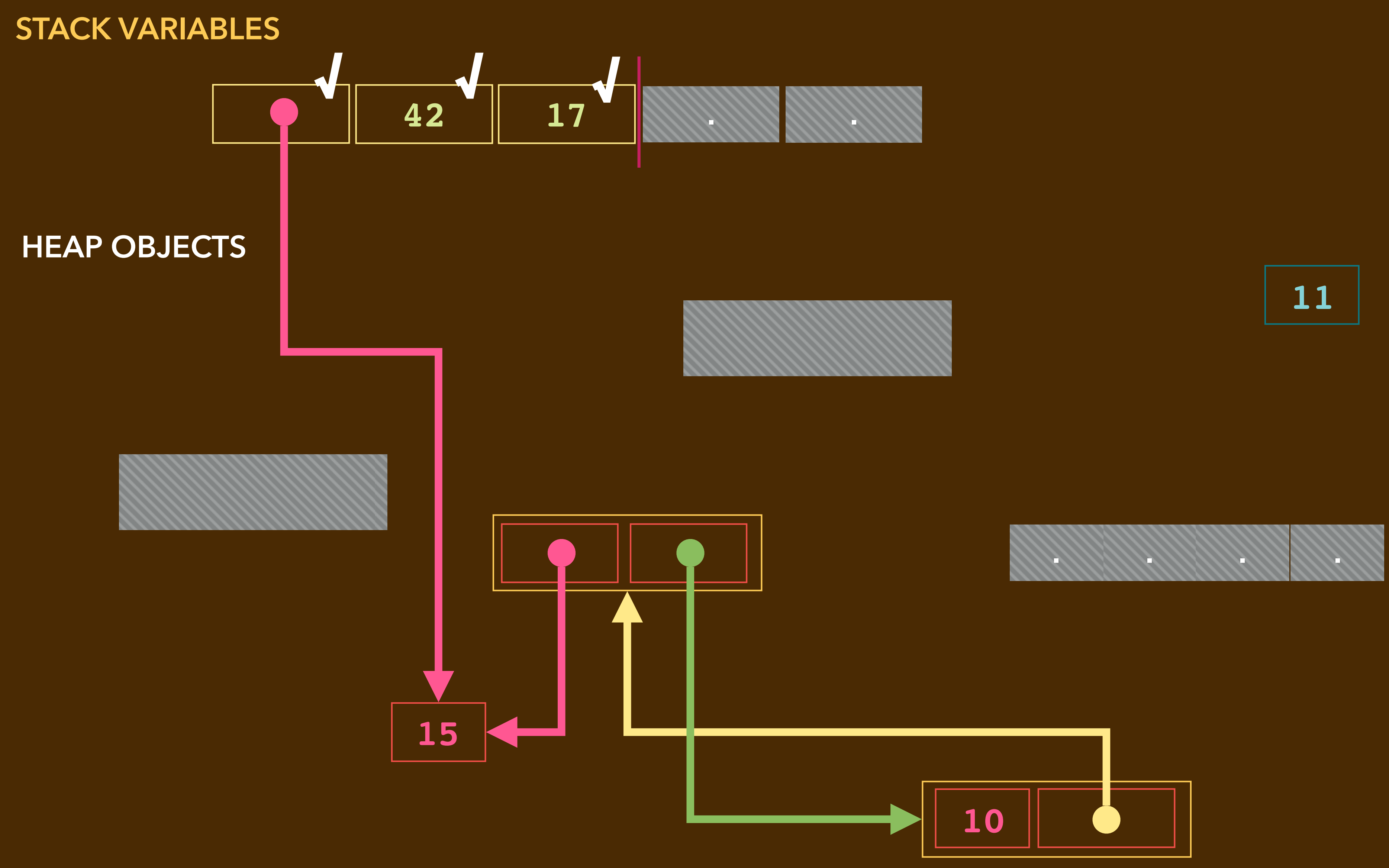
HEAP OBJECTS



MEMORY DIAGRAM: MARK AND SWEEP



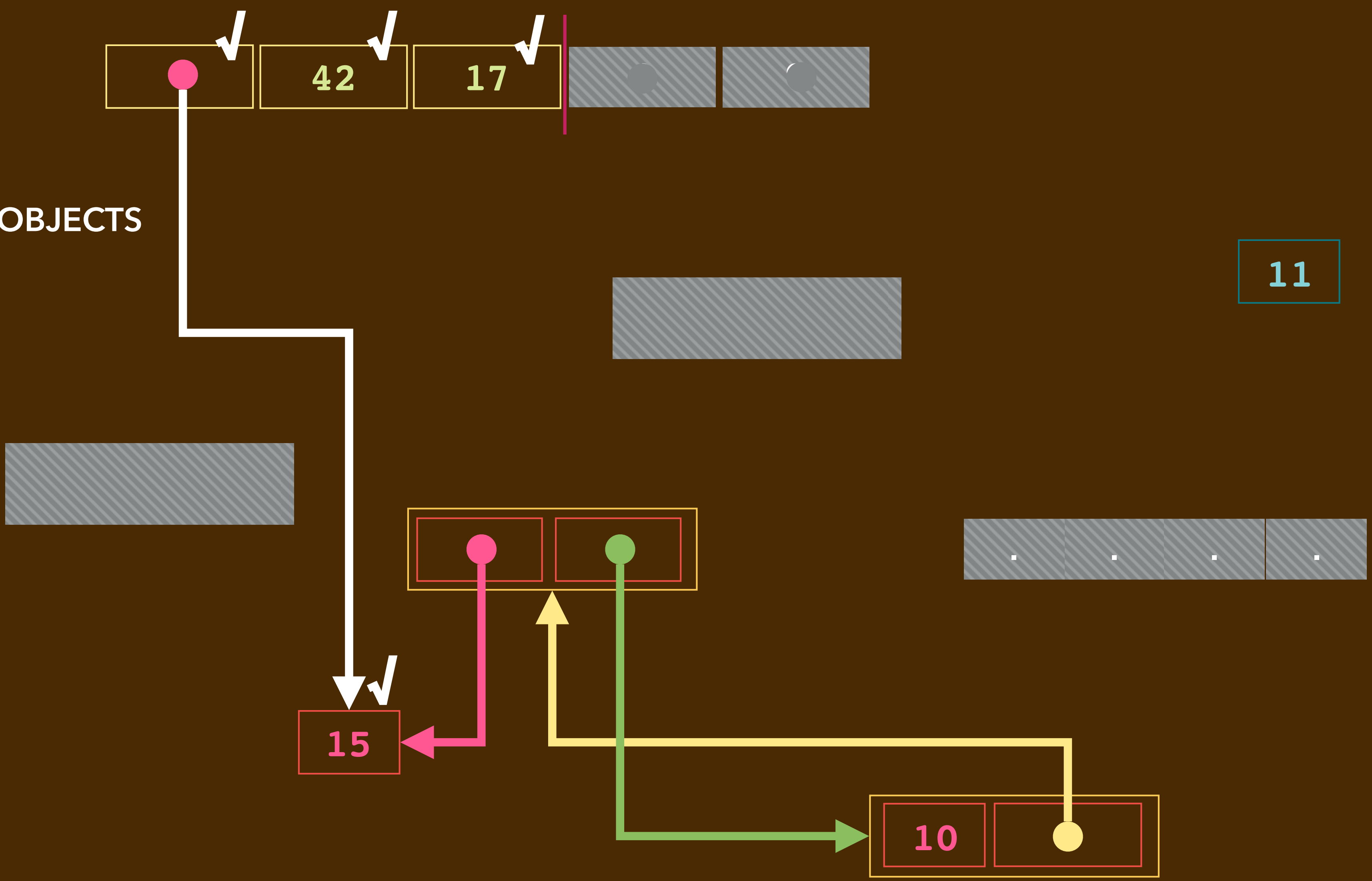
MEMORY DIAGRAM: MARK AND SWEEP



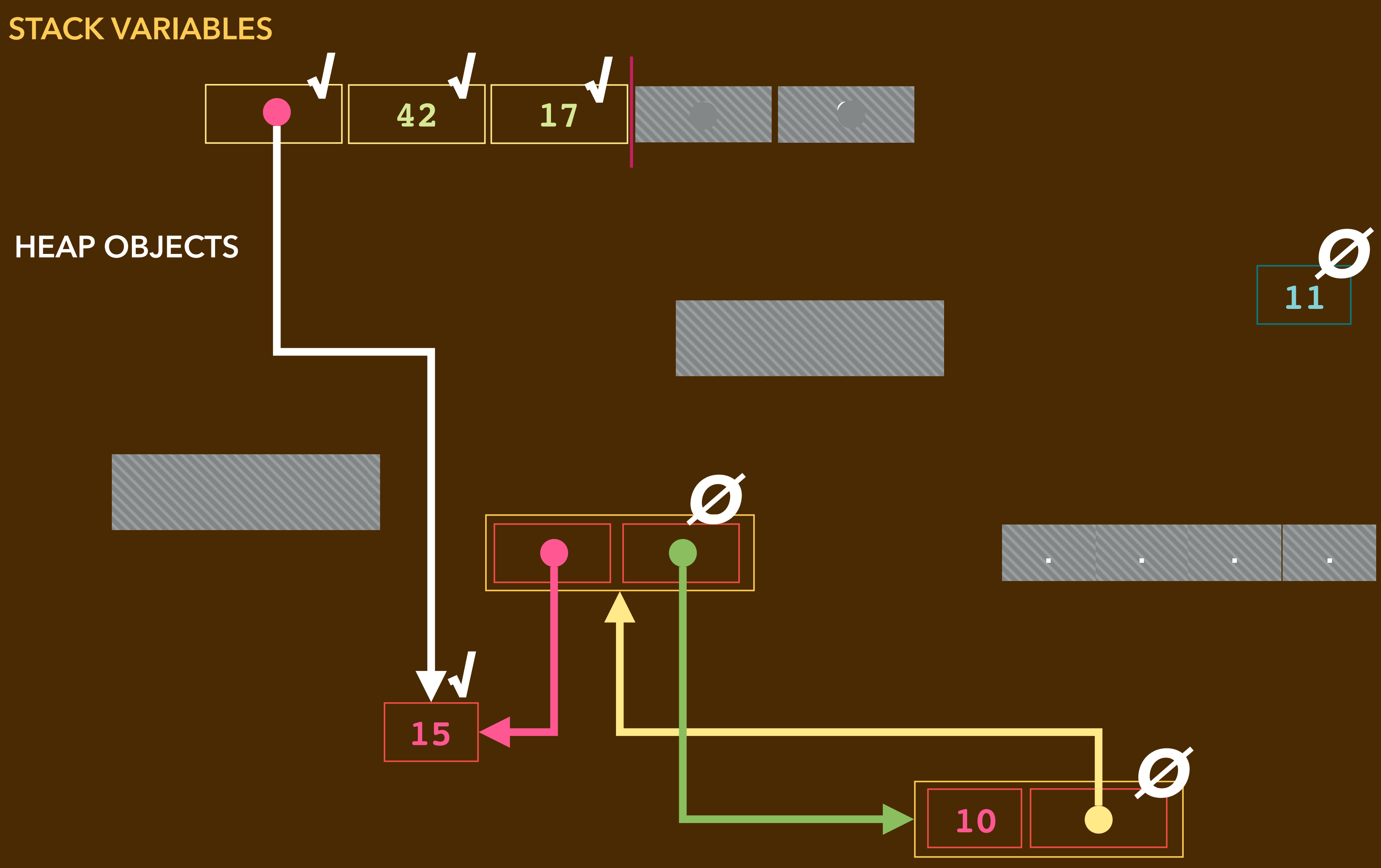
MEMORY DIAGRAM: MARK AND SWEEP

STACK VARIABLES

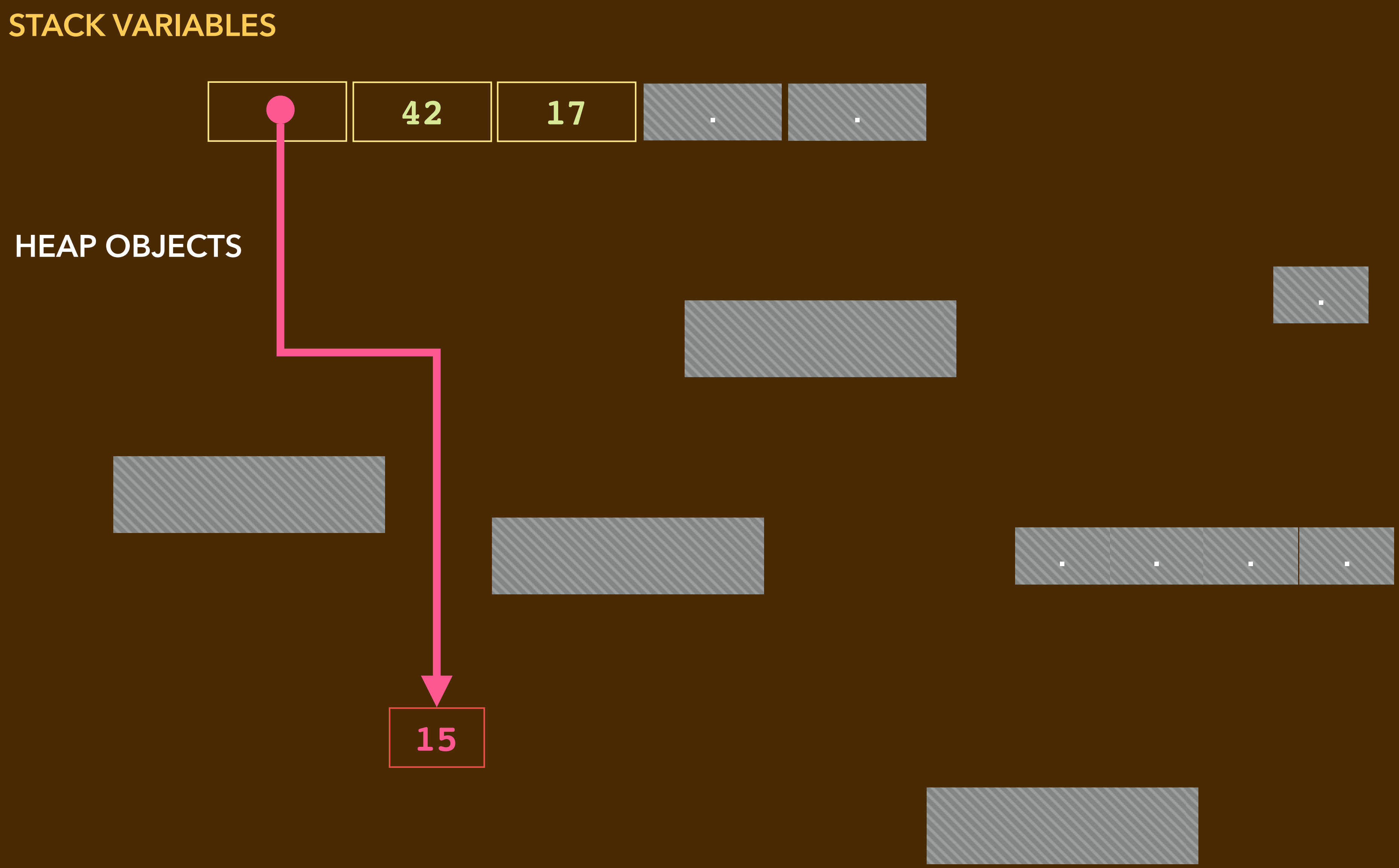
HEAP OBJECTS



MEMORY DIAGRAM: MARK AND SWEEP



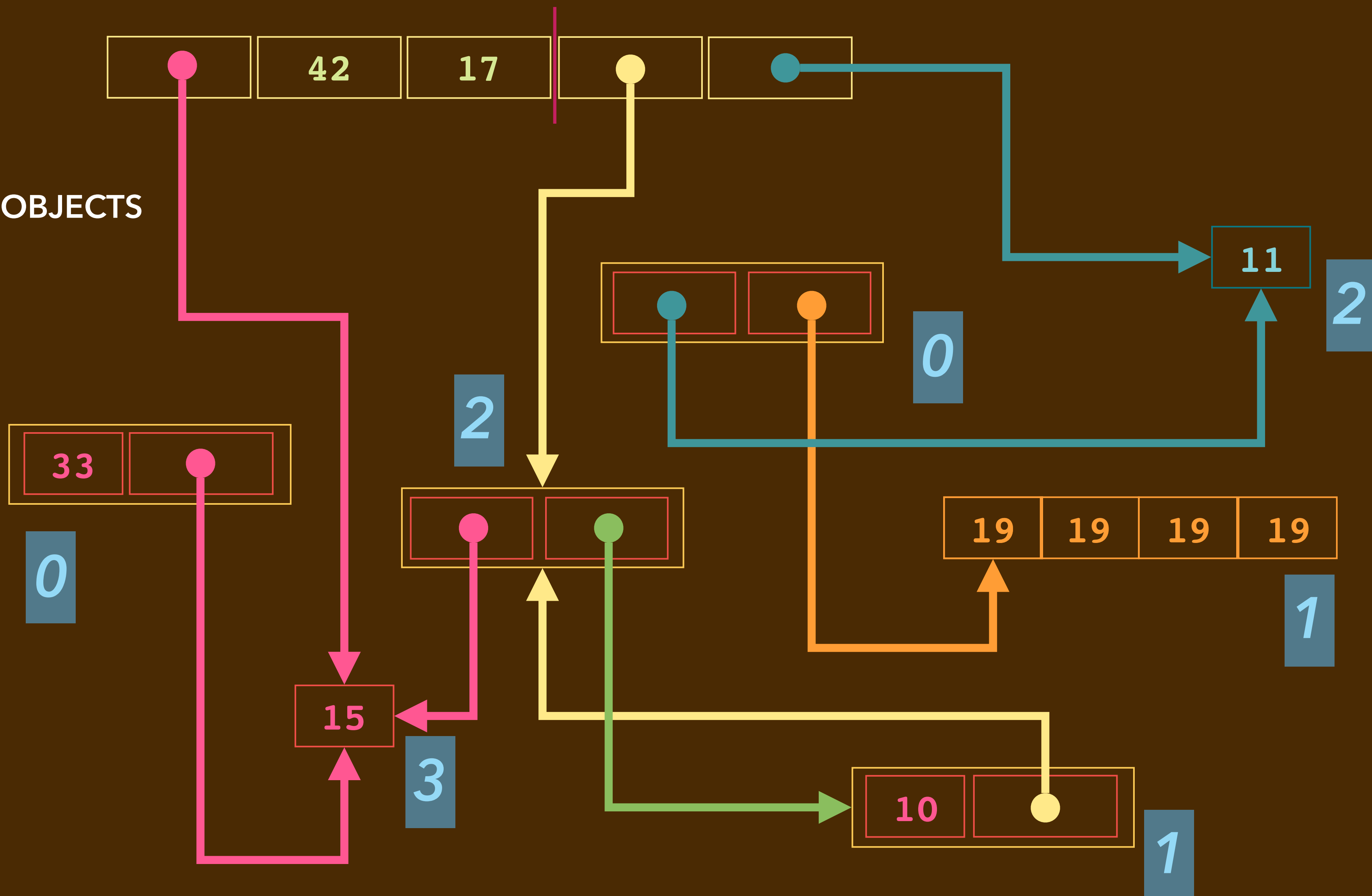
MEMORY DIAGRAM: MARK AND SWEEP



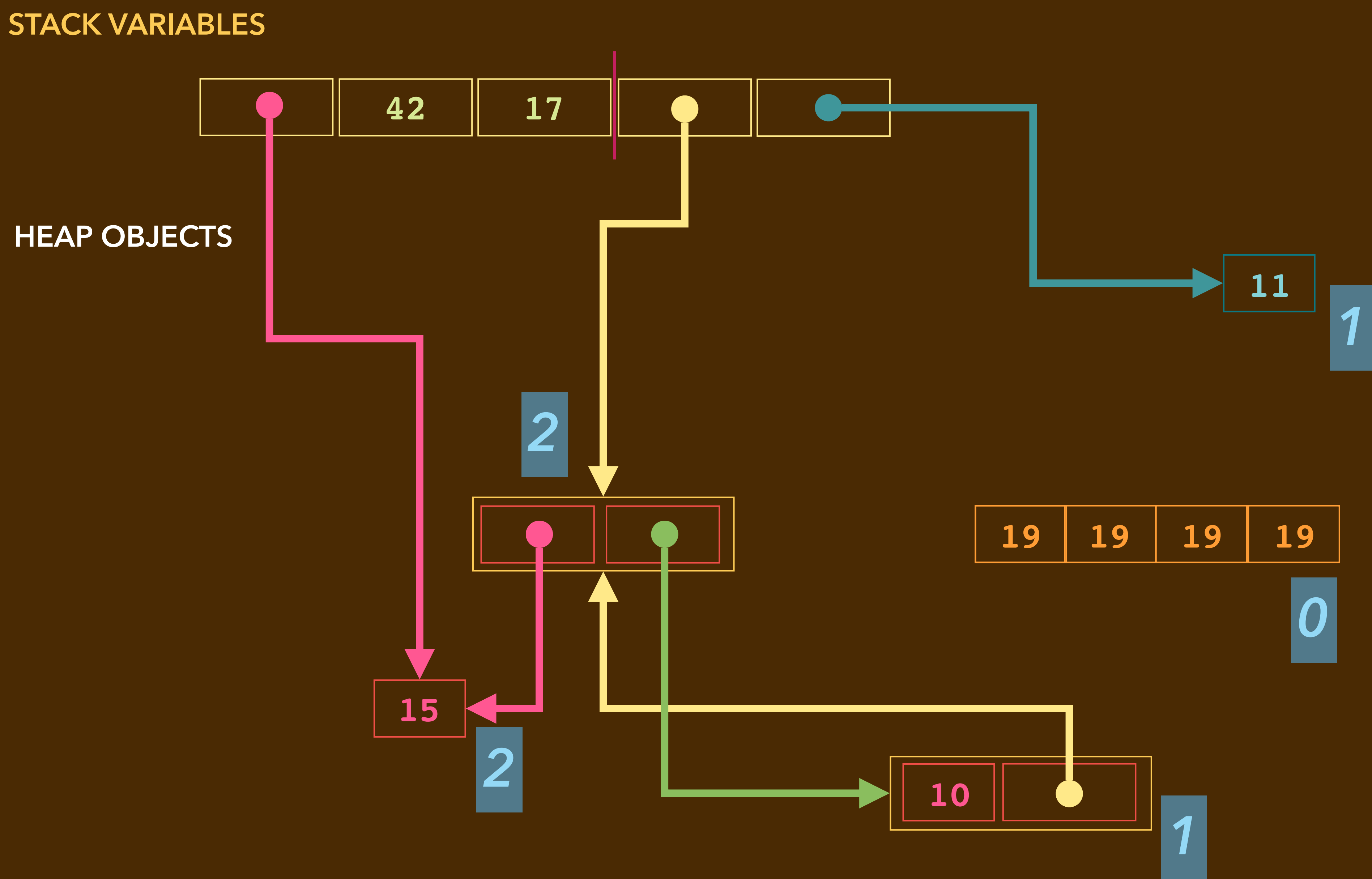
MEMORY DIAGRAM: REFERENCE COUNTS

STACK VARIABLES

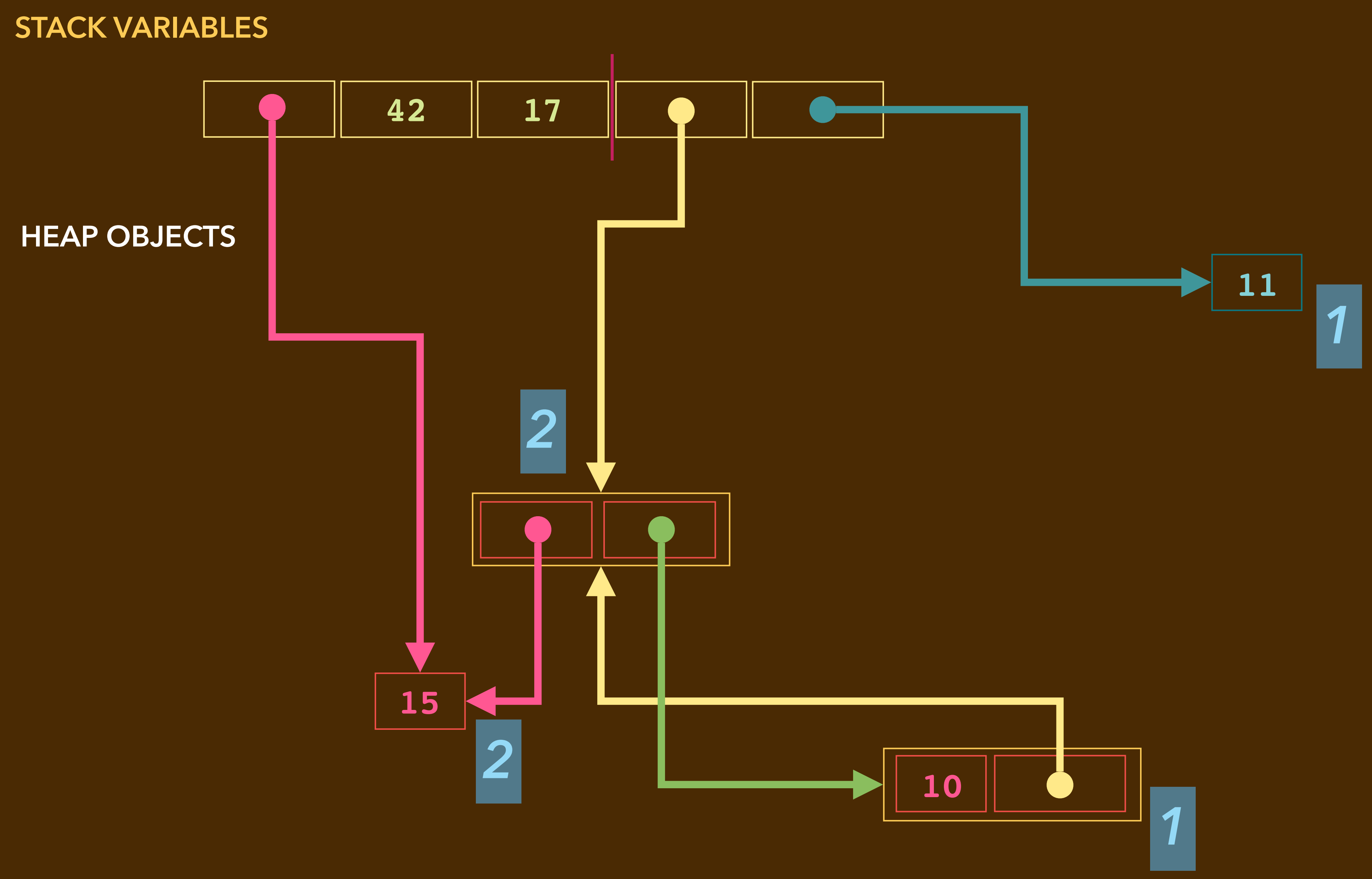
HEAP OBJECTS



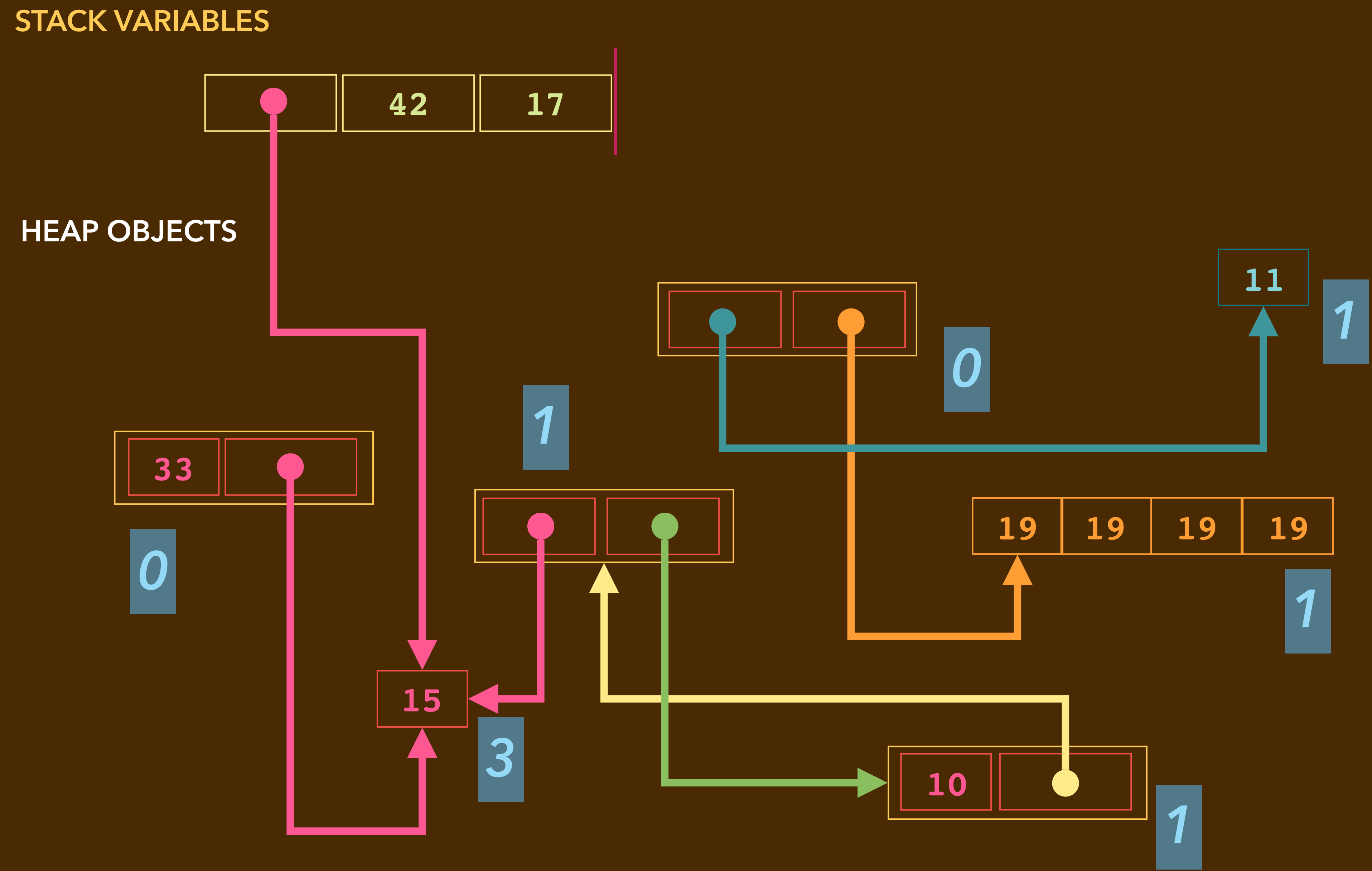
MEMORY DIAGRAM: REFERENCE COUNTS



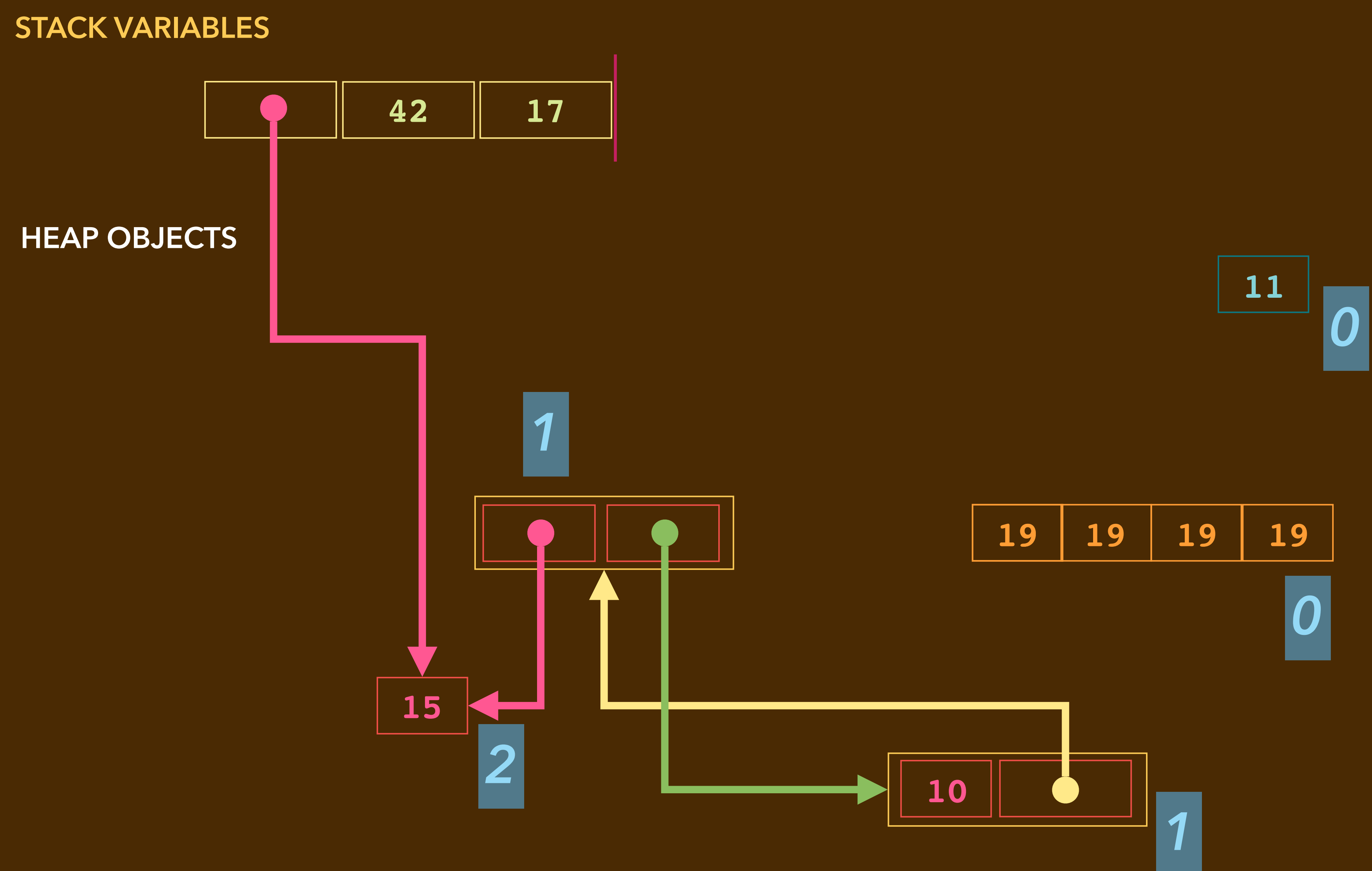
MEMORY DIAGRAM: REFERENCE COUNTS



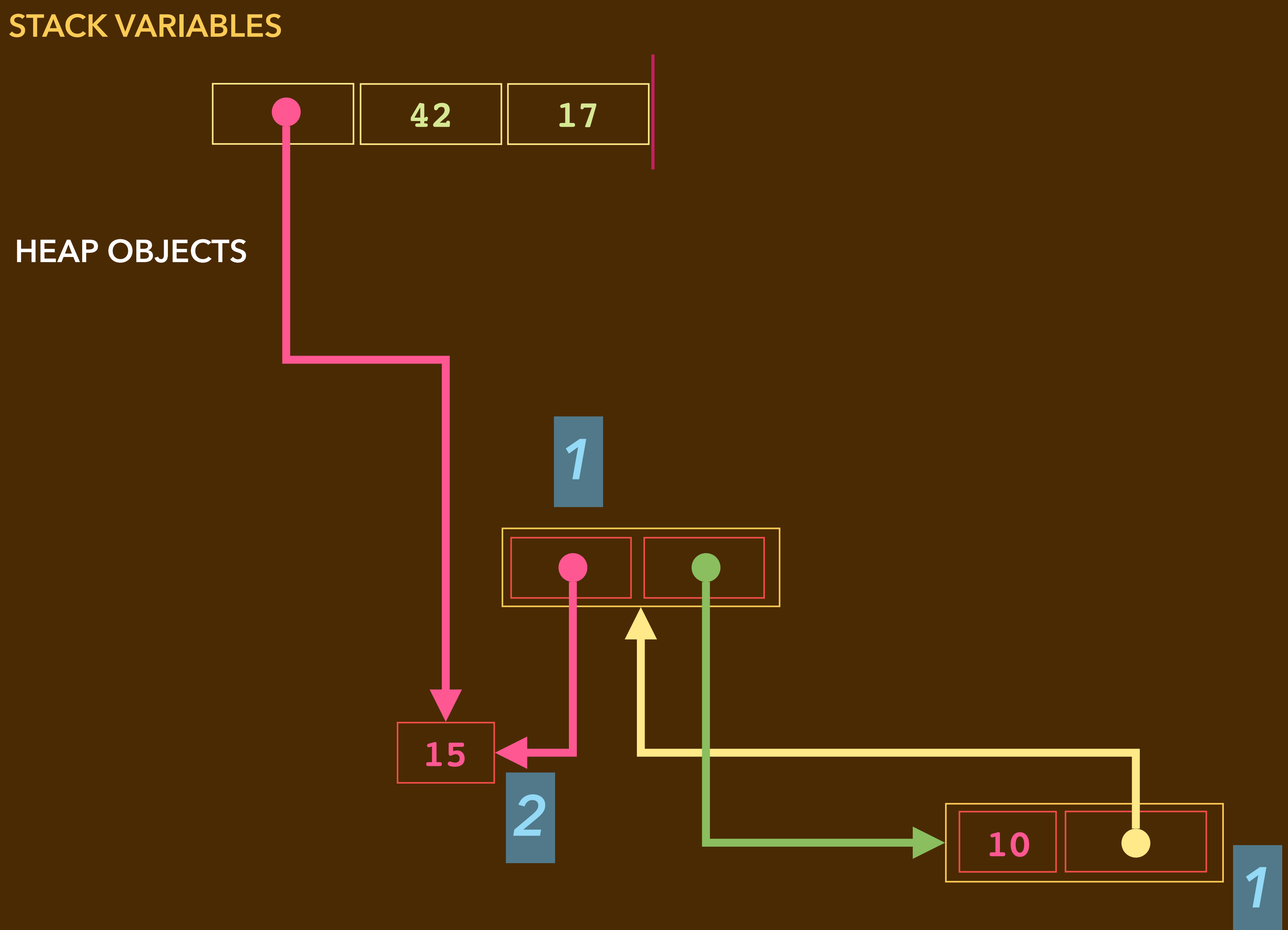
MEMORY DIAGRAM: REFERENCE COUNTS WITH CYCLES



MEMORY DIAGRAM: REFERENCE COUNTS WITH CYCLES



MEMORY DIAGRAM: REFERENCE COUNTS WITH CYCLES



MEMORY MANAGEMENT IN C++

- As we've discussed, C++ forces you to manage memory by hand.
- There is no automatic garbage collection.
- There are rampant bugs related to memory management in C++ code.

MEMORY MANAGEMENT IN C++

- As we've discussed, C++ forces you to manage memory by hand.
- There is no automatic garbage collection.
- There are rampant bugs related to memory management in C++ code.
- C++11 introduced ***smart pointers*** with its STL.
  - These mimic some of the ideas we just surveyed.

SMART POINTERS IN THE C++ STL

- The C++ STL provides three template types (`#include <memory>`)
  - `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*.
  - `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.

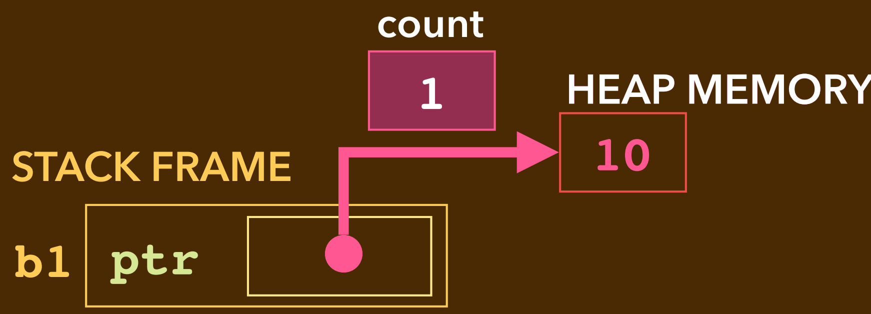
SMART POINTERS IN THE C++ STL

- The C++ STL provides three template types (**#include <memory>**)
  - **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*.
  - **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.

- We'll look at use of **shared_ptr** in a linked list implementation.

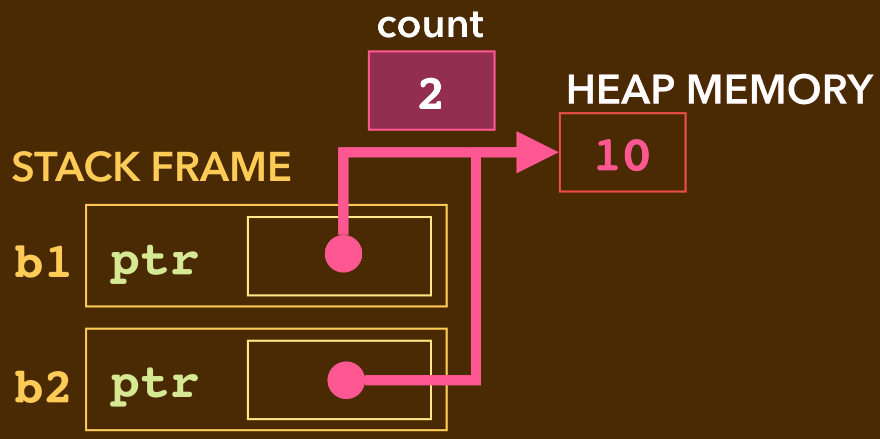
A SHARED_PTR BOX

```
#include <memory>
class Box {
public:
  std::shared_ptr<int> ptr;
  Box(int value) : {new int[value]} { }
  Box(const Box& b) : {b.ptr} { }
  Box& operator=(const Box& b) { ptr = b.ptr; return *this; }
 ~Box(void) { delete ptr; }
};

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



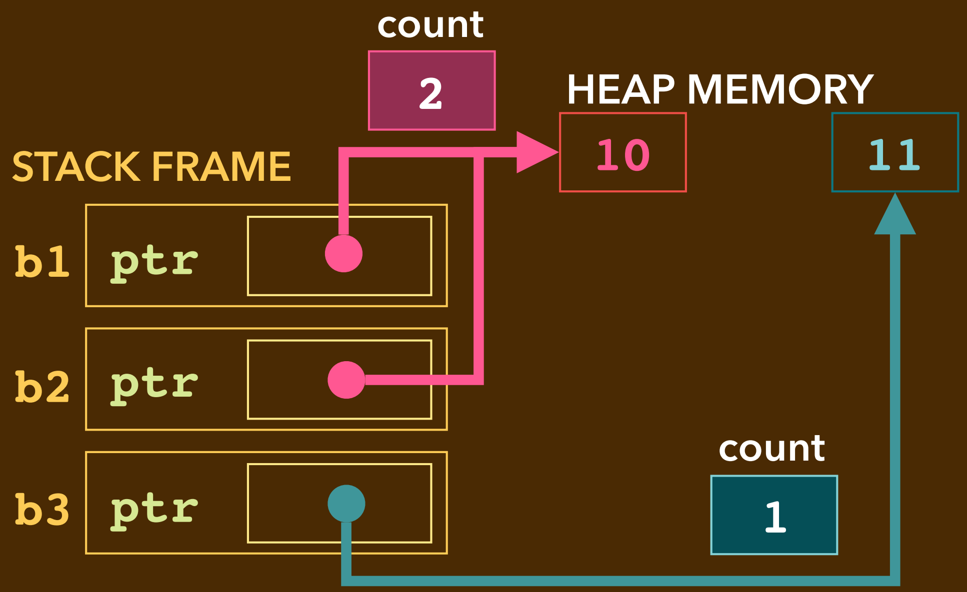
A SHARED_PTR BOX

```
#include <memory>
class Box {
public:
  std::shared_ptr<int> ptr;
  Box(int value) : {new int[value]} { }
  Box(const Box& b) : {b.ptr} { }
  Box& operator=(const Box& b) { ptr = b.ptr; return *this; }
 ~Box(void) { delete 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) : {new int[value]} { }
   Box(const Box& b) : {b.ptr} { }
   Box& operator=(const Box& b) { ptr = b.ptr; return *this; }
  ~Box(void) { delete ptr; }
};

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

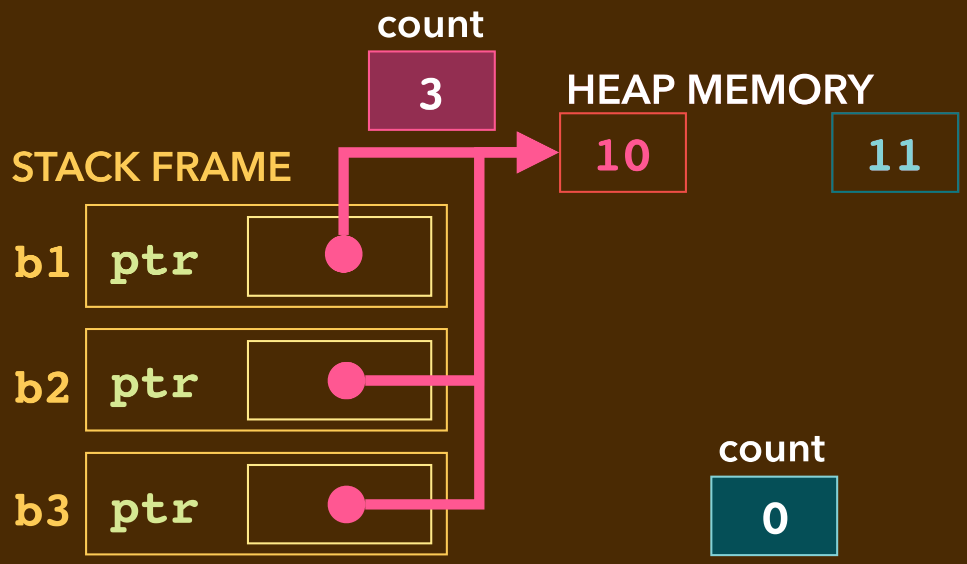


A SHARED_PTR BOX

```
#include <memory>
class Box {
public:
  std::shared_ptr<int> ptr;
  Box(int value) : {new int[value]} { }
  Box(const Box& b) : {b.ptr} { }
  Box& operator=(const Box& b) { ptr = b.ptr; return *this; }
 ~Box(void) { delete ptr; }
};

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

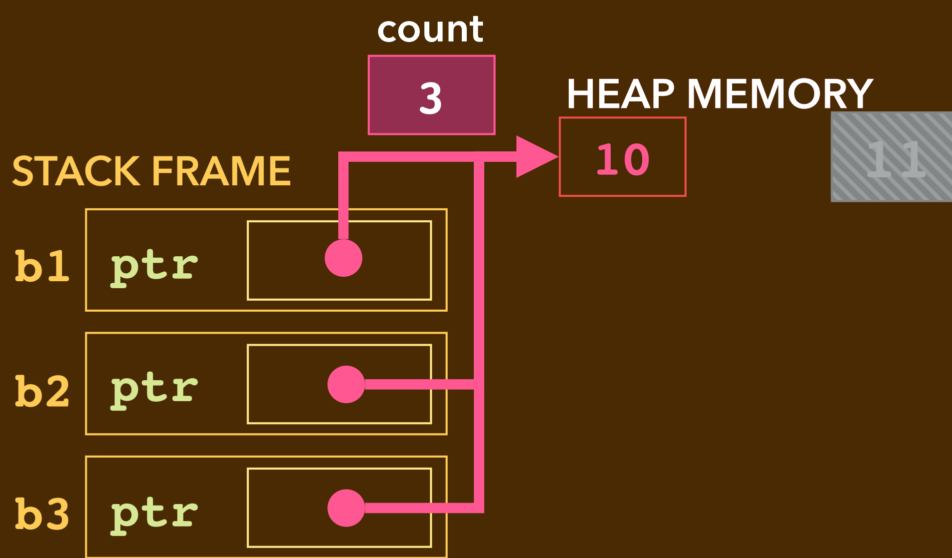
b1.ptr count increments to 3

old b3.ptr decrements to 0



A SHARED_PTR BOX

```
#include <memory>
class Box {
public:
  std::shared_ptr<int> ptr;
  Box(int value) : {new int[value]} { }
  Box(const Box& b) : {b.ptr} { }
  Box& operator=(const Box& b) { ptr = b.ptr; return *this; }
 ~Box(void) { delete ptr; }
};

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

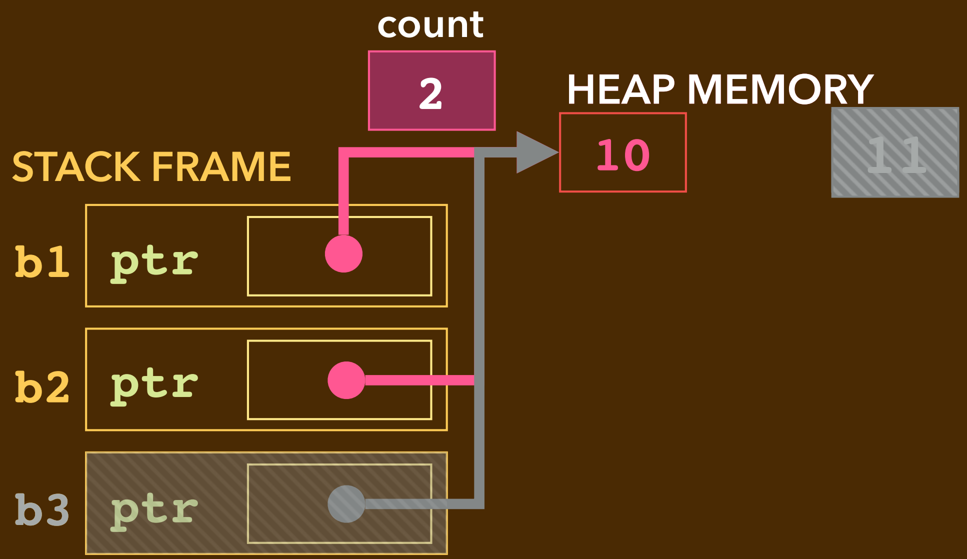


old b3.ptr 's raw pointer is deleted

A SHARED_PTR BOX

```
#include <memory>
class Box {
public:
  std::shared_ptr<int> ptr;
  Box(int value) : {new int[value]} { }
  Box(const Box& b) : {b.ptr} { }
  Box& operator=(const Box& b) { ptr = b.ptr; return *this; }
 ~Box(void) { delete ptr; }
};

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

Destructor called on b3; decrement.

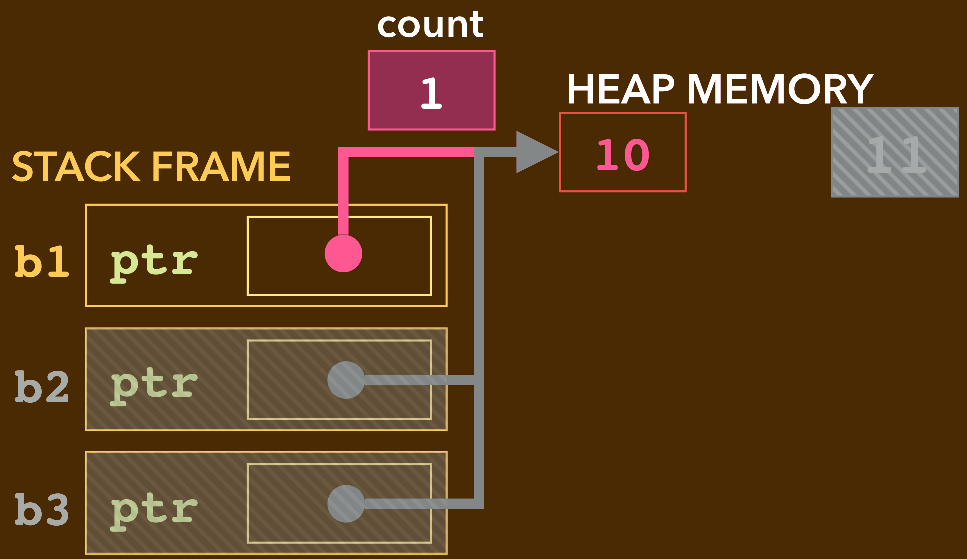


A SHARED_PTR BOX

```
#include <memory>
class Box {
public:
  std::shared_ptr<int> ptr;
  Box(int value) : {new int[value]} { }
  Box(const Box& b) : {b.ptr} { }
  Box& operator=(const Box& b) { ptr = b.ptr; return *this; }
 ~Box(void) { delete ptr; }
};

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

Destructor called on b3; decrement.

Destructor called on b2; decrement.



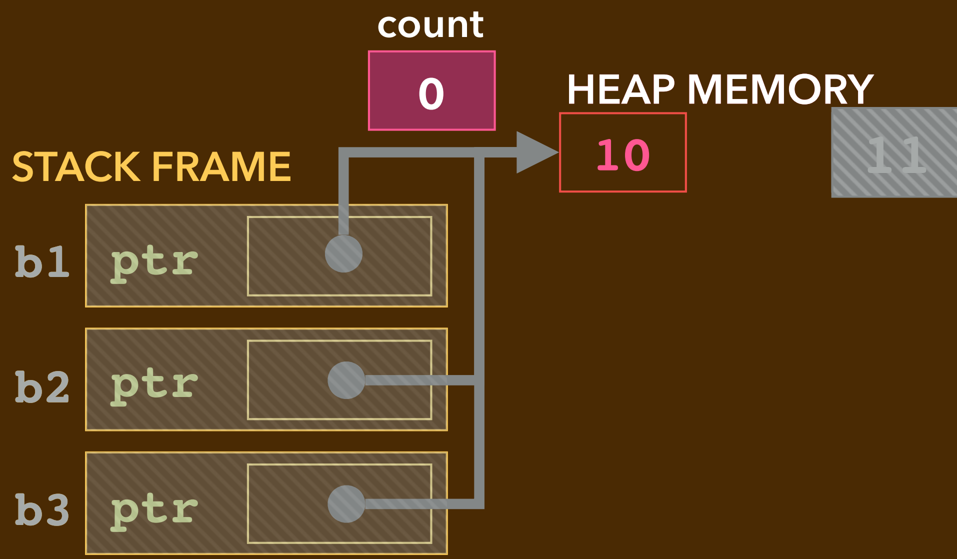
A SHARED_PTR BOX

```
#include <memory>
class Box {
public:
  std::shared_ptr<int> ptr;
  Box(int value) : {new int[value]} { }
  Box(const Box& b) : {b.ptr} { }
  Box& operator=(const Box& b) { ptr = b.ptr; return *this; }
 ~Box(void) { delete ptr; }
};

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

Destructor called on b3; decrement.

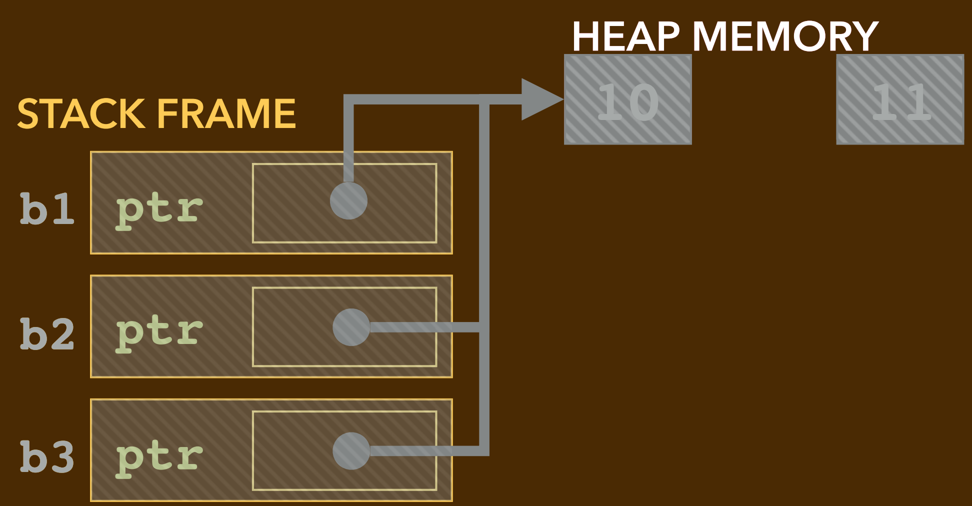
Destructor called on b2; decrement.

Destructor called on b1; decrement.



A SHARED_PTR BOX

```
#include <memory>
class Box {
public:
   std::shared_ptr<int> ptr;
   Box(int value) : {new int[value]} { }
   Box(const Box& b) : {b.ptr} { }
   Box& operator=(const Box& b) { ptr = b.ptr; return *this; }
  ~Box(void) { delete ptr; }
};

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



Destructor called on b3; decrement.

Destructor called on b2; decrement.

Destructor called on b1; decrement. The raw pointer of b1.ptr is deleted.

LINKED LISTS USING SMART POINTERS

- We end this lecture with a re-implementation of linked lists using C++'s **`shared_ptr`**.
- The end result is an implementation where **`delete`** is never explicitly called,

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 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);
  void append(int value);
  void remove(int value);
};
```

LINKED LIST SHARED_PTR USE

```
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;
}
```

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` goes out of scope; count goes to 0; node is reclaimed

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 explicitly calls* **`delete`**.

CHECK OUT MY SAMPLE CODE

- I have four versions of linked lists that use `shared_ptr` in samples:
  - **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
- In the last, `weak_ptr` back references aren't counted... one typical use.

WE'RE DONE!

- Next week I'll talk about code that communicates over a network.
  - We'll look at the Berkeley **`socket`** library.
- Next week I'll talk about code that does several things *at once*.
  - It's written so that it can run on *multiple processor cores*.
  - We'll look at the POSIX **`pthread`** library.
- These is extra material...

WE'RE DONE!

- Next week I'll talk about code that communicates over a network.
  - We'll look at the Berkeley **`socket`** library.
- Next week I'll talk about code that does several things *at once*.
  - It's written so that it can run on *multiple processor cores*.
  - We'll look at the POSIX **`pthread`** library.
- *These is extra material; FYI; "not on the exam."*

WE'RE DONE!

- Next week I'll talk about code that communicates over a network.
  - We'll look at the Berkeley `socket` library.
- Next week I'll talk about code that does several things *at once*.
  - It's written so that it can run on *multiple processor cores*.
  - We'll look at the POSIX `pthread` library.
- These is extra material; FYI; "not on the exam."

- *We'll have a last homework assignment.*
- *We'll have a comprehensive final exam.*