SMART POINTERS CONT'D

LECTURE 12-1

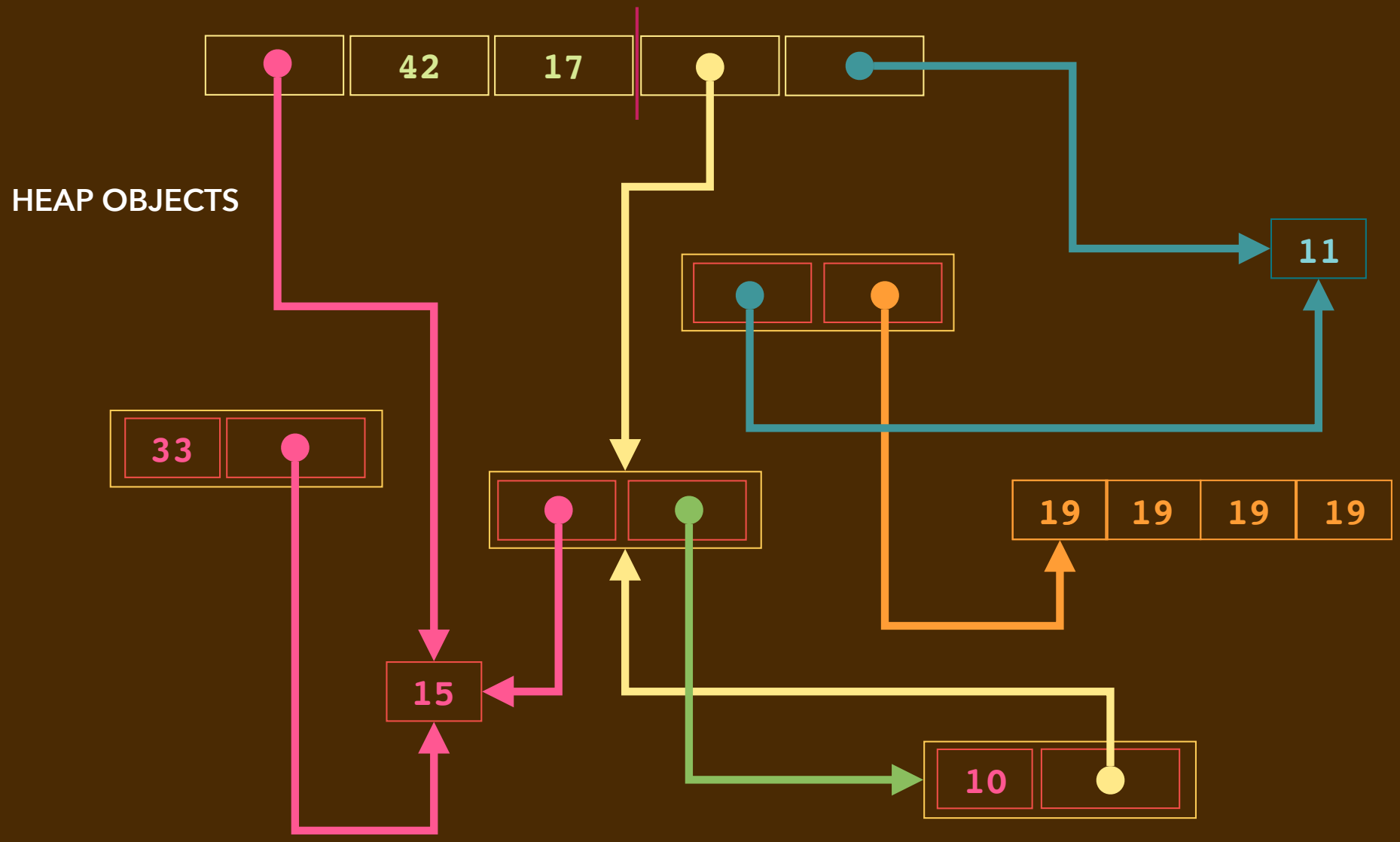
JIM FIX, REED COLLEGE CS2-S20

TODAY'S PLAN

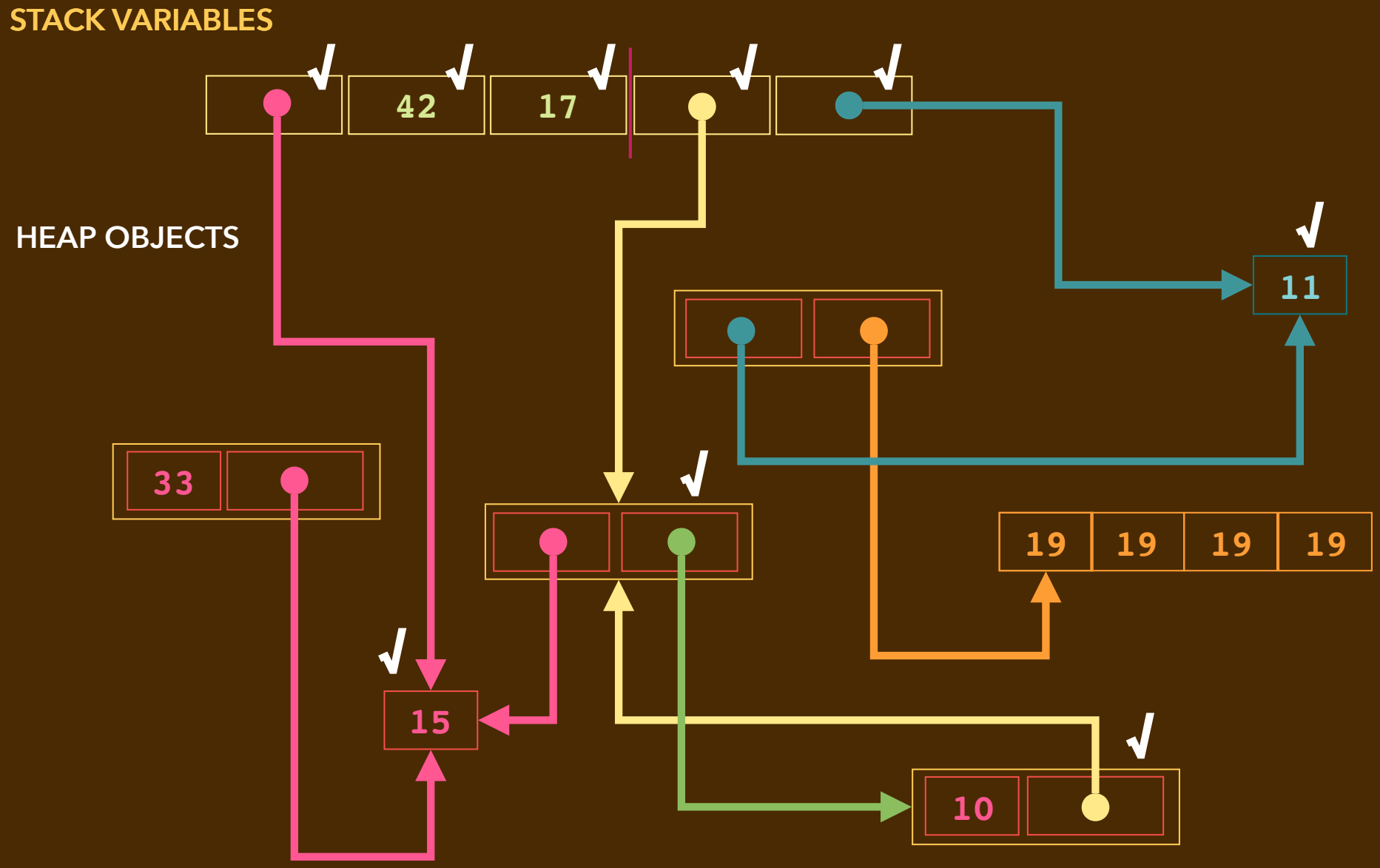
- RECALL: REFERENCE COUNTING
- LOOK AT C++ STL's *SMART POINTERS*
  - BOX EXAMPLE WITH `shared_ptr`
  - SINGLY LINKED LIST EXAMPLE WITH `shared_ptr`
  - DOUBLY LINKED LIST EXAMPLE WITH `shared_ptr` AND `weak_ptr`

- A PREVIEW OF MULTITHREADED PROGRAMMING

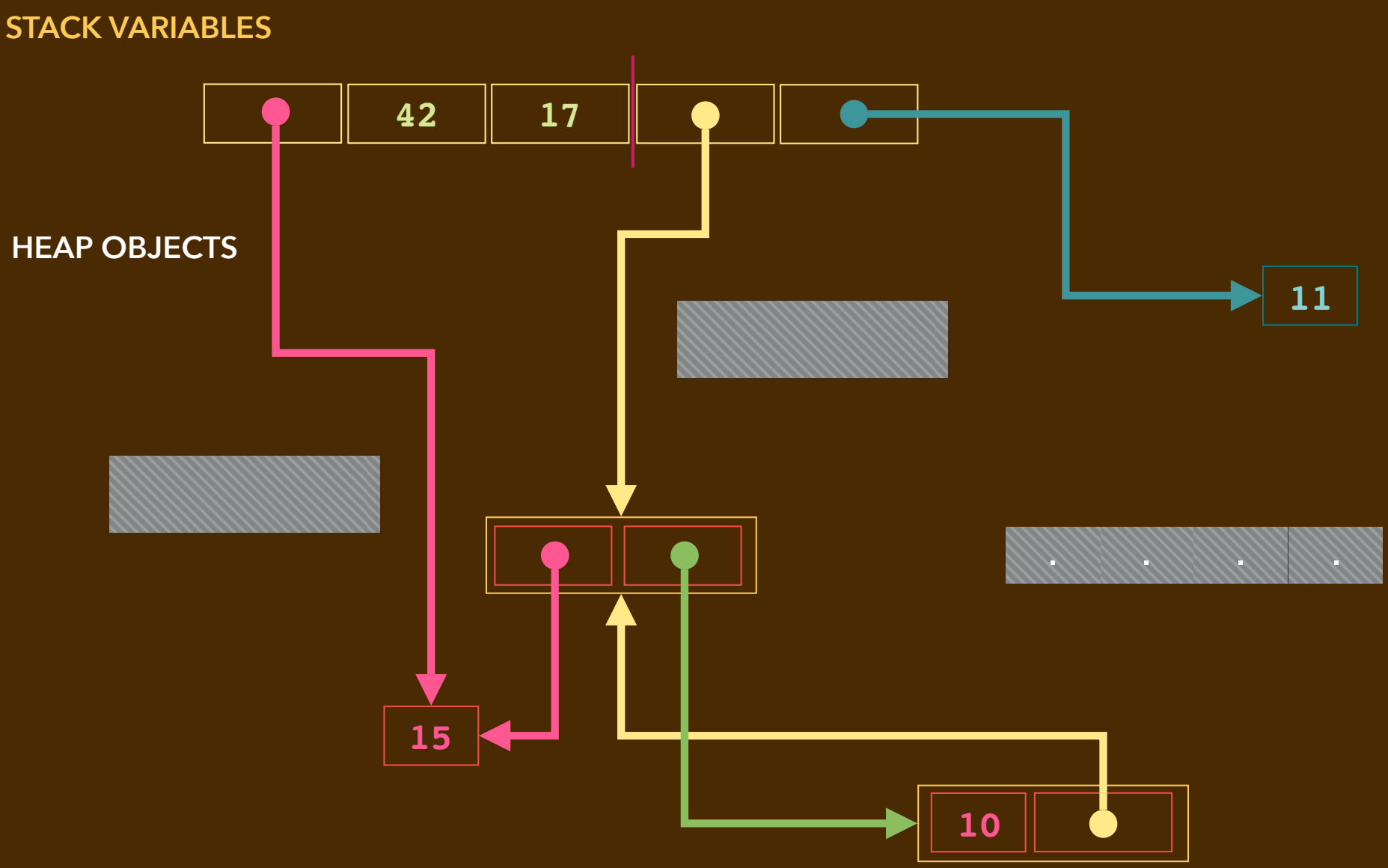
MEMORY DIAGRAM: MARK AND SWEEP



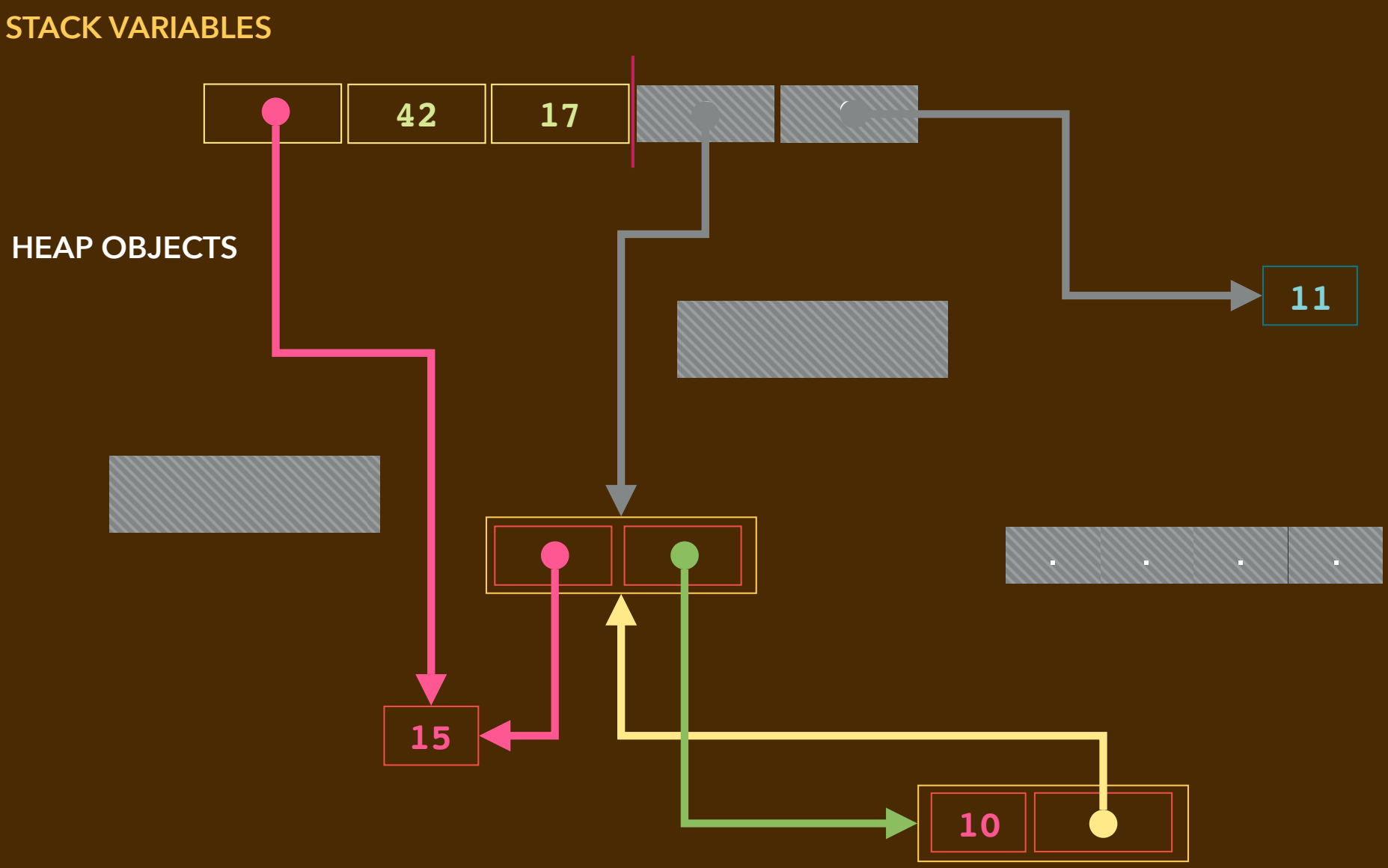
MEMORY DIAGRAM: MARK AND SWEEP



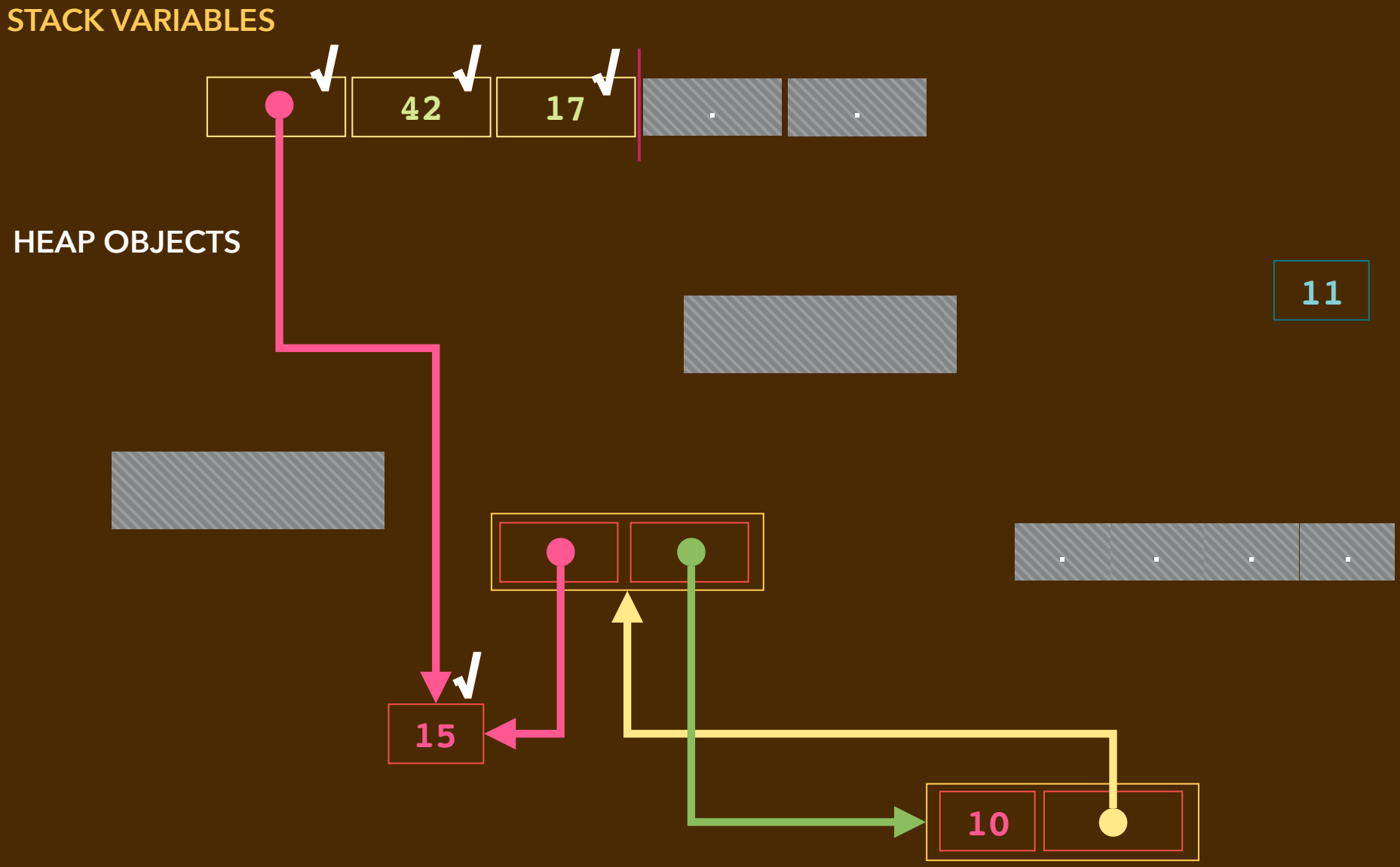
MEMORY DIAGRAM: MARK AND SWEEP



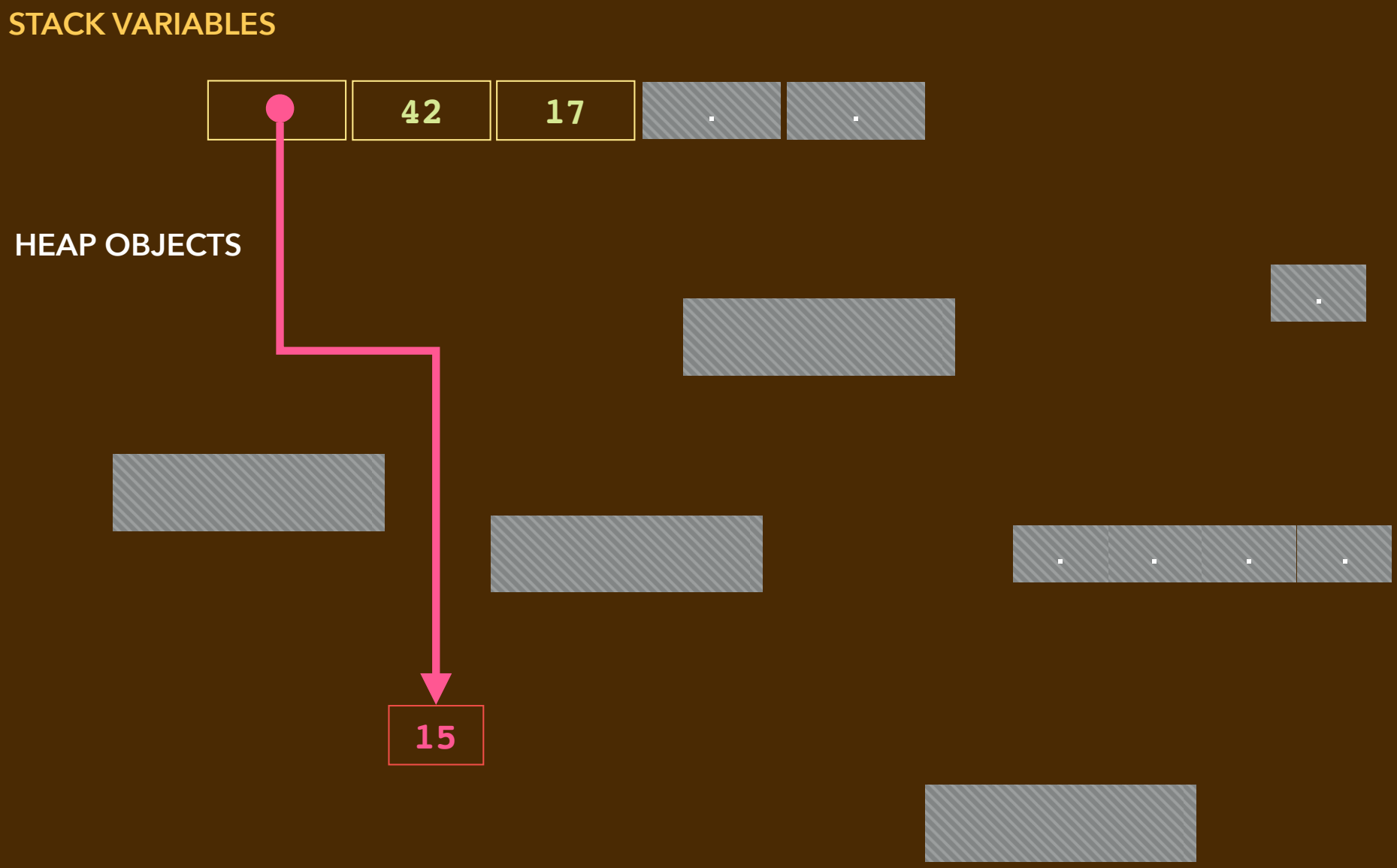
MEMORY DIAGRAM: MARK AND SWEEP



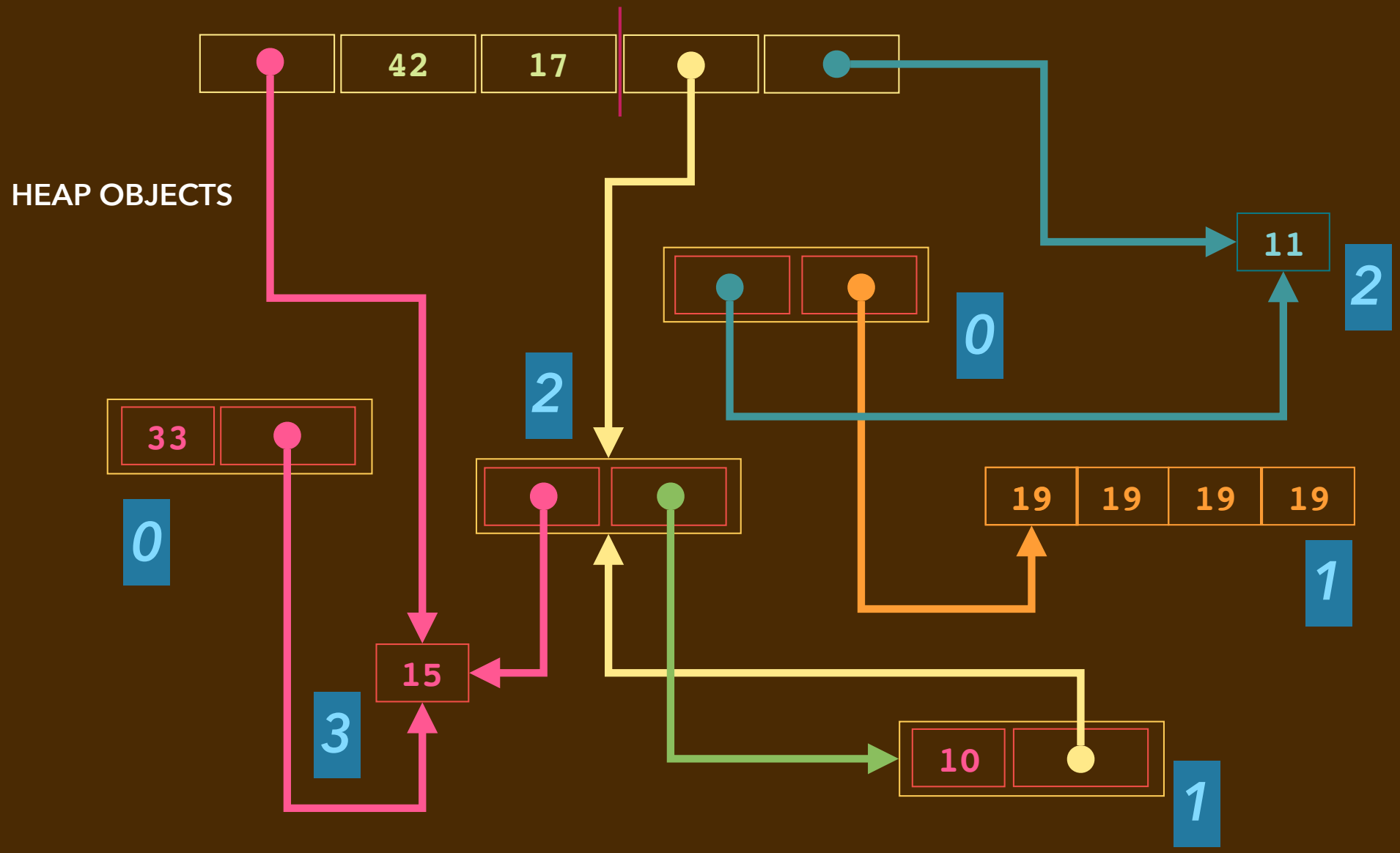
MEMORY DIAGRAM: MARK AND SWEEP



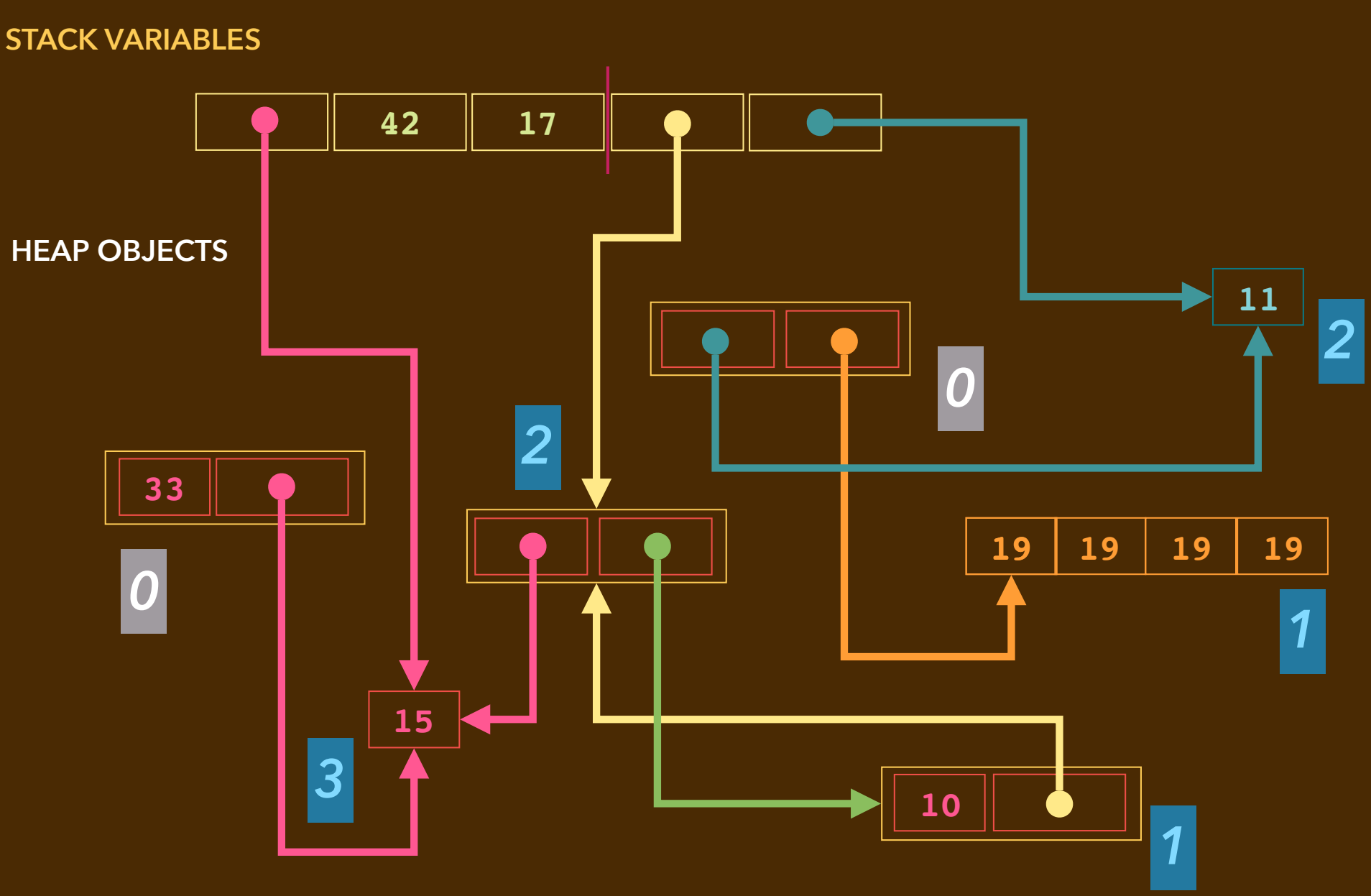
MEMORY DIAGRAM: MARK AND SWEEP



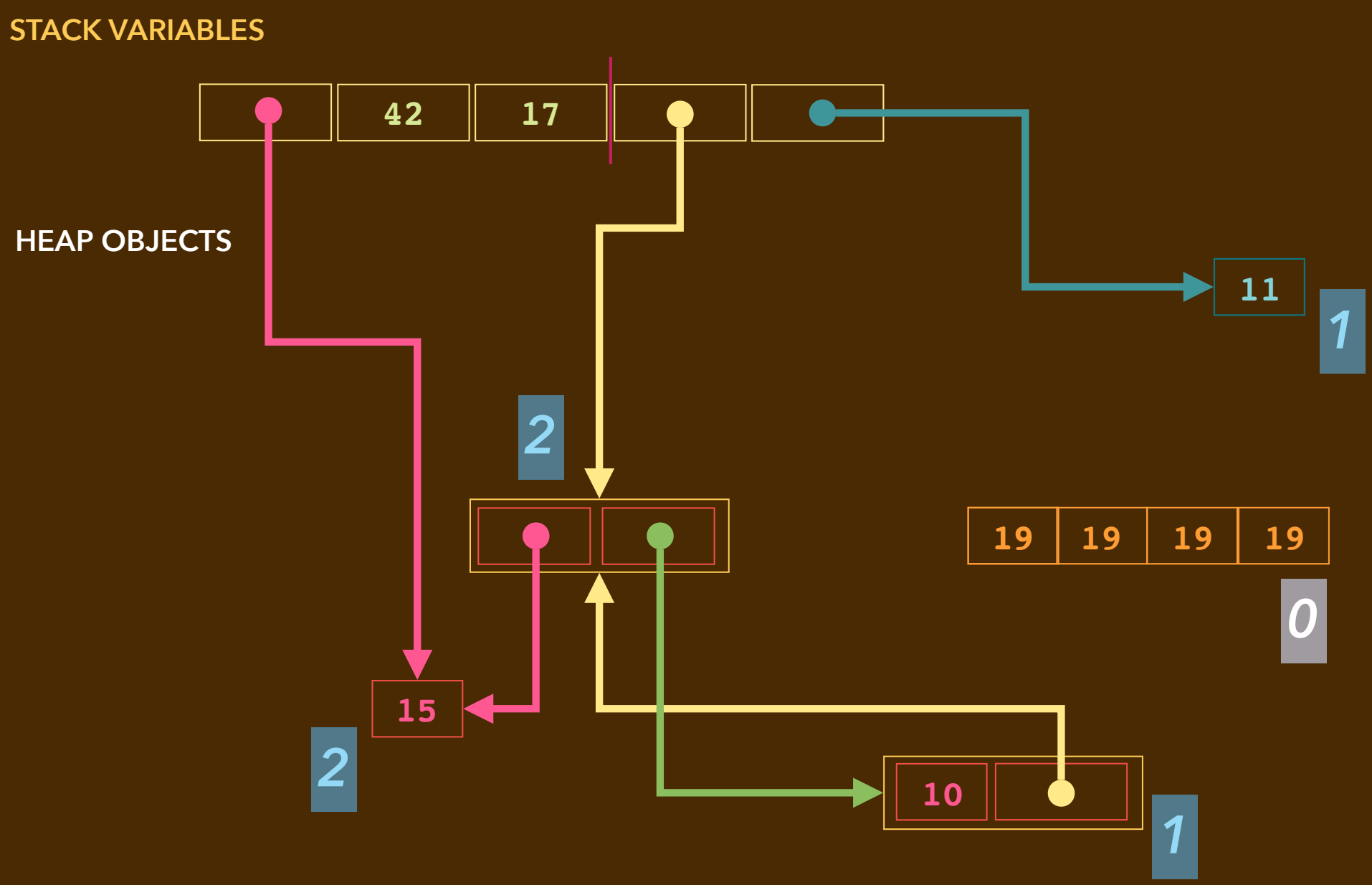
MEMORY DIAGRAM: REFERENCE COUNTS



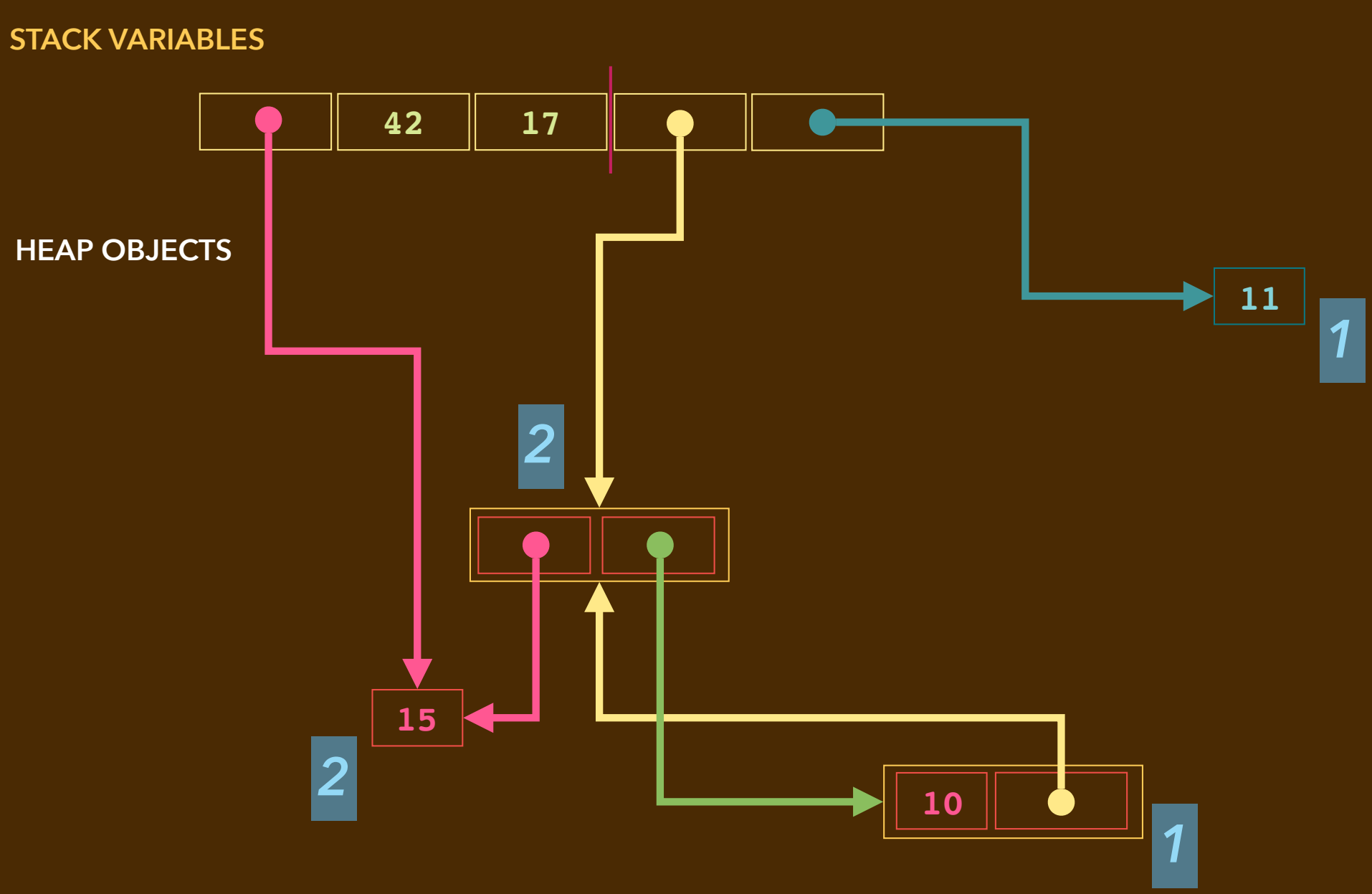
MEMORY DIAGRAM: REFERENCE COUNTS



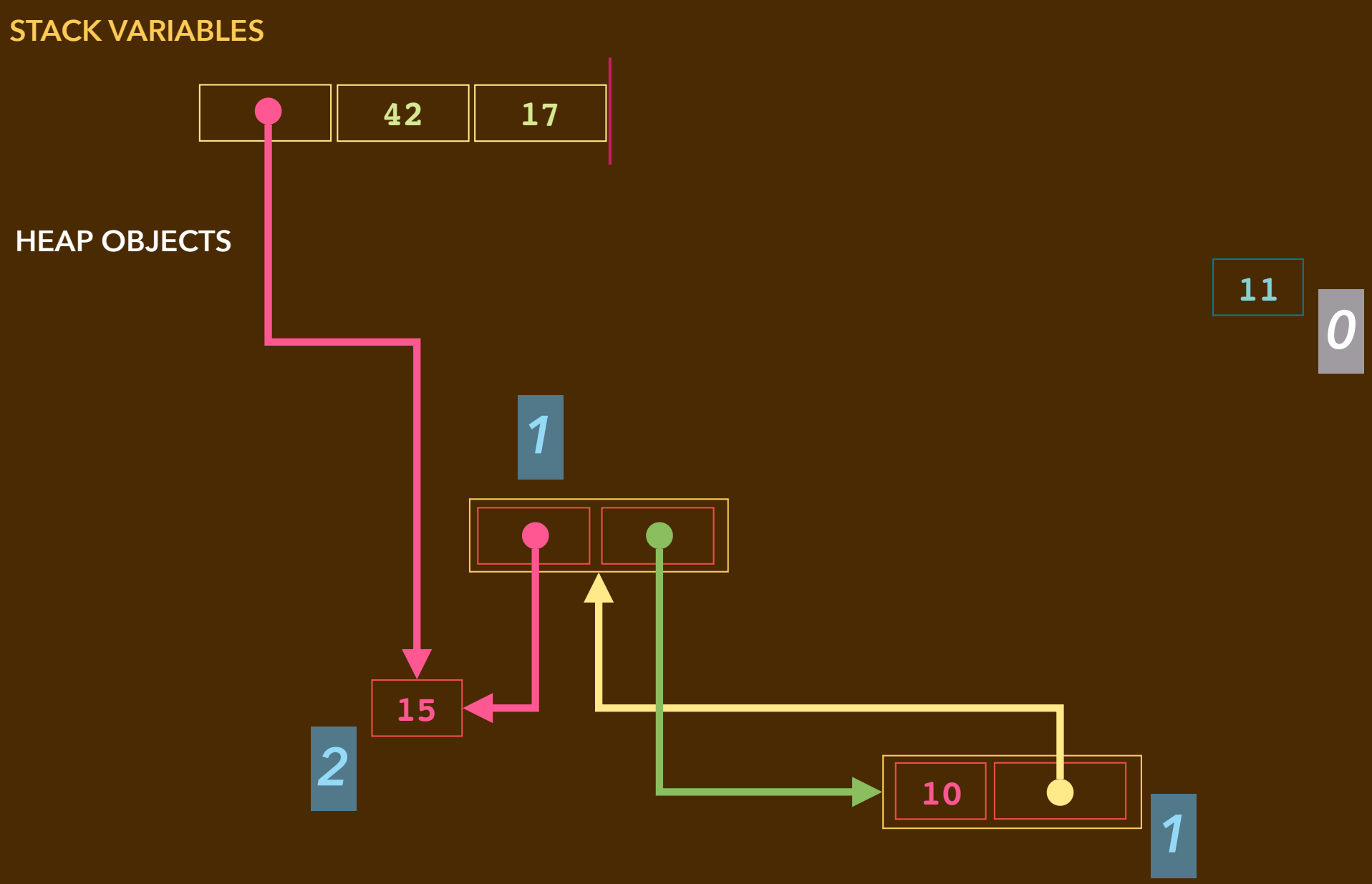
MEMORY DIAGRAM: REFERENCE COUNTS



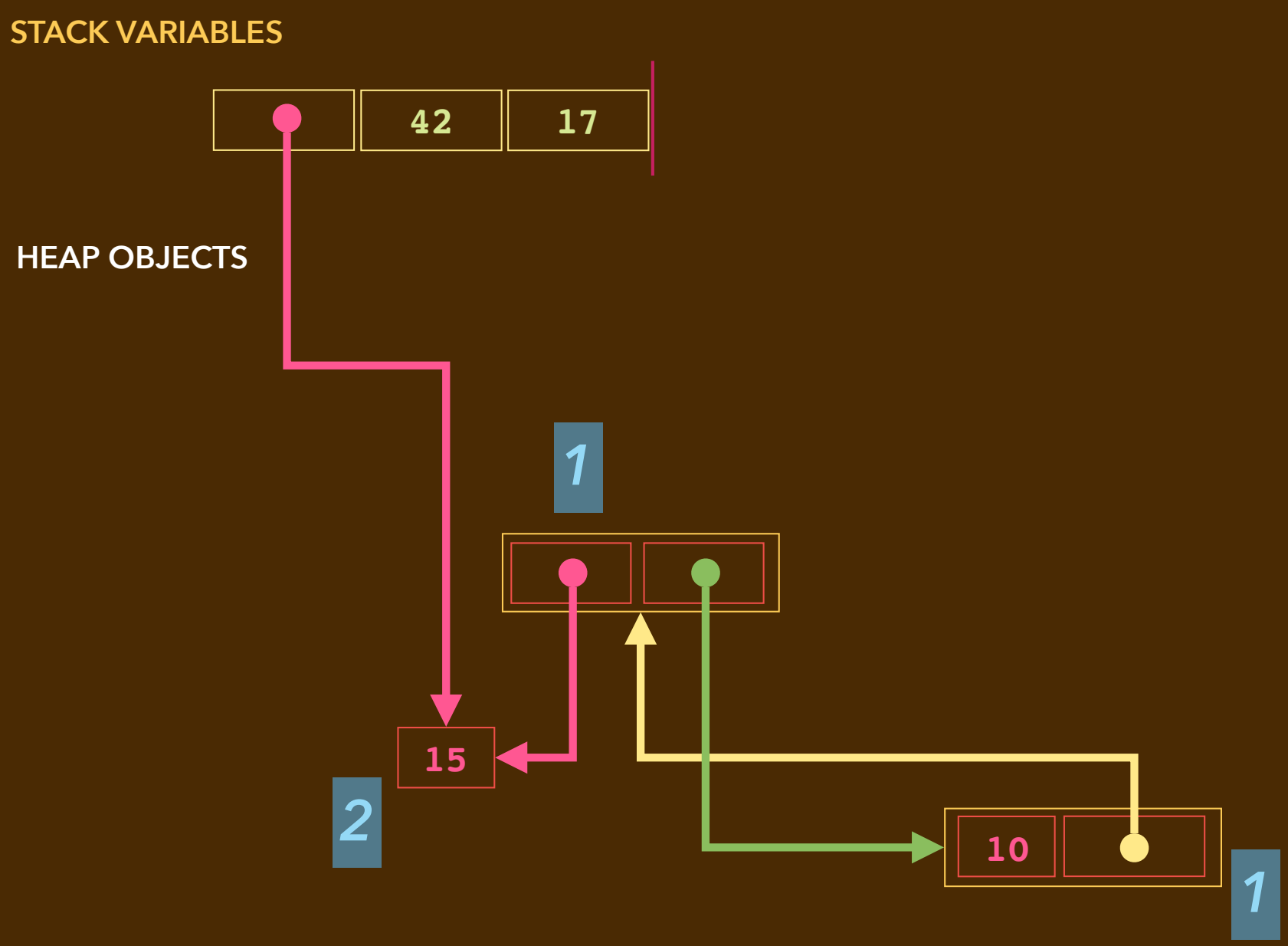
MEMORY DIAGRAM: REFERENCE COUNTS



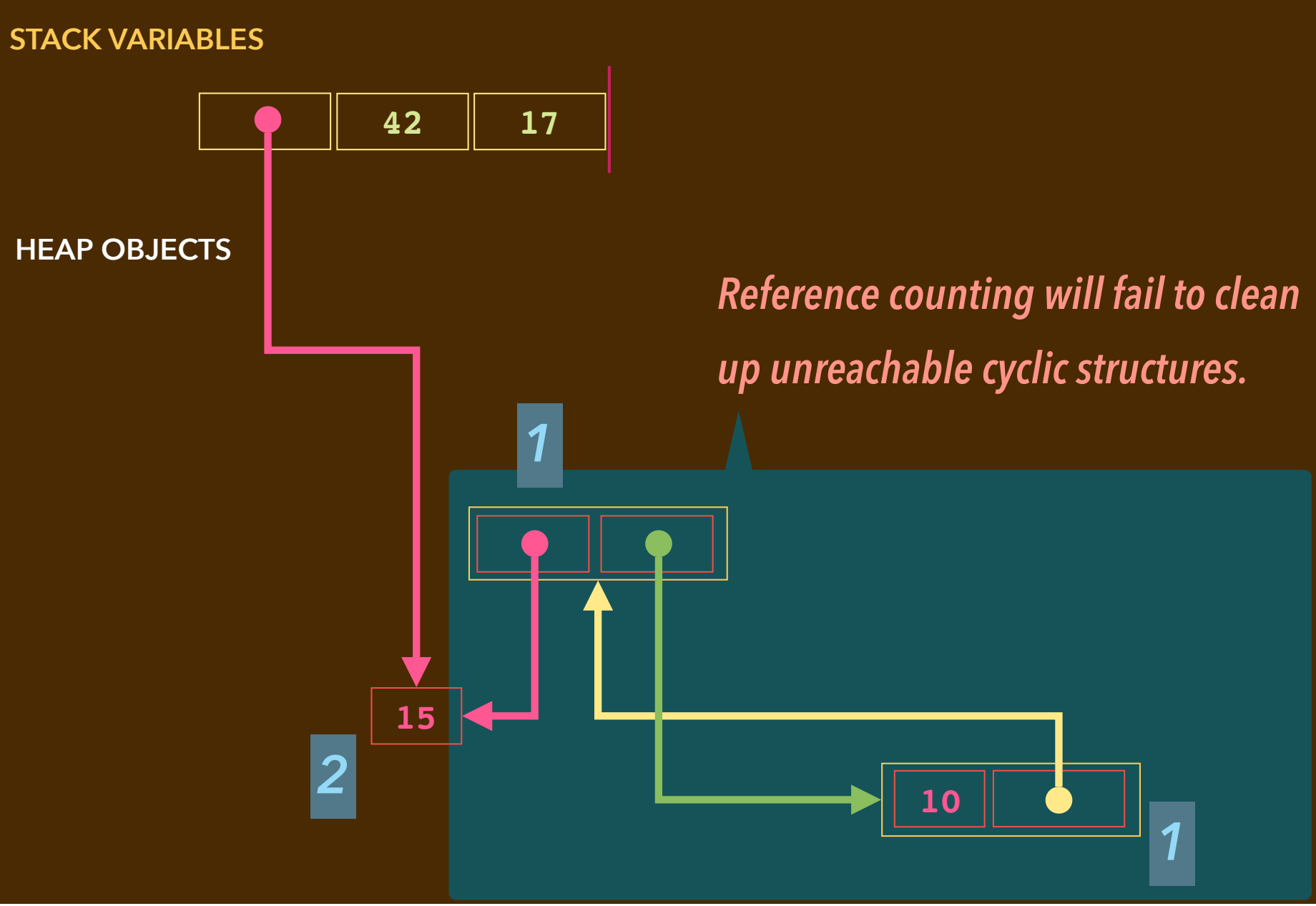
MEMORY DIAGRAM: REFERENCE COUNTS



MEMORY DIAGRAM: REFERENCE COUNTS



MEMORY DIAGRAM: REFERENCE COUNTS



RECALL: SMART POINTERS IN THE C++ STL

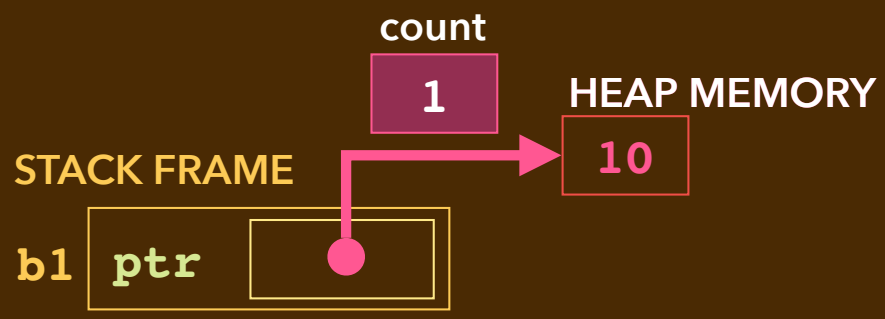
- The C++ STL provides three template types (`#include <memory>`)
  - `std::unique_ptr<T>`: used to reference an object owned by one code component (i.e. one variable). It cannot be *copied*. It can be *moved*.
  - `std::shared_ptr<T>`: used to reference an object shared by several code components. It 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.

RECALL: 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 look at use of `shared_ptr` in a linked list implementation.
- We look at use of `weak_ptr` in a doubly linked list implementation.

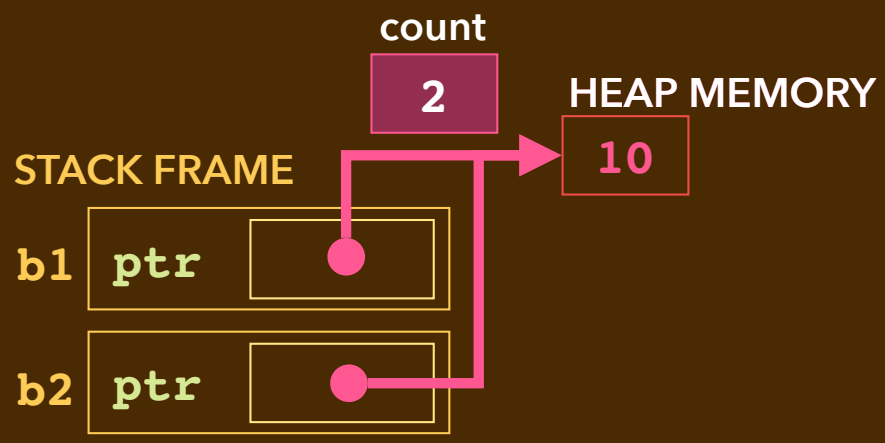
A SHARED_PTR BOX

```
#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; }
};

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



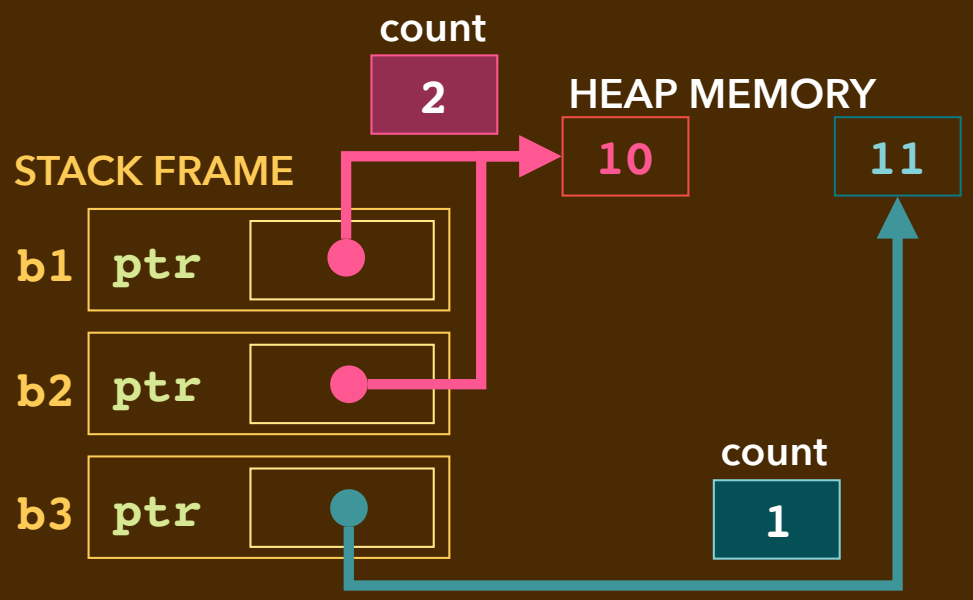
A SHARED_PTR BOX

```
#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; }
};

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



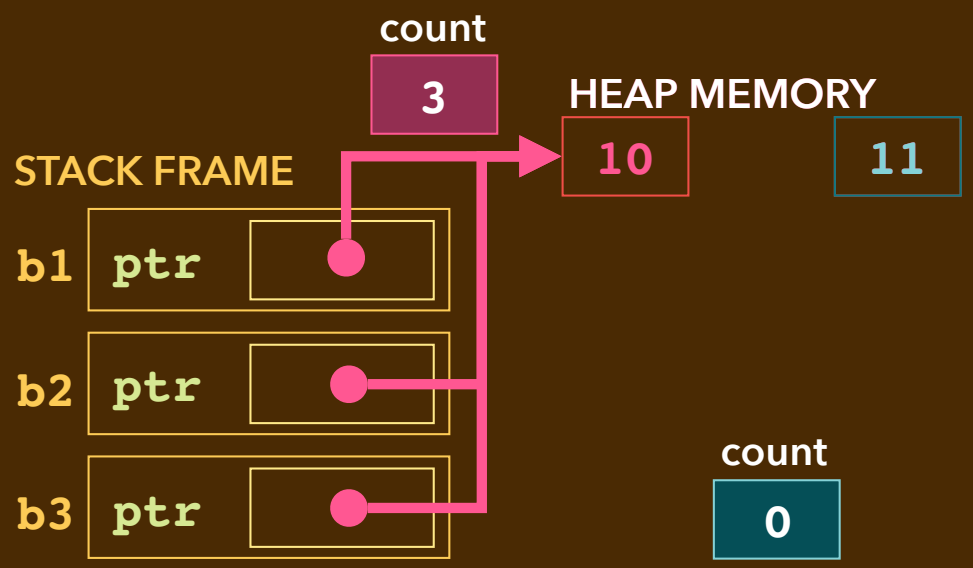
A SHARED_PTR BOX

```
#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; }
};

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; }
};

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

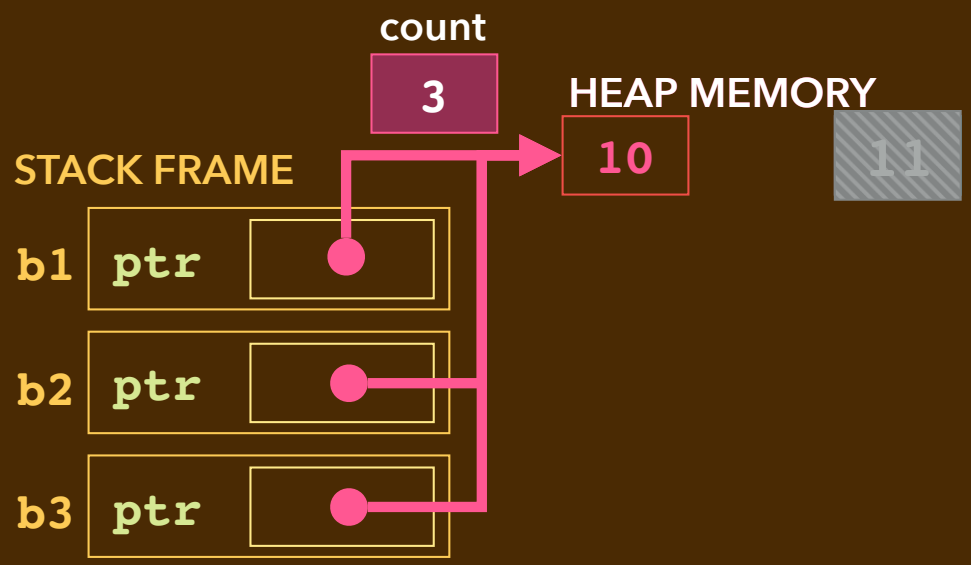


b1.ptr count increments to 3

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; }
};

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

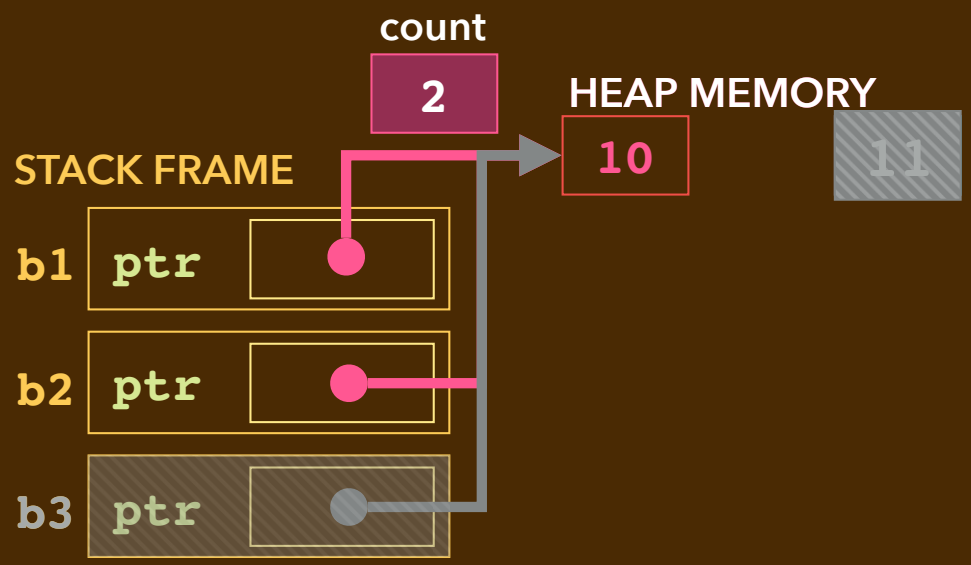


old b3.ptr 's raw pointer is deleted

A SHARED_PTR BOX

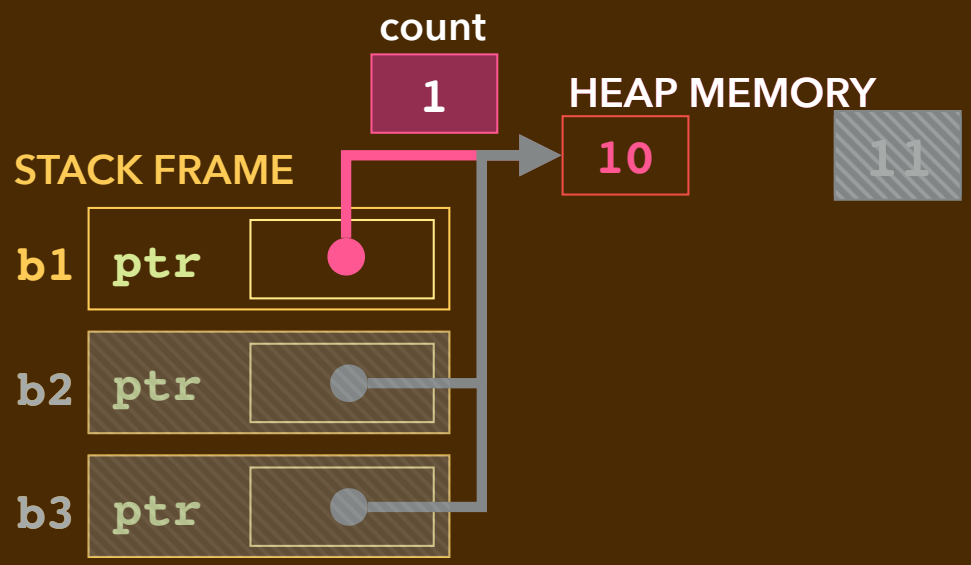
```
#include <memory>
class Box {
public:
  std::shared_ptr<int> ptr;
  Box(int value) : {new int[value]} { }
  Box(const Box& b) : {b.ptr} { }
  Box& operator=(const Box& b) { ptr = b.ptr; return *this; }
};

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) : {new int[value]} { }
  Box(const Box& b) : {b.ptr} { }
  Box& operator=(const Box& b) { ptr = b.ptr; return *this; }
};

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

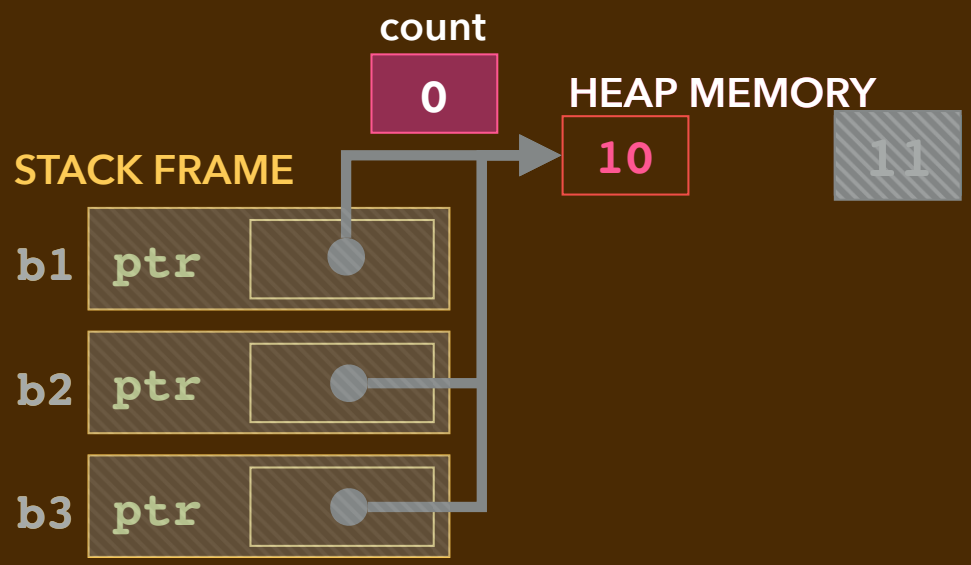


Box b3 loses scope; count decrements.

Box b2 loses scope; count decrements.

A SHARED_PTR BOX

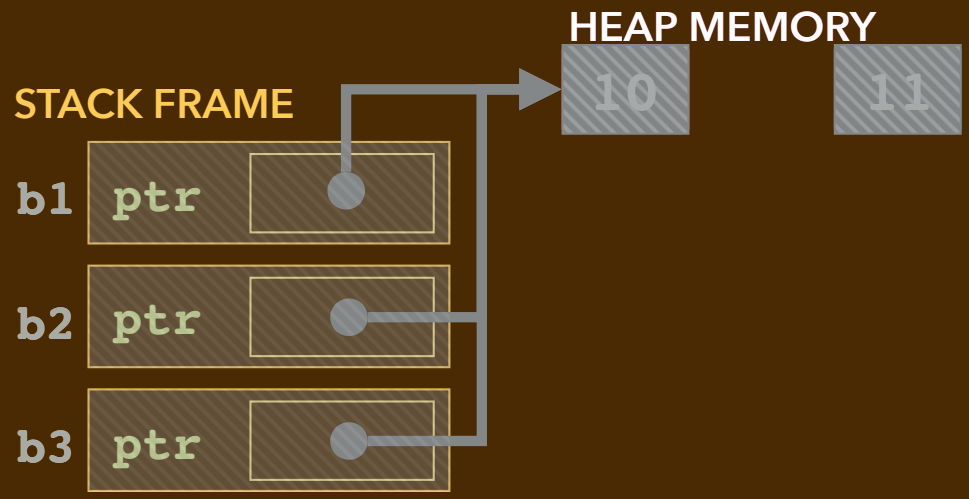
```
#include <memory>
class Box {
public:
  std::shared_ptr<int> ptr;
  Box(int value) : {new int[value]} { }
  Box(const Box& b) : {b.ptr} { }
  Box& operator=(const Box& b) { ptr = b.ptr; return *this; }
};

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) : {new int[value]} { }
  Box(const Box& b) : {b.ptr} { }
  Box& operator=(const Box& b) { ptr = b.ptr; return *this; }
};

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

- Last 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;
      }
    }
  }
}
```

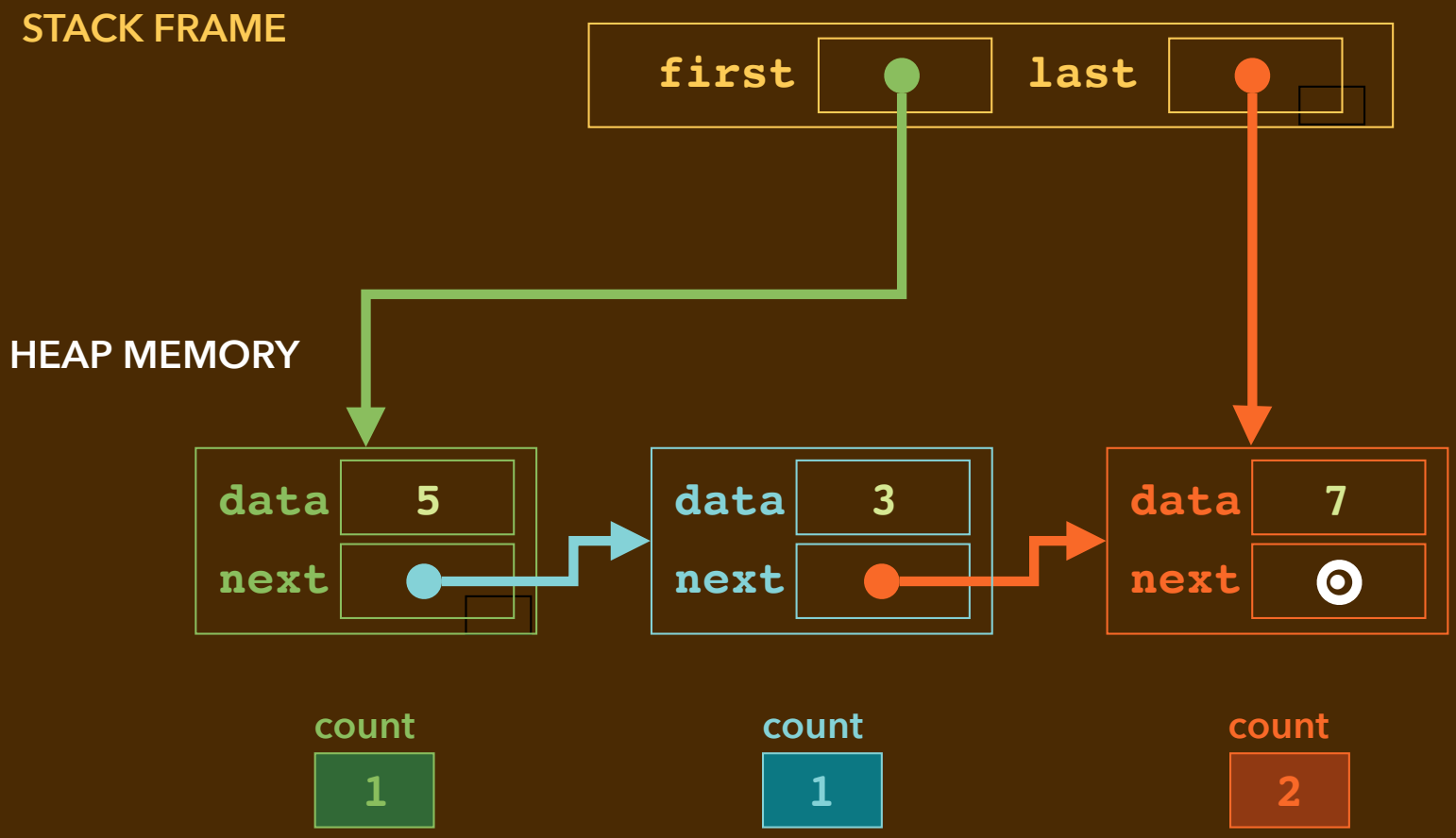
Here `current` loses scope; removed node's reference count goes to 0 and is reclaimed.

NO DESTRUCTOR CODE NEEDED

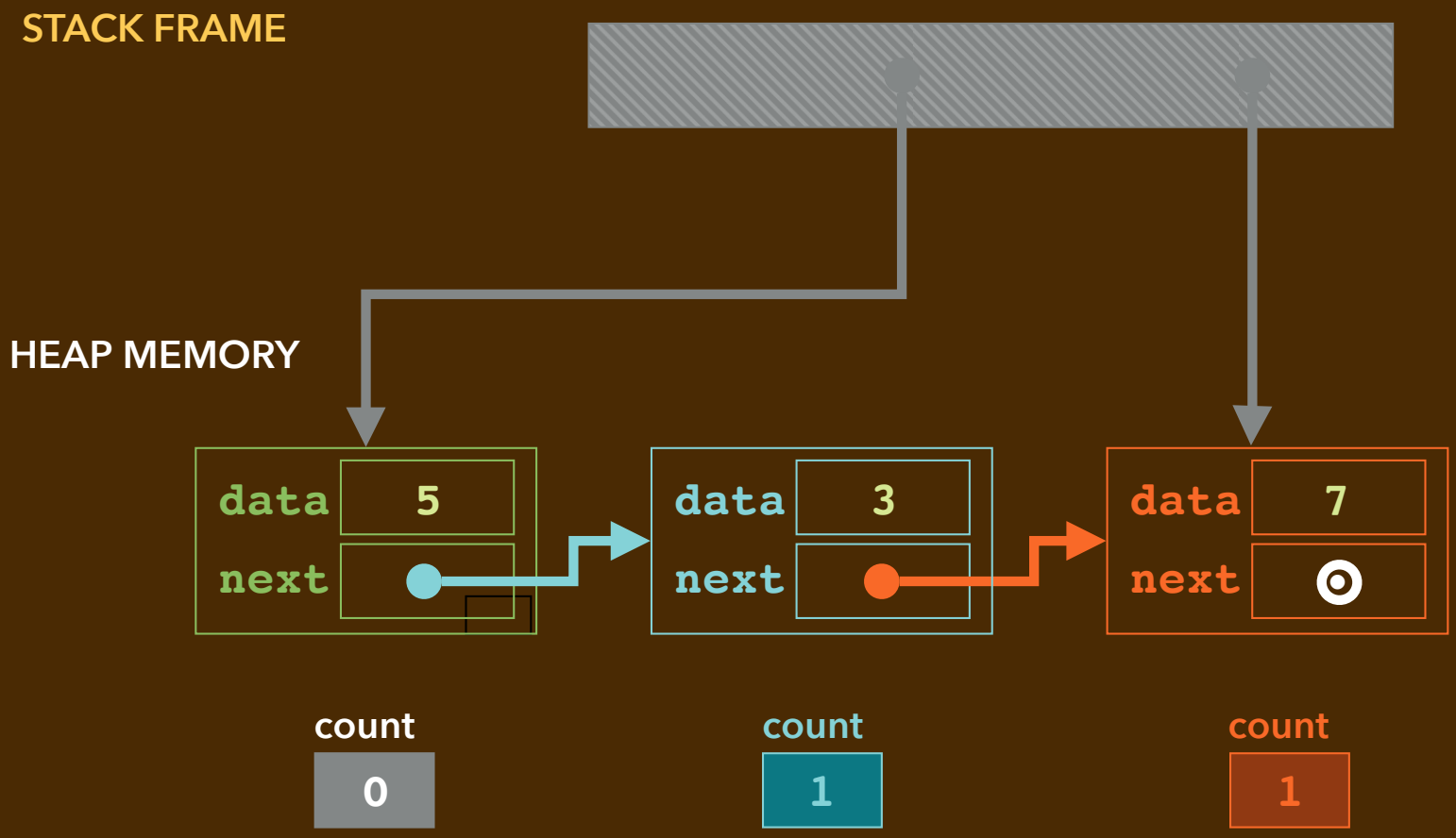
```
class node {
 int data;
  std::shared_ptr<node> next;
  node(int value) : data {value}, next {nullptr} { }
 ~node(void) { }
};
class llist {
  std::shared_ptr<node> first;
  std::shared_ptr<node> last;
  llist(void) : first {nullptr}, last {nullptr} { }
 ~llist(void) { // NOTHING HERE!! }
  void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

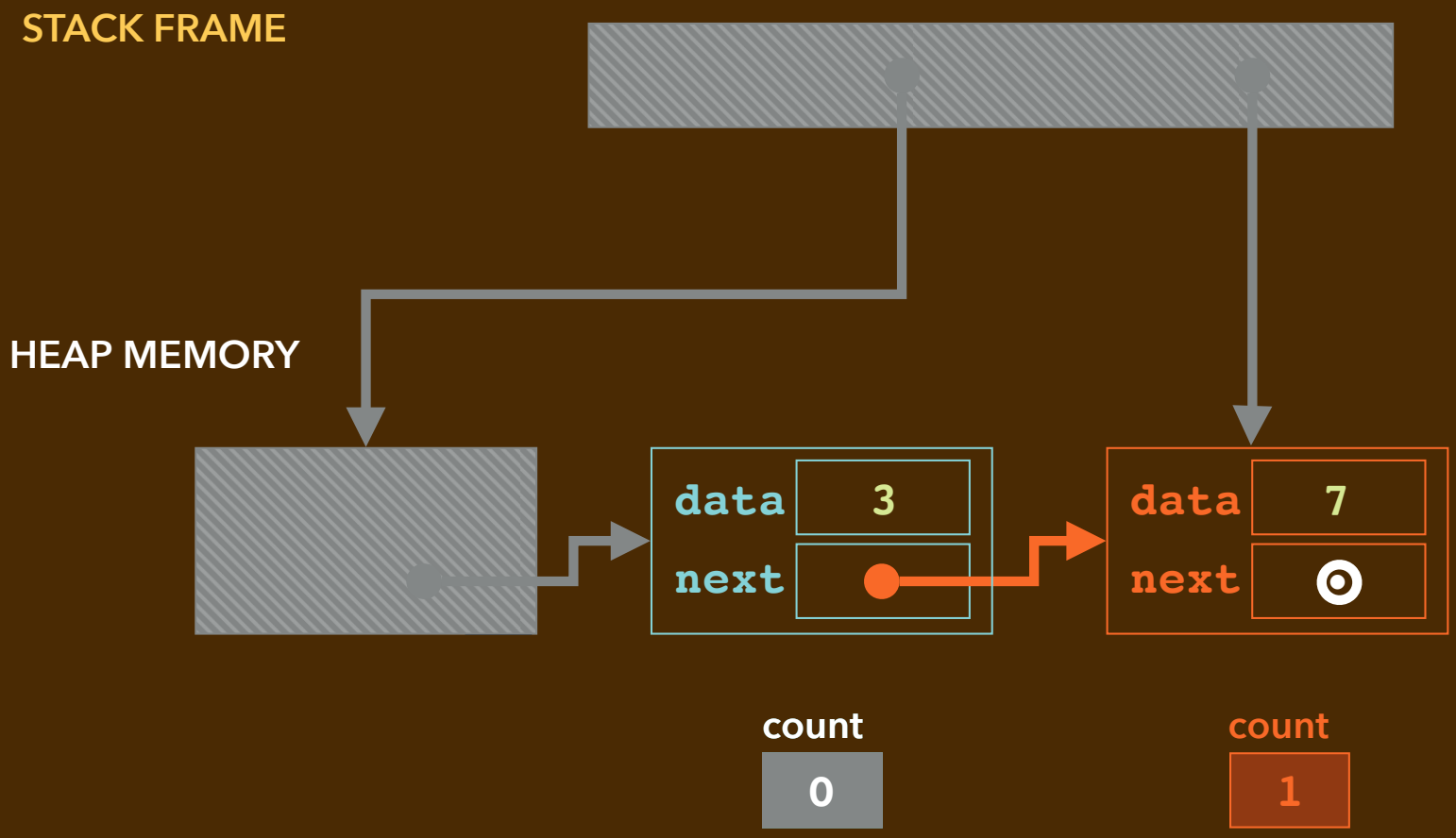
WHY IT WORKS: SINGLY LINKED LIST



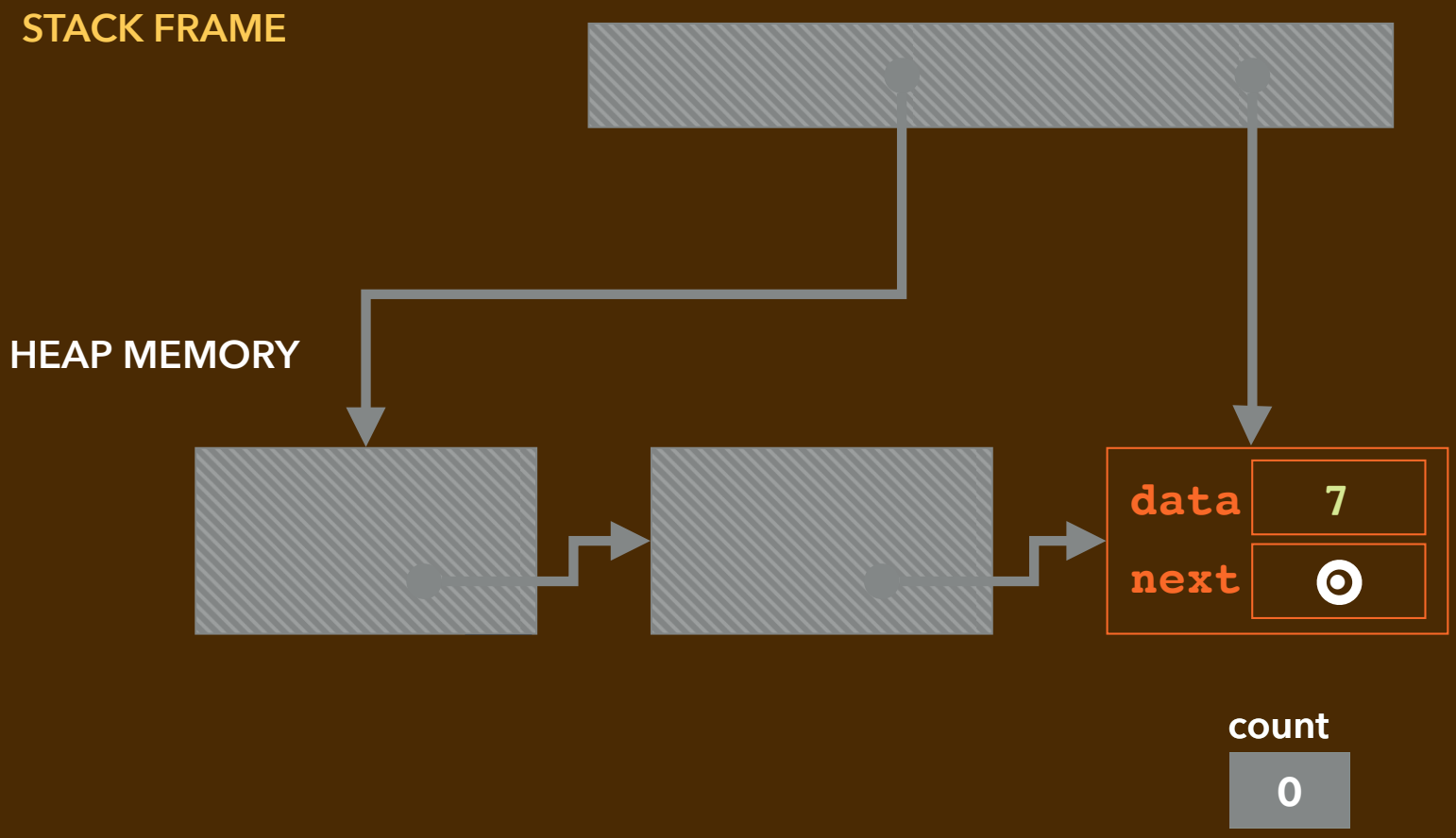
WHY IT WORKS: SINGLY LINKED LIST



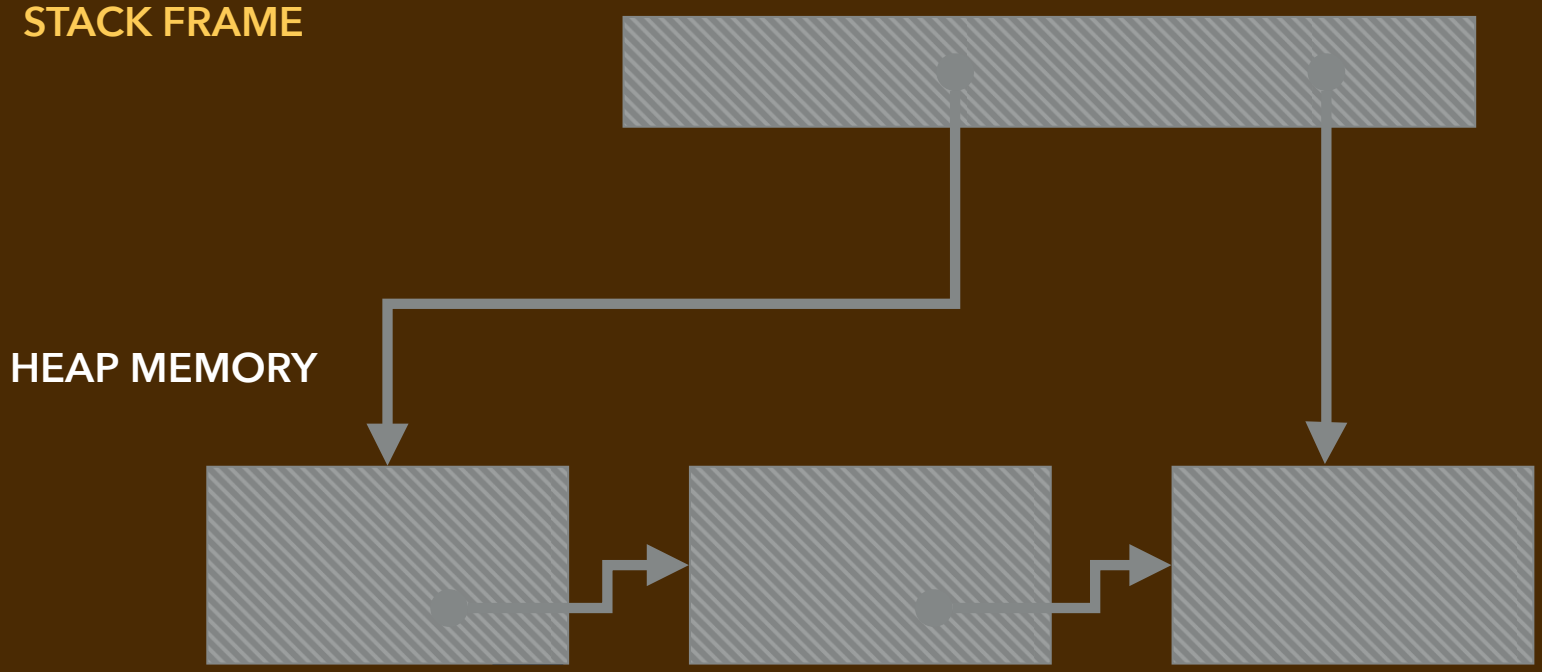
WHY IT WORKS: SINGLY LINKED LIST



WHY IT WORKS: SINGLY LINKED LIST



WHY IT WORKS: SINGLY LINKED LIST



A SHARED_PTR SINGLY LINKED LIST SUMMARY

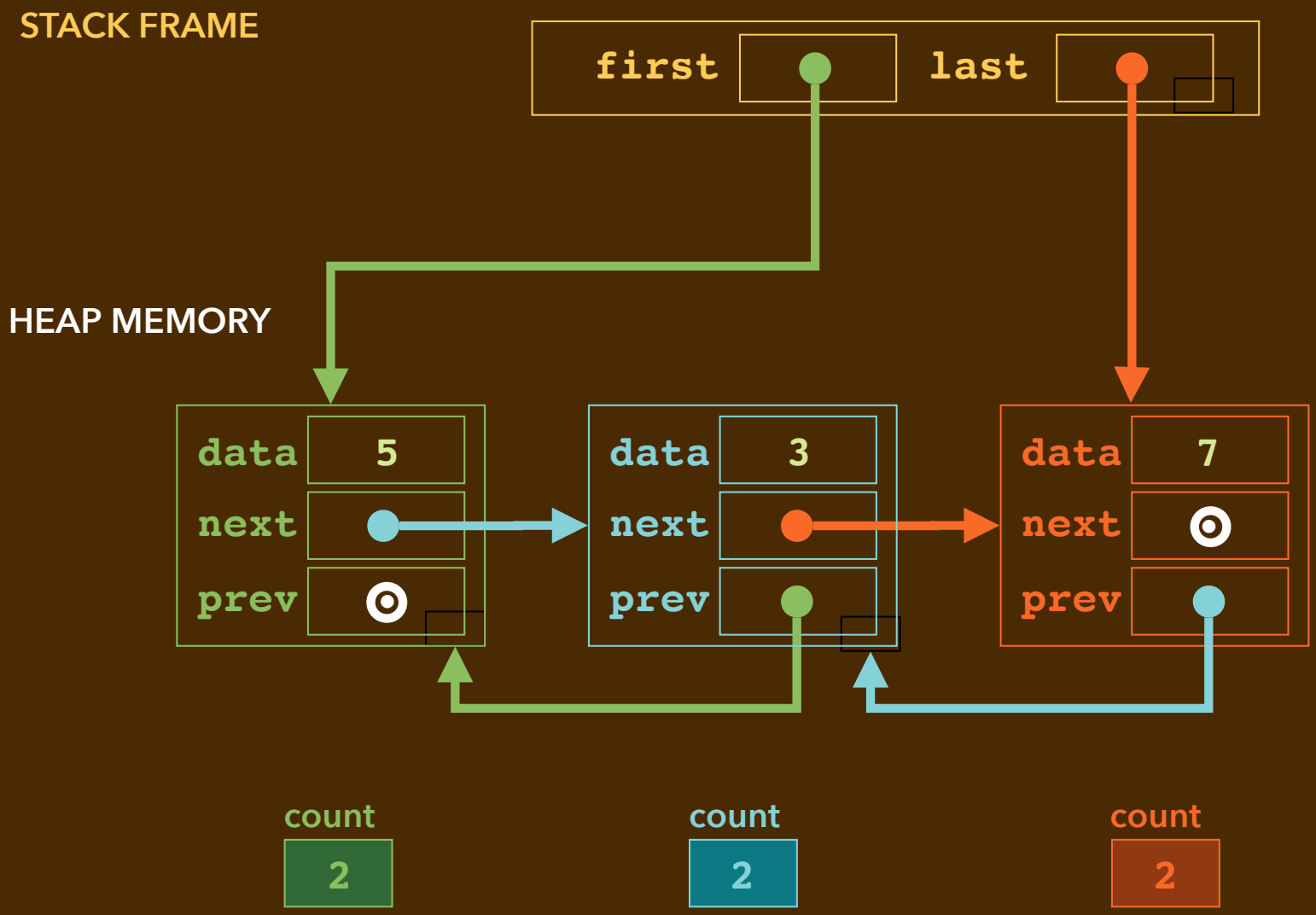
- By using **`shared_ptr`**, every reference to a node is counted.
- When a new node is made, a **`shared_ptr`** is invented with a count of 1.
  - It has an underlying raw pointer obtained from **`new`**.
- When a relink happens:
  - ➡ A non-null reference's count decrements.
  - ➡ Another reference's 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

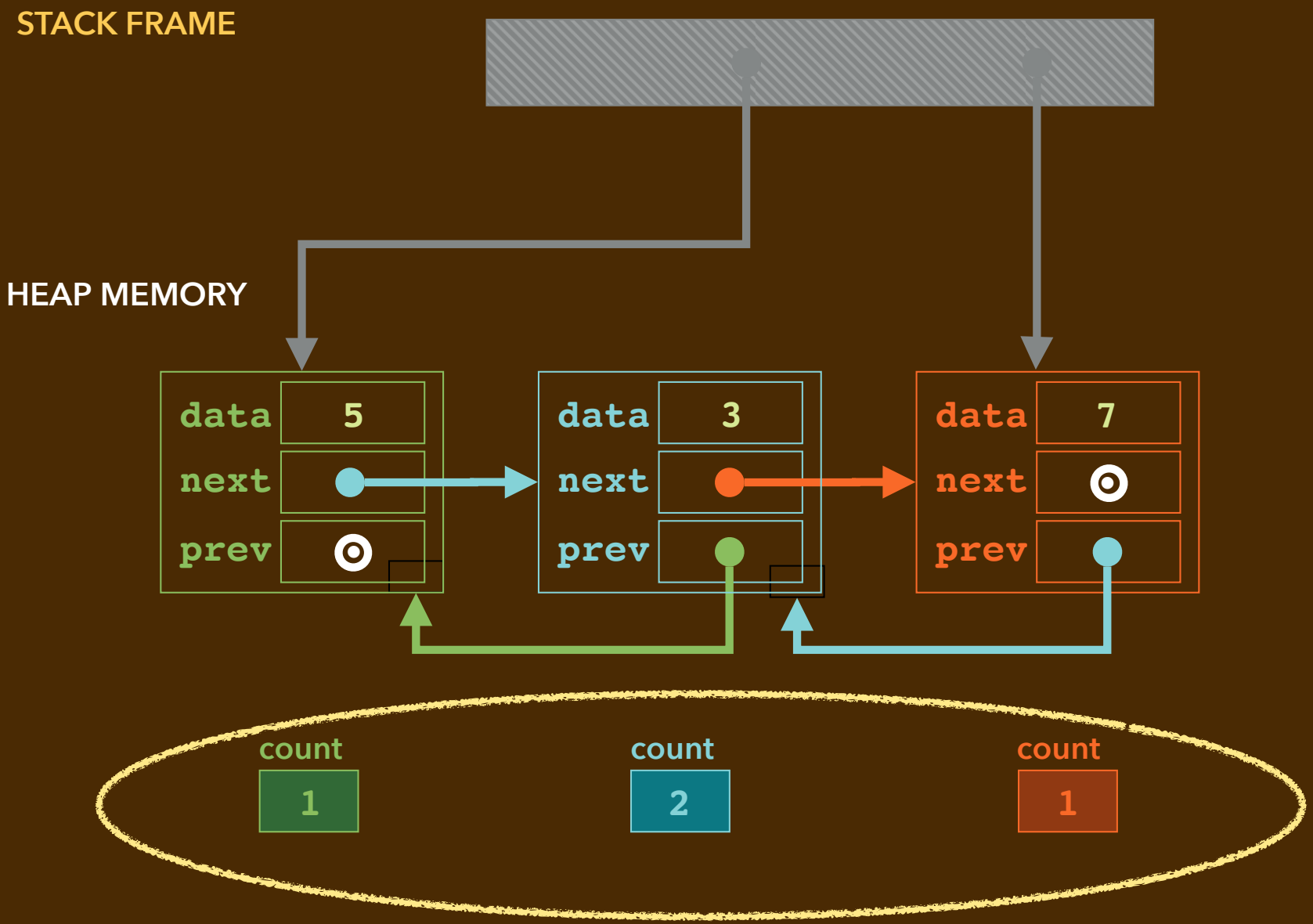
```
#include <memory>

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

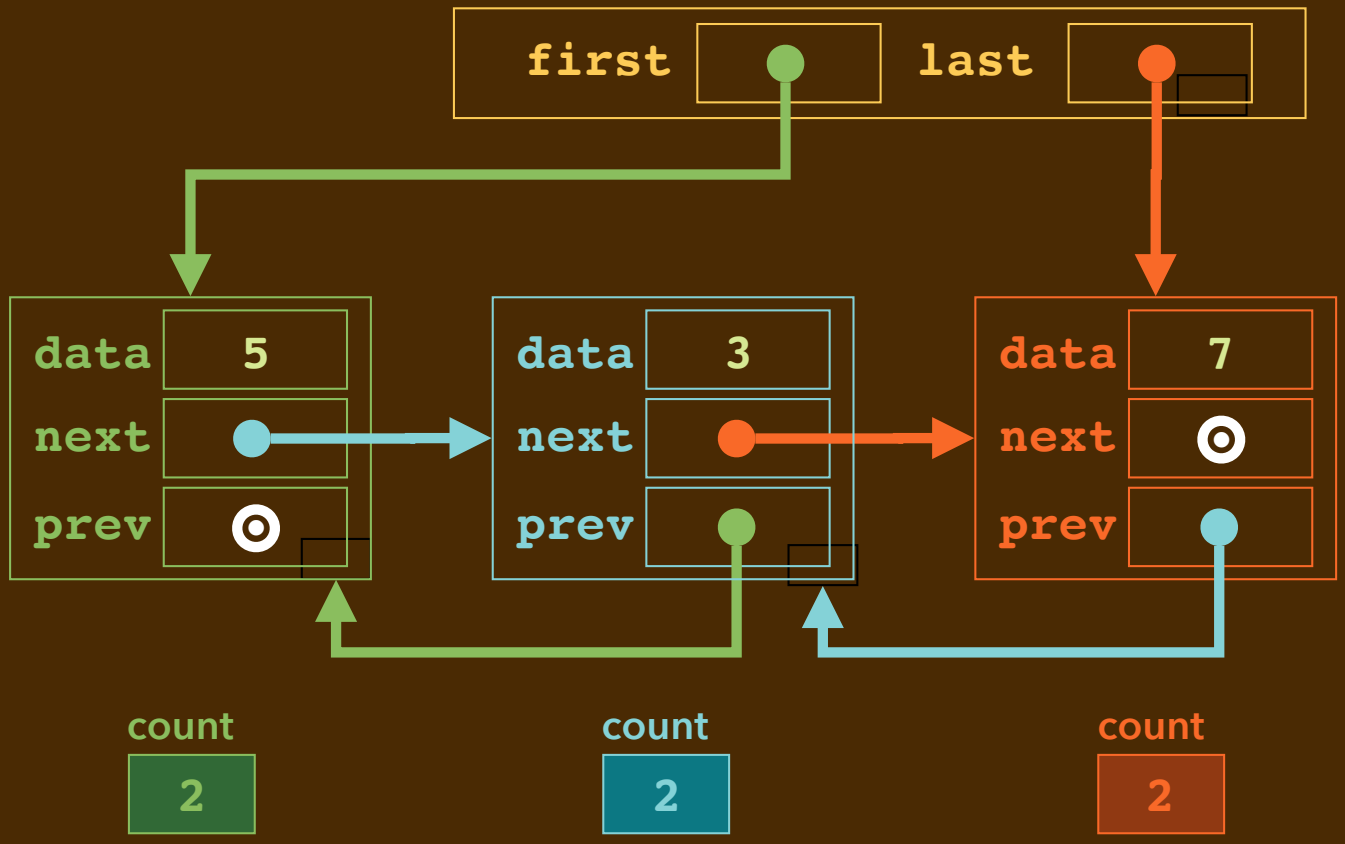


FIX #1: A DESTRUCTOR THAT UNLINKS PREV POINTERS

```
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);
}
```

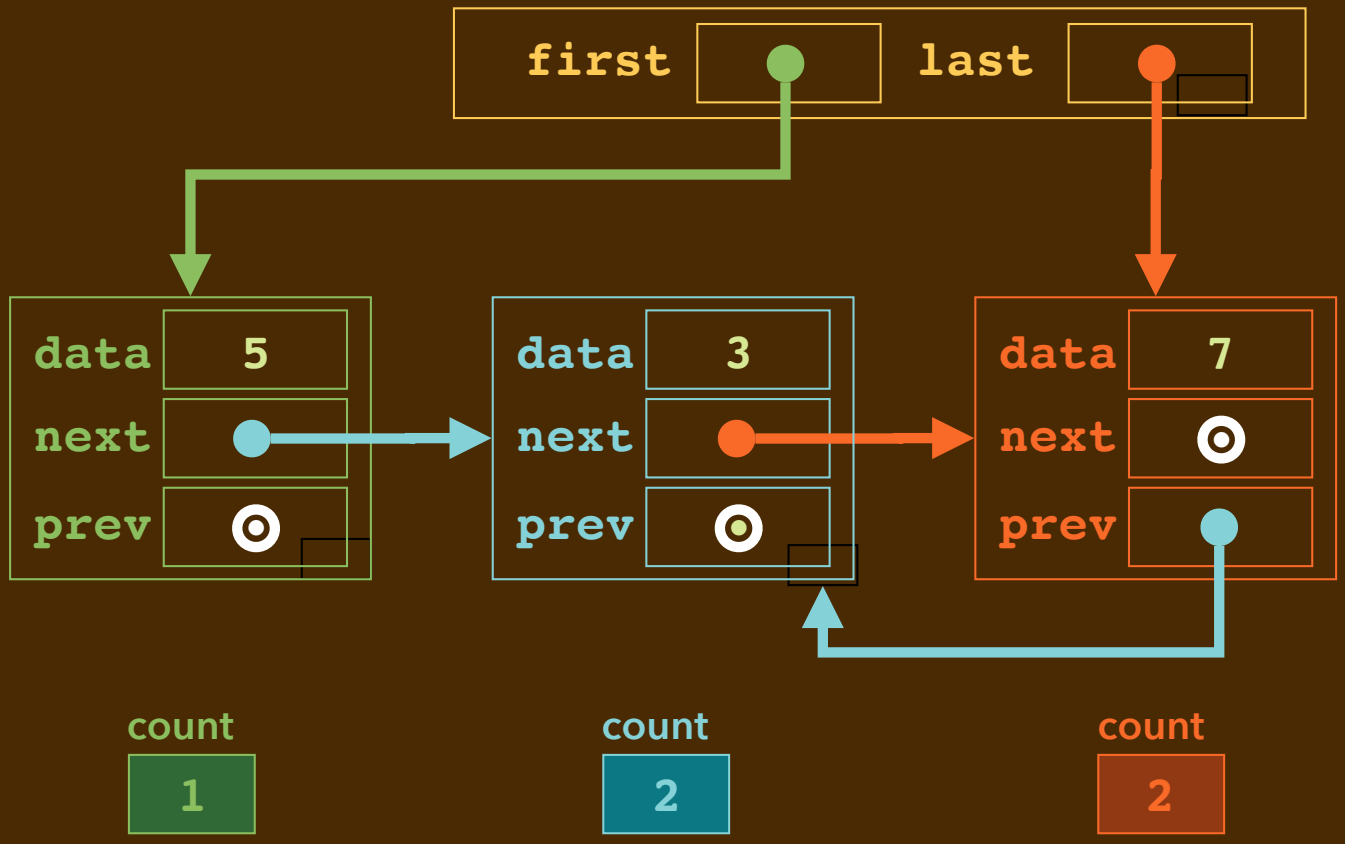
FIX #1: A DESTRUCTOR THAT UNLINKS EACH PREV

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



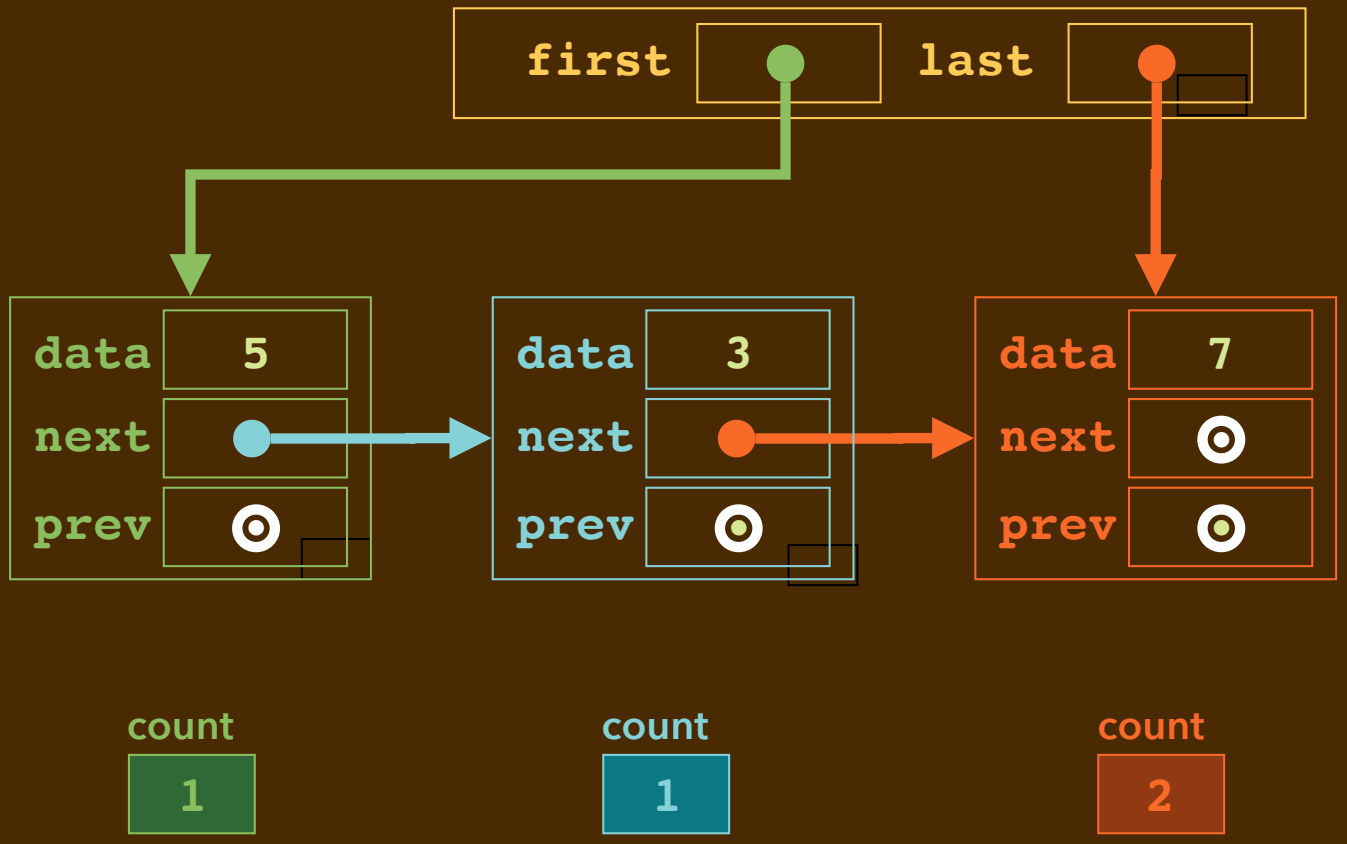
FIX #1: A DESTRUCTOR THAT UNLINKS EACH PREV

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



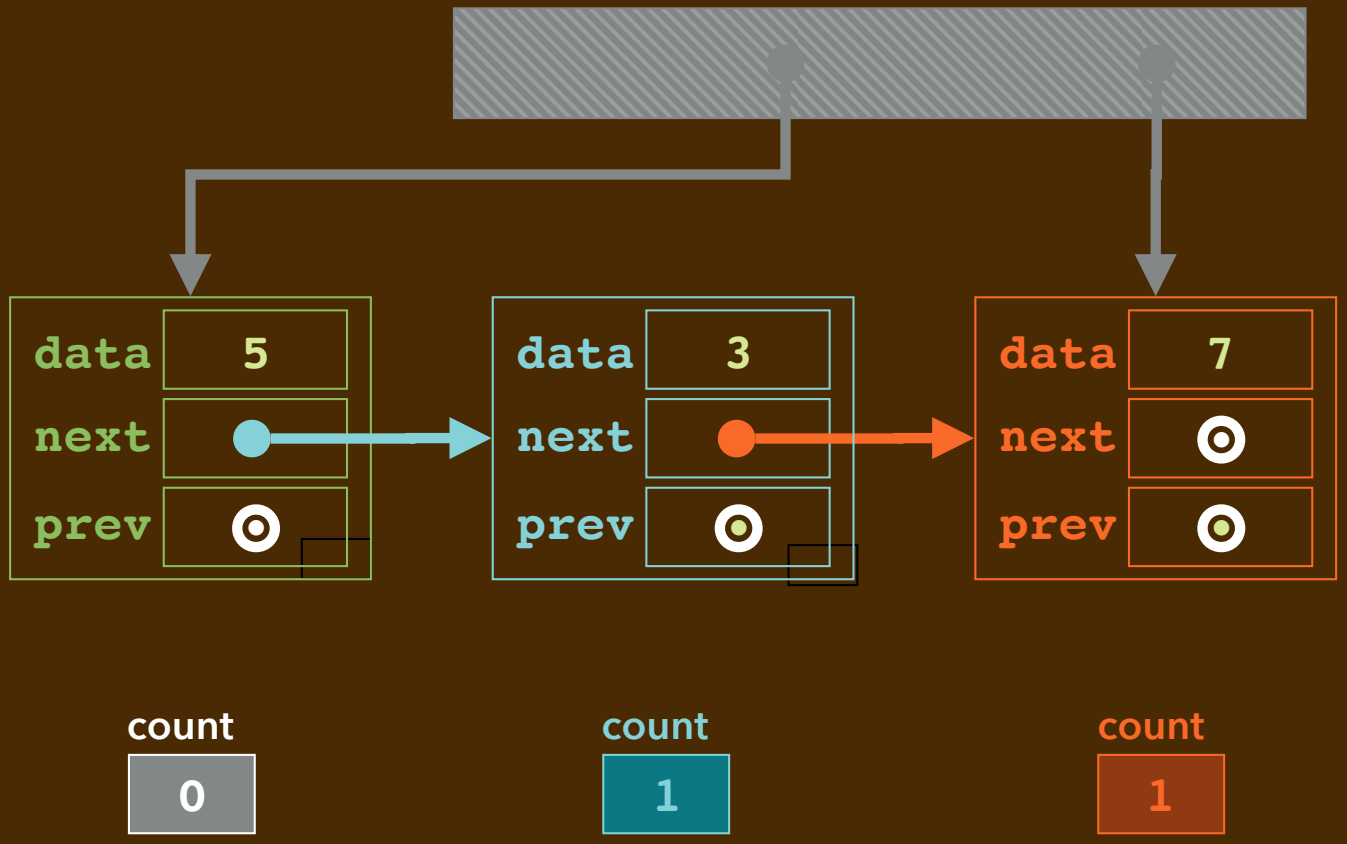
FIX #1: A DESTRUCTOR THAT UNLINKS EACH PREV

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



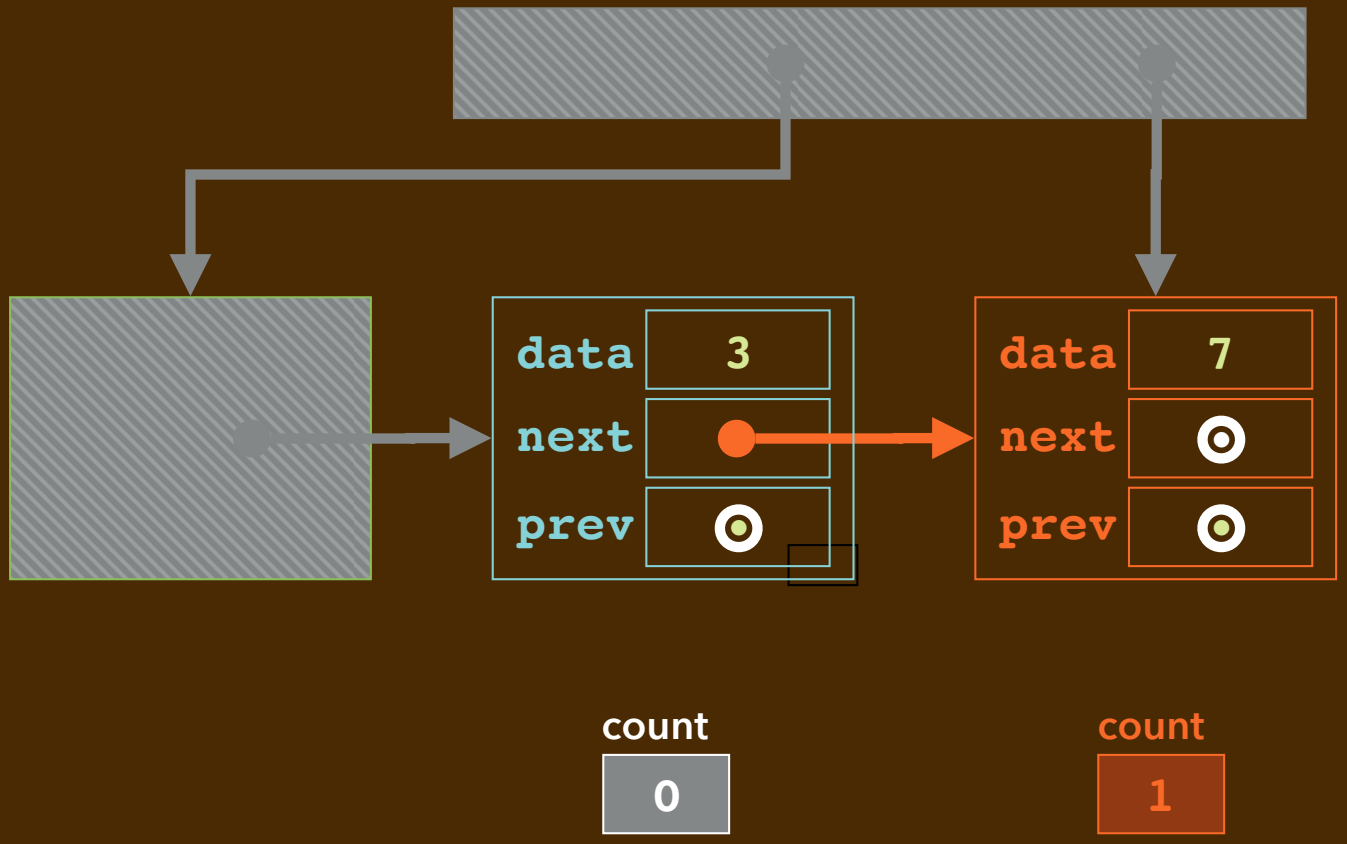
FIX #1: A DESTRUCTOR THAT UNLINKS EACH PREV

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



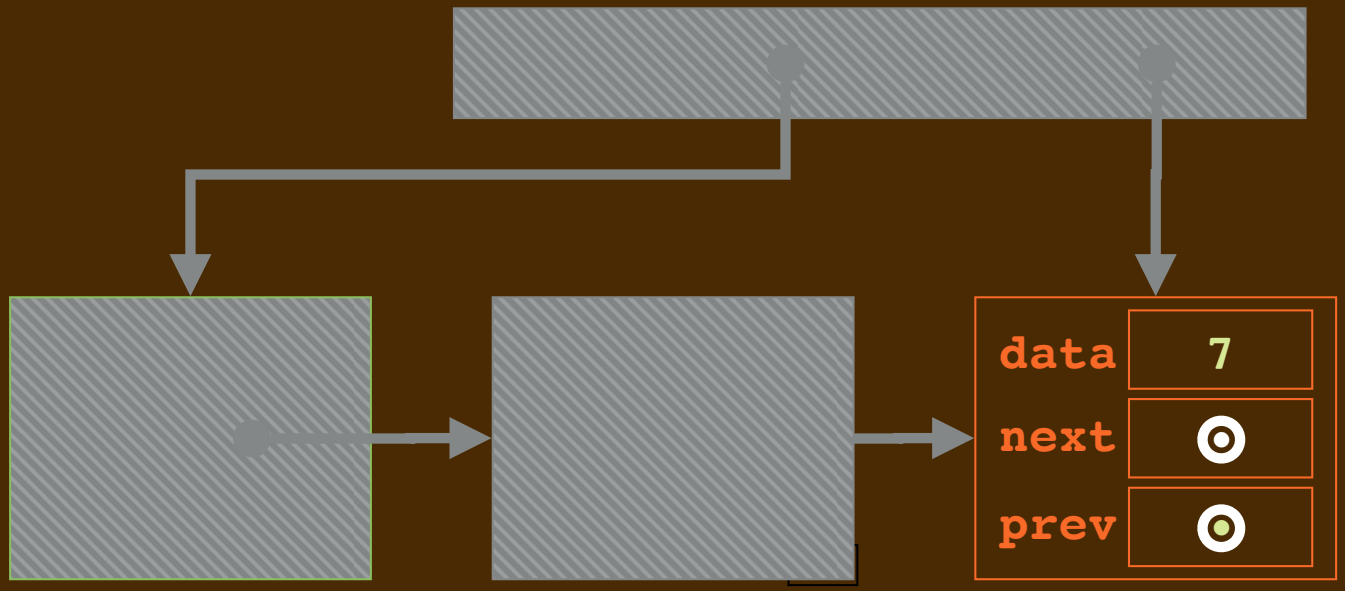
FIX #1: A DESTRUCTOR THAT UNLINKS EACH PREV

```
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;
  }
}
```

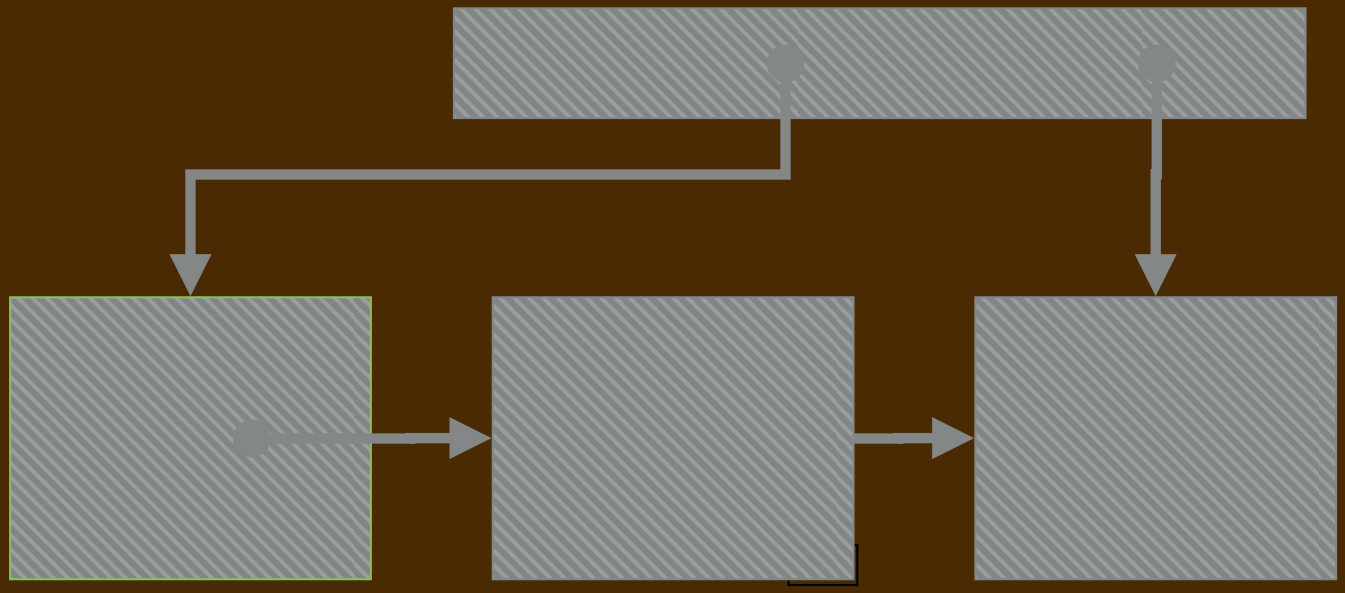


count

0

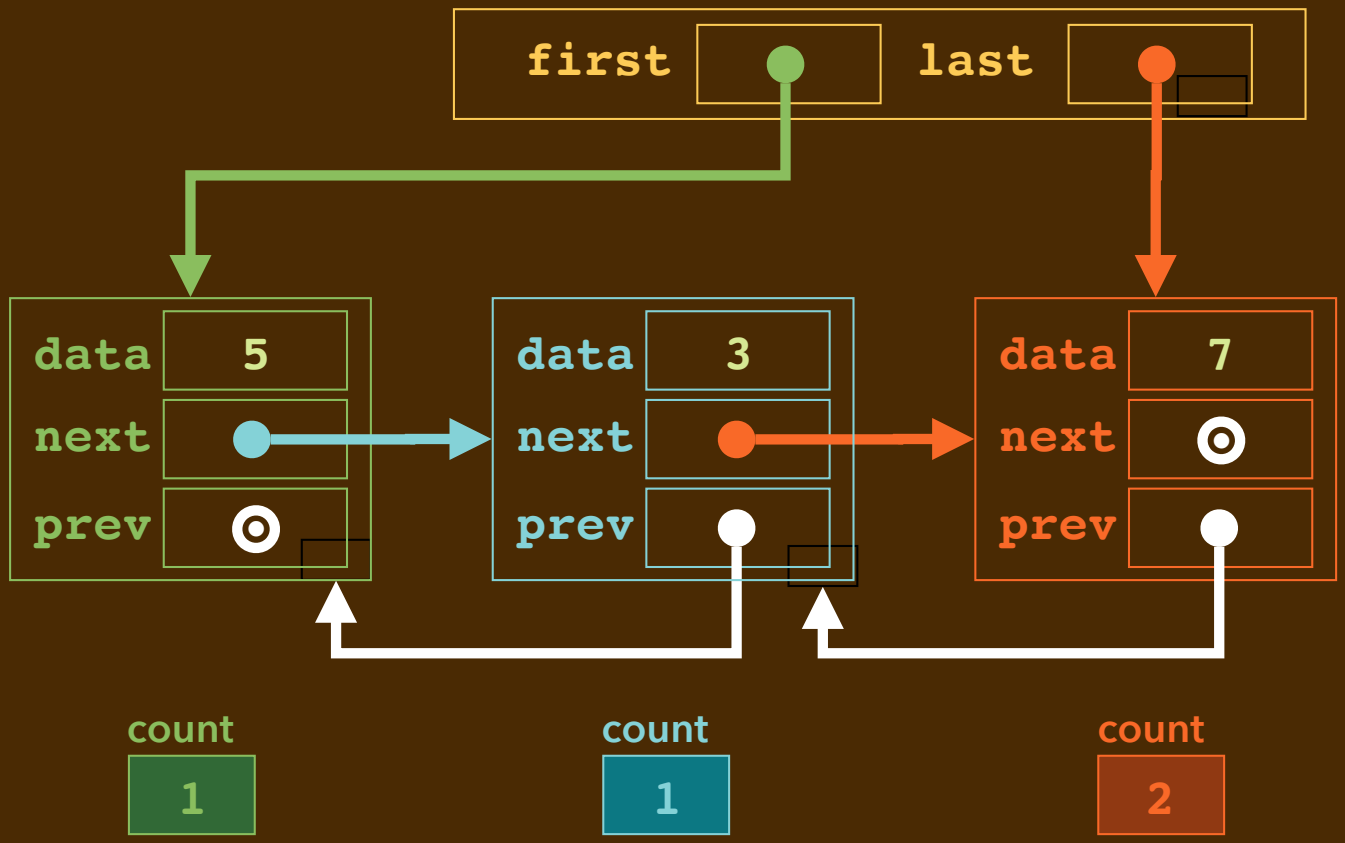
FIX #1: A DESTRUCTOR THAT UNLINKS EACH PREV

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



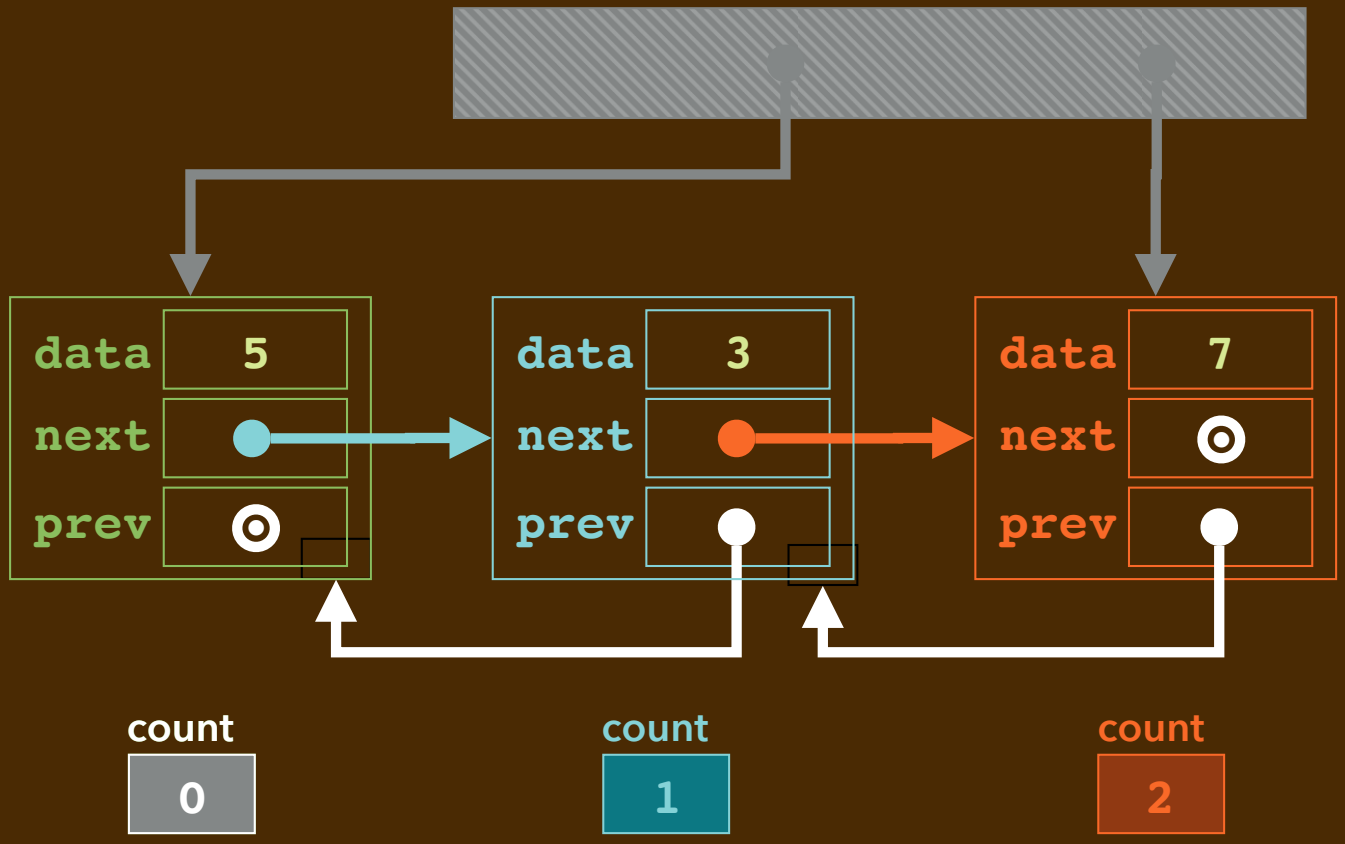
FIX #2: PREV POINTERS THAT DON'T COUNT

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



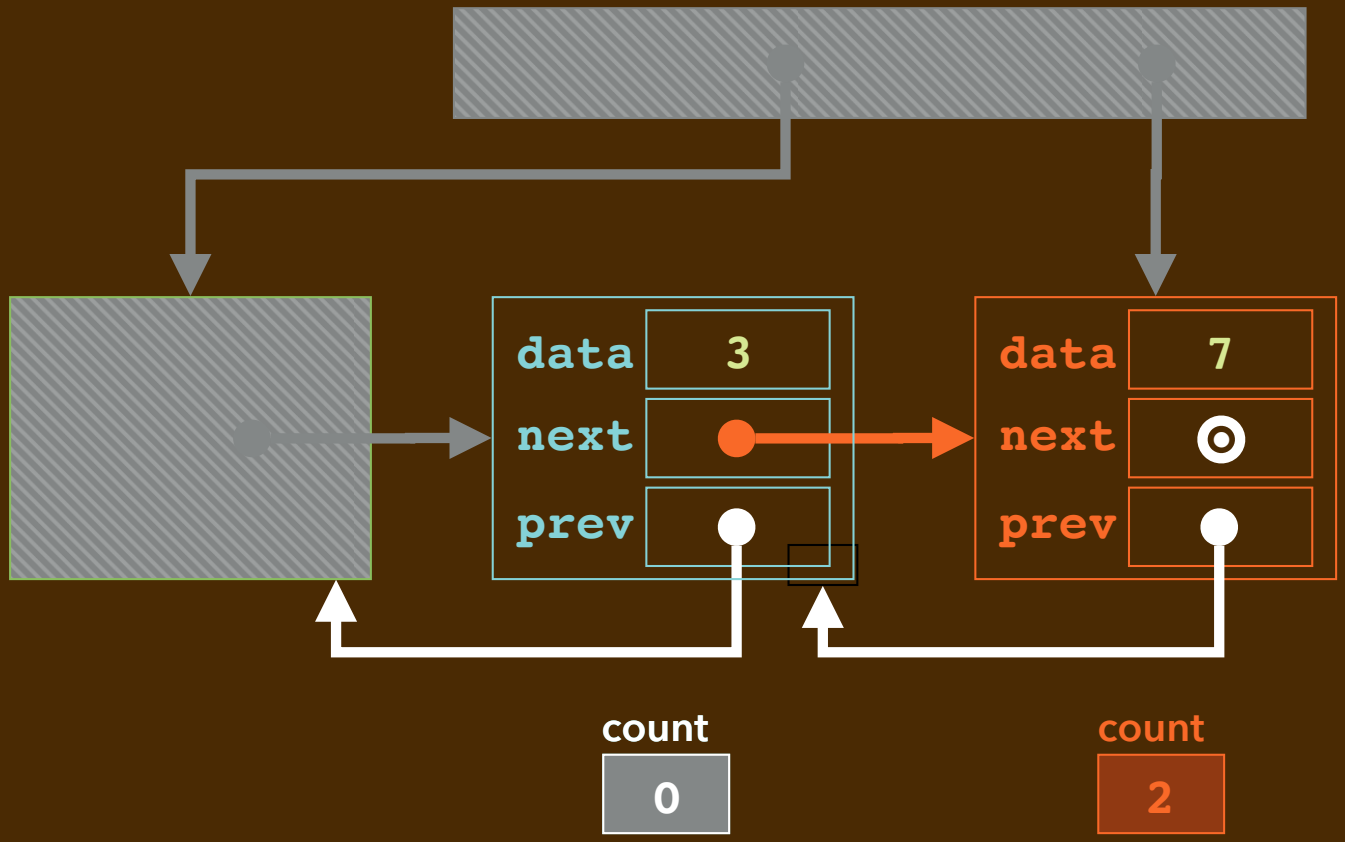
FIX #2: PREV POINTERS THAT DON'T COUNT

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



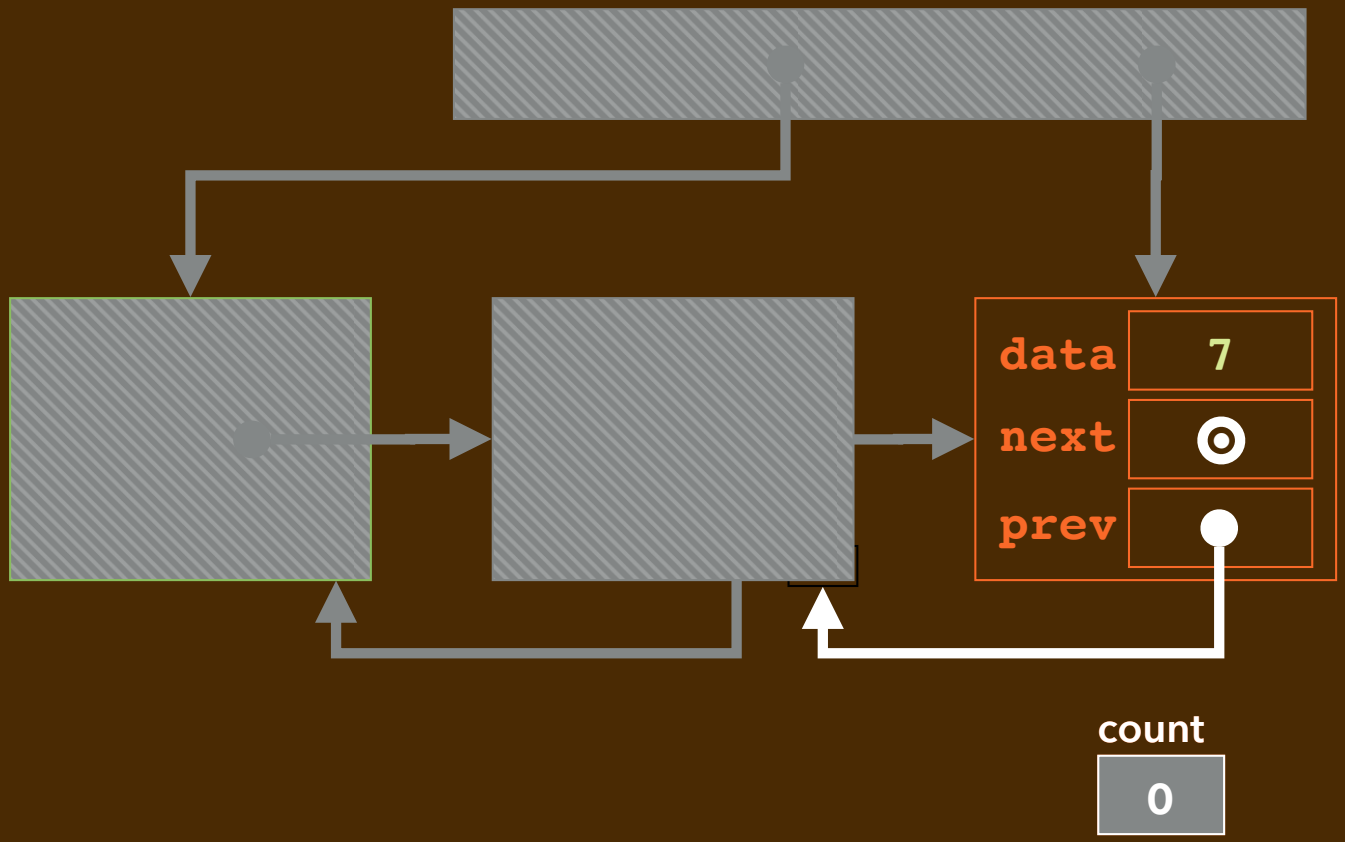
FIX #2: PREV POINTERS THAT DON'T COUNT

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



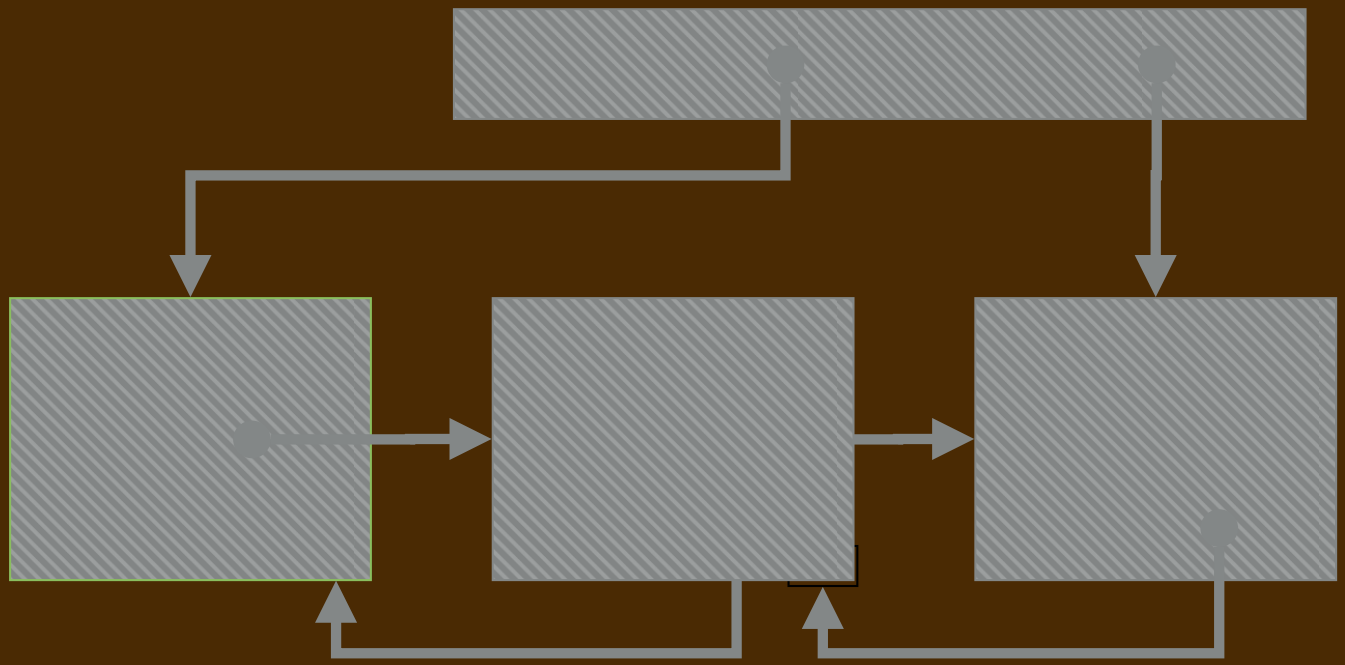
FIX #2: PREV POINTERS THAT DON'T COUNT

```
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 FROM LECTURE 11-3

- I have four versions of linked lists that use **`shared_ptr`** in the **samples** folder of the last lecture:
  - **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

WE'RE DONE!

- This week I'll talk about code that communicates over a network.
  - We'll look at the Berkeley **`socket`** library.
- This 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.
- This is extra material...

WE'RE DONE!

- This week I'll talk about code that communicates over a network.
  - We'll look at the Berkeley **`socket`** library.
- This 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.
- *This 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.
- This is extra material; FYI; "not on the exam."

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

INTRO TO THREADS

LECTURE 12-1 PART 2

JIM FIX, REED COLLEGE CS2-S20

SUPPOSE YOU HAVE TWO TASKS TO COMPLETE

- You write code that performs the one task:
  - `void walk(...) { ... }`
- You write code that performs the other task:
  - `void chewGum(...) { ... }`
- You write the code that actually executes both tasks:

```
int main(void) {
  walk(...);
  chewGum(...);
}
```

- But what if they can be executed independently?
- And what if you had two *processor cores* to execute them?

CO-SCHEDULED PROCESSES

BTW: You *could* write them as two programs instead

- You write **walk.cc**:

```
int main(void) {
  walk(...);
}
```

- You write **chewGum.cc**:

```
int main(void) {
  chewGum(...);
}
```

In a Unix-based console, can execute both by running in the background:

```
> ./walk &
> ./chewGum
```

A WEB BROWSER PROGRAM HAS SEVERAL TASKS IT'S EXECUTING

- On one open tab:
  - It's displaying an article.
  - It's also fetching and displaying animated ads...
- In another open tab:
  - It's streaming a latest episode of a show, fetching that video from a site.
- And in another tab:
  - You are committing your code to GitHub, uploading changes...
- Etc. Etc.

But of course you don't actually run several web browser programs to do that.

- ✦ How is the browser's code written to do that?

A WEB SERVICE IS PERFORMING MULTIPLE TASKS: GITHUB

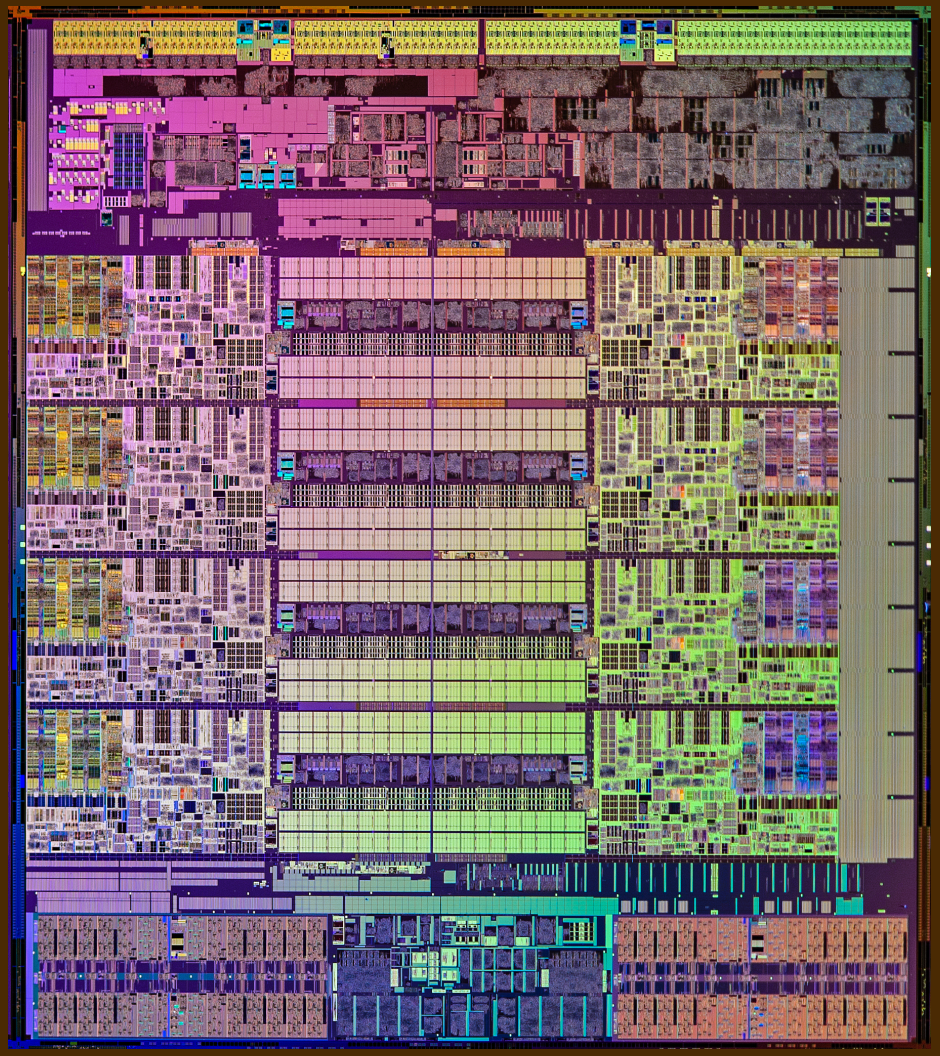
- With one GitHub user's connection:
  - It's uploading some source code changes that are being committed.
- While another GitHub user...
  - is requesting the repo contents of their instructor's slide deck for Lec 12-1.
- While another GitHub user is logging in...
  - and GitHub is checking their credentials and list of repos they can access.
- Etc. Etc.

And 1000s of other users are doing this simultaneously?

- ✦ Does the web service host computer handle these one at a time??
- ✦ No! It can handle them essentially simultaneously; *concurrently*.

TODAY'S COMPUTERS HAVE MULTIPLE CORES

- A chip holds several processing cores on one die:



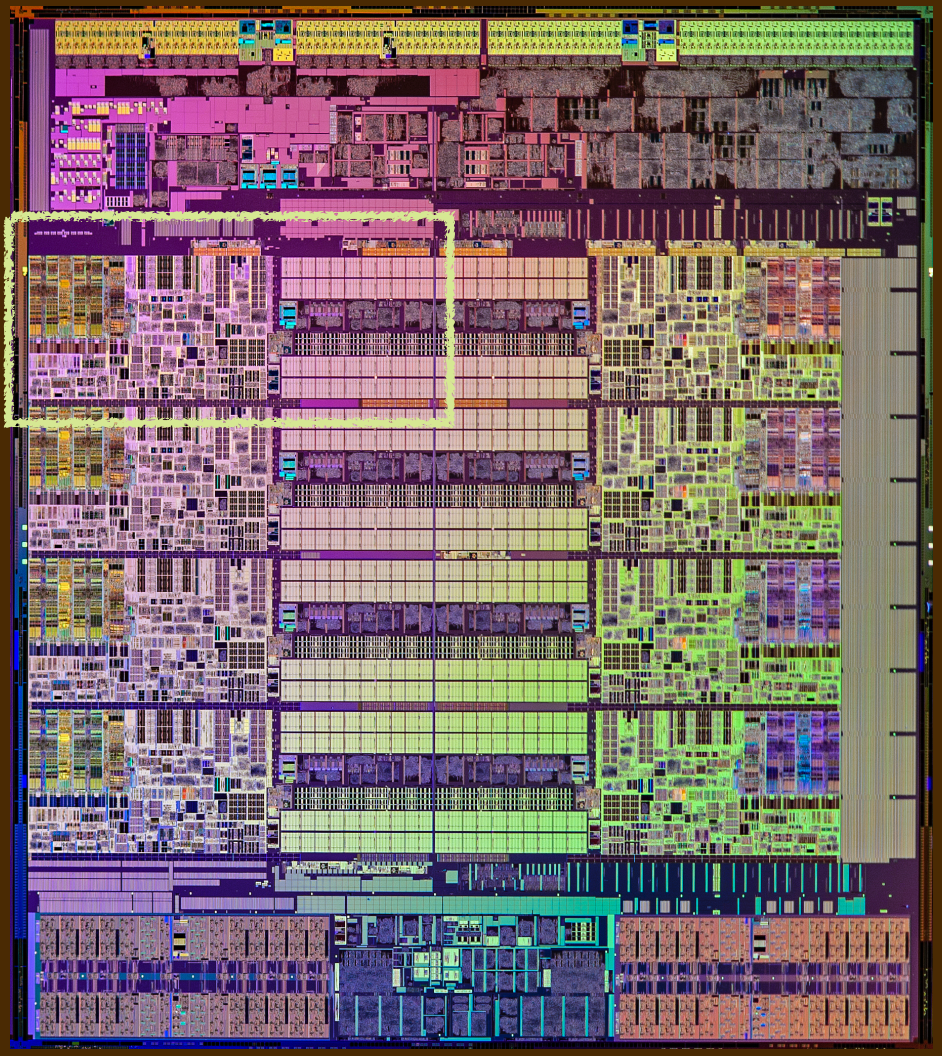
Intel Core i7-5960X Haswell-E 8-Core (2014)

- *https://www.anandtech.com/show/8426/the-intel-haswell-e-cpu-review-core-i7-5960x-i7-5930k-i7-5820k-tested*

TODAY'S COMPUTERS HAVE MULTIPLE CORES

- A chip holds several processing cores on one die:

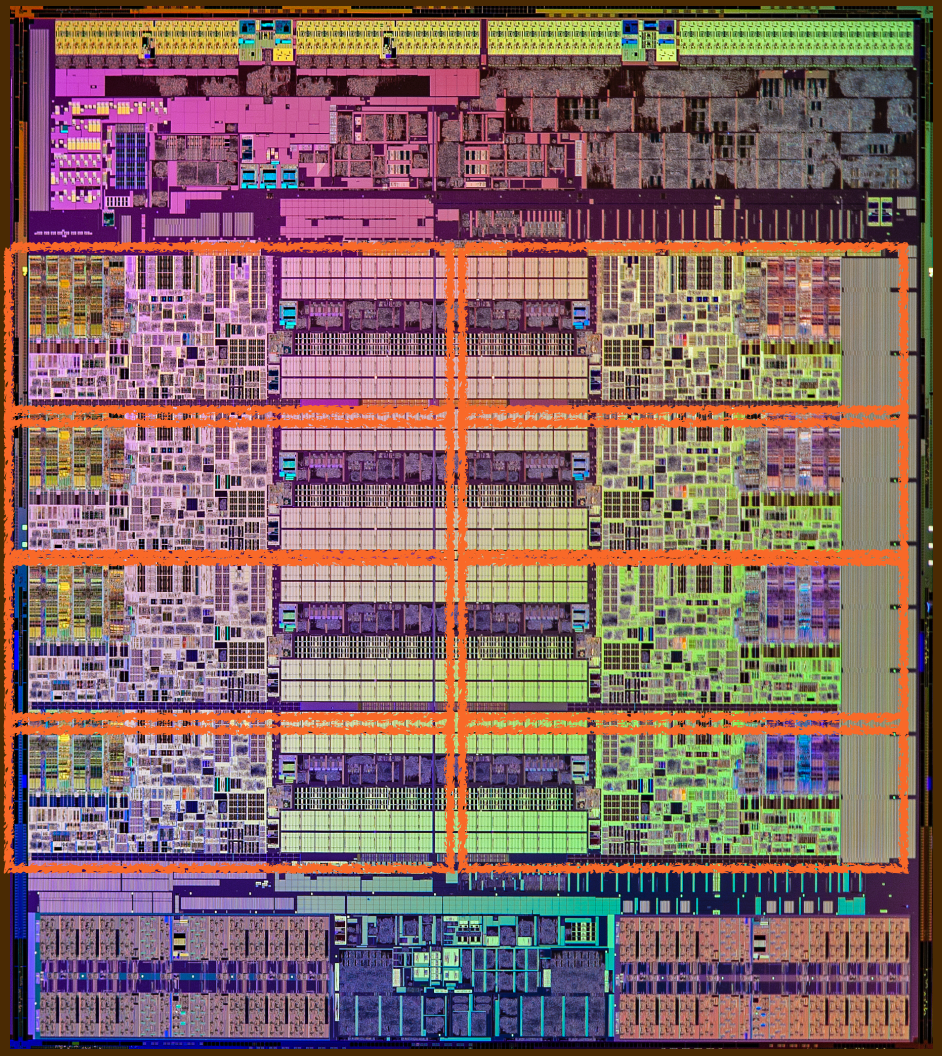
one CPU core



TODAY'S COMPUTERS HAVE MULTIPLE CORES

- A chip holds several processing cores on one die:

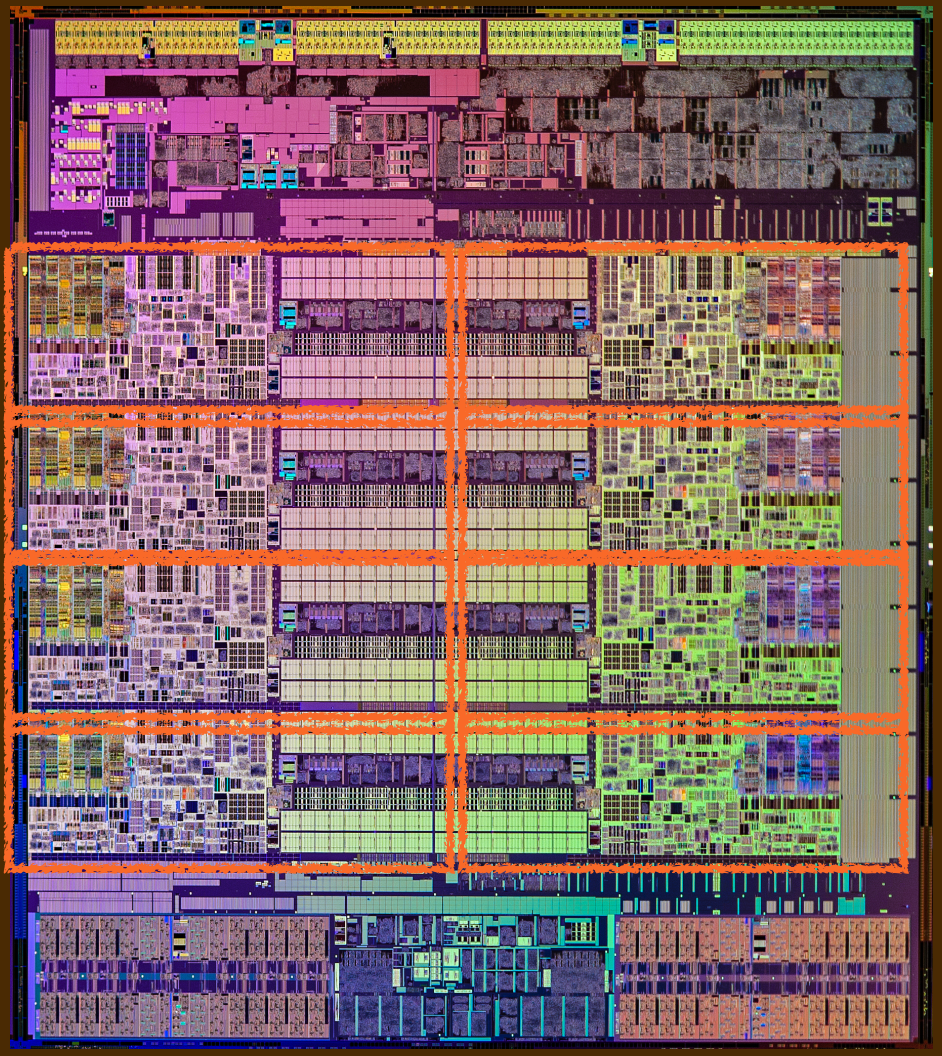
eight CPU cores



TODAY'S COMPUTERS HAVE MULTIPLE CORES

- A chip holds several processing cores on one die:

eight CPU cores

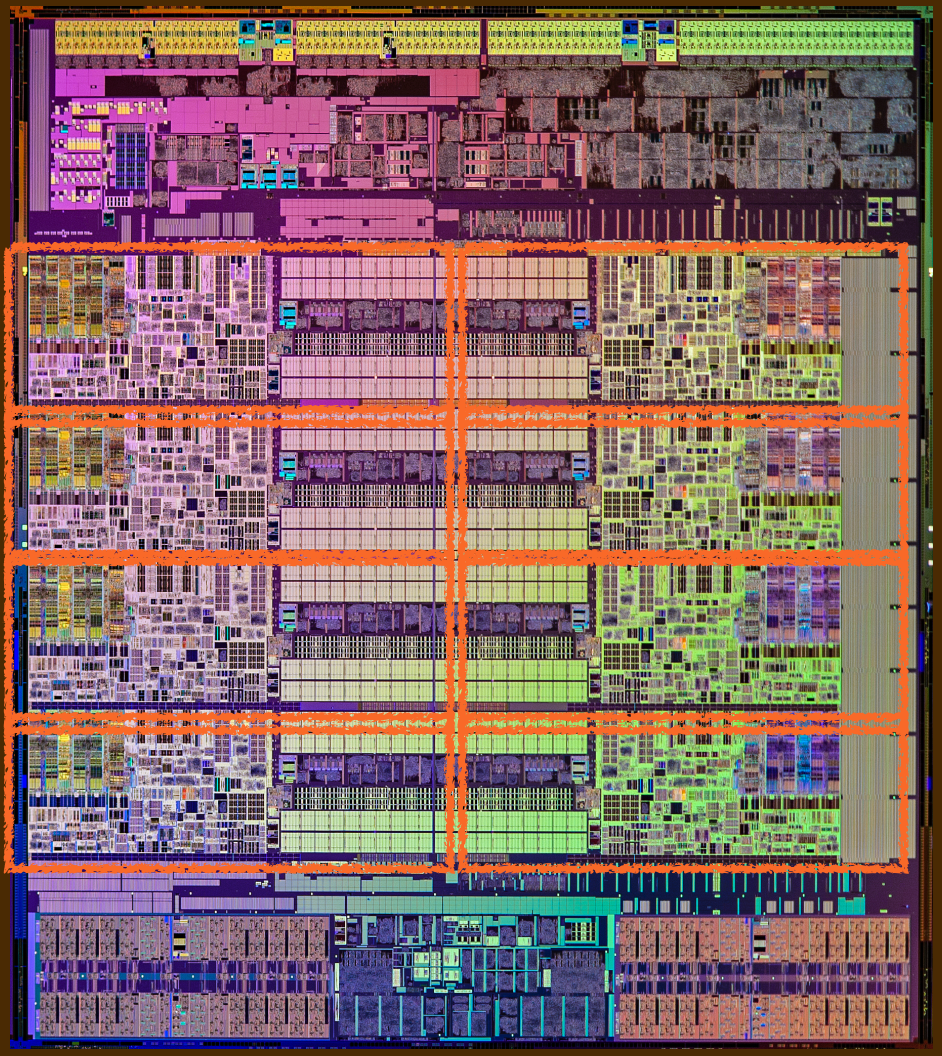


They share access to the same RAM memory, but not on this chip.

TODAY'S COMPUTERS HAVE MULTIPLE CORES

- A chip holds several processing cores on one die:

eight CPU cores



They each can be "fed" their own instruction stream.

They share access to the same RAM memory, but not on this chip.

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

- A ***thread*** is a code executing entity:
  - It has its own instruction pointer into the code.
  - It has its own call stack.
  - It has its own copy of local variables.

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

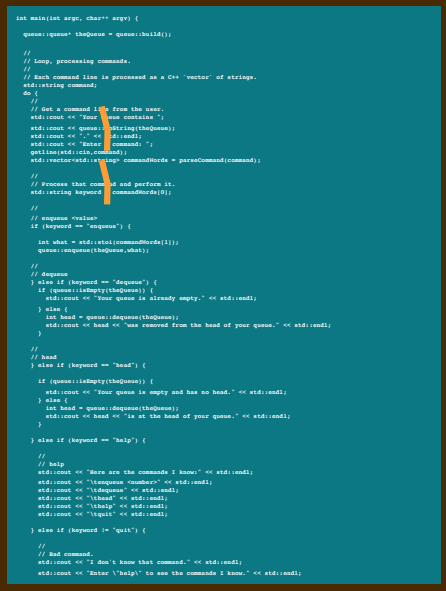
```
int main(int argc, char** argv) {

  queue::queue* theQueue = queue::build();

  //
  // Loop, processing commands.
  //
  // Each command line is processed as a C++ `vector` of strings.
  std::string command;
  do {
    //
    // Get a command line from the user.
    std::cout << "Your queue contains ";
    std::cout << queue::toString(theQueue);
    std::cout << "." << std::endl;
    std::cout << "Enter command: ";
    getline(std::cin,command);
    std::vector<std::string> commandWords = parseCommand(command);

    //
    // Process that command and perform it.
    std::string keyword = commandWords[0];

    //
    // enqueue <value>
    if (keyword == "enqueue") {

      int what = std::stoi(commandWords[1]);
      queue::enqueue(theQueue,what);

    //
    // dequeue
    } else if (keyword == "dequeue") {
      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is already empty." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << "was removed from the head of your queue." << std::endl;
      }

    //
    // head
    } else if (keyword == "head") {

      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is empty and has no head." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << "is at the head of your queue." << std::endl;
      }

    } else if (keyword == "help") {

      //
      // help
      std::cout << "Here are the commands I know:" << std::endl;
      std::cout << "\tenqueue <number>" << std::endl;
      std::cout << "\tdequeue" << std::endl;
      std::cout << "\thead" << std::endl;
      std::cout << "\thelp" << std::endl;
      std::cout << "\tquit" << std::endl;

    } else if (keyword != "quit") {

      //
      // Bad command.
      std::cout << "I don't know that command." << std::endl;
      std::cout << "Enter \"help\" to see the commands I know." << std::endl;
```

- A ***thread*** is a code executing entity:
  - It has its own instruction pointer into the code.
  - It has its own call stack.
  - It has its own copy of local variables.
  - *It "threads" its way through the code, executing it line-by-line.*

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc



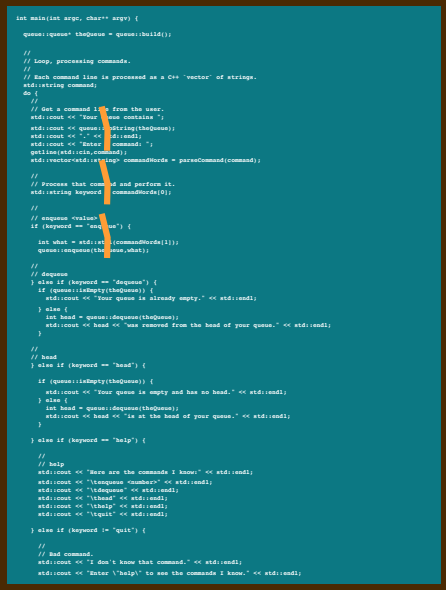
```
int main(int argc, char** argv) {

  queue::queue* theQueue = queue::build();

  //
  // Loop, processing commands.
  //
  // Each command line is processed as a C++ `vector` of strings.
  std::string command;
  do {
    //
    // Get a command line from the user.
    std::cout << "Your queue contains ";
    std::cout << queue::toString(theQueue);
    std::cout << "." << std::endl;
    std::cout << "Enter command: ";
    getline(std::cin,command);
    std::vector<std::string> commandWords = parseCommand(command);

    //
    // Process that command and perform it.
    std::string keyword = commandWords[0];

    //
    // enqueue <value>
    if (keyword == "enqueue") {

      int what = std::stoi(commandWords[1]);
      queue::enqueue(theQueue,what);

    //
    // dequeue
    } else if (keyword == "dequeue") {
      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is already empty." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << "was removed from the head of your queue." << std::endl;
      }

    //
    // head
    } else if (keyword == "head") {

      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is empty and has no head." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << "is at the head of your queue." << std::endl;
      }

    } else if (keyword == "help") {

      //
      // help
      std::cout << "Here are the commands I know:" << std::endl;
      std::cout << "\tenqueue <number>" << std::endl;
      std::cout << "\tdequeue" << std::endl;
      std::cout << "\thead" << std::endl;
      std::cout << "\thelp" << std::endl;
      std::cout << "\tquit" << std::endl;

    } else if (keyword != "quit") {

      //
      // Bad command.
      std::cout << "I don't know that command." << std::endl;
      std::cout << "Enter \"help\" to see the commands I know." << std::endl;
```

- A ***thread*** is a code executing entity:
  - It has its own instruction pointer into the code.
  - It has its own call stack.
  - It has its own copy of local variables.
  - *It "threads" its way through the code, executing it line-by-line.*

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc



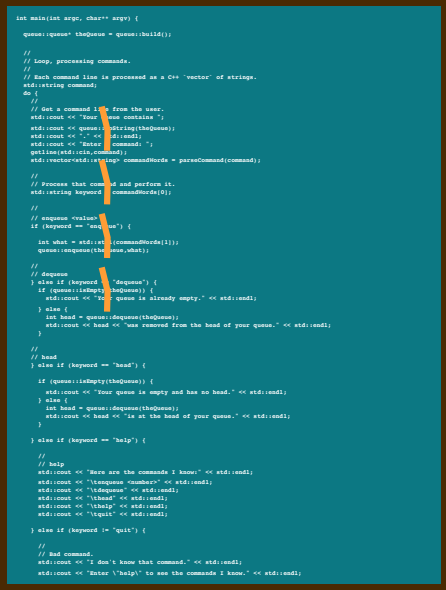
```
int main(int argc, char** argv) {

  queue::queue* theQueue = queue::build();

  //
  // Loop, processing commands.
  //
  // Each command line is processed as a C++ `vector` of strings.
  std::string command;
  do {
    //
    // Get a command line from the user.
    std::cout << "Your queue contains ";
    std::cout << queue::toString(theQueue);
    std::cout << "." << std::endl;
    std::cout << "Enter a command: ";
    getline(std::cin,command);
    std::vector<std::string> commandWords = parseCommand(command);

    //
    // Process that command and perform it.
    std::string keyword = commandWords[0];

    //
    // enqueue <value>
    if (keyword == "enqueue") {

      int what = std::stoi(commandWords[1]);
      queue::enqueue(theQueue,what);

    //
    // dequeue
    } else if (keyword == "dequeue") {
      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is already empty." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << " was removed from the head of your queue." << std::endl;
      }

    //
    // head
    } else if (keyword == "head") {

      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is empty and has no head." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << " is at the head of your queue." << std::endl;
      }

    } else if (keyword == "help") {

      //
      // help
      std::cout << "Here are the commands I know:" << std::endl;
      std::cout << "\tenqueue <number>" << std::endl;
      std::cout << "\tdequeue" << std::endl;
      std::cout << "\thead" << std::endl;
      std::cout << "\thelp" << std::endl;
      std::cout << "\tquit" << std::endl;

    } else if (keyword != "quit") {

      //
      // Bad command.
      std::cout << "I don't know that command." << std::endl;
      std::cout << "Enter \"help\" to see the commands I know." << std::endl;
```

- A ***thread*** is a code executing entity:
  - It has its own instruction pointer into the code.
  - It has its own call stack.
  - It has its own copy of local variables.
  - *It "threads" its way through the code, executing it line-by-line.*

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc



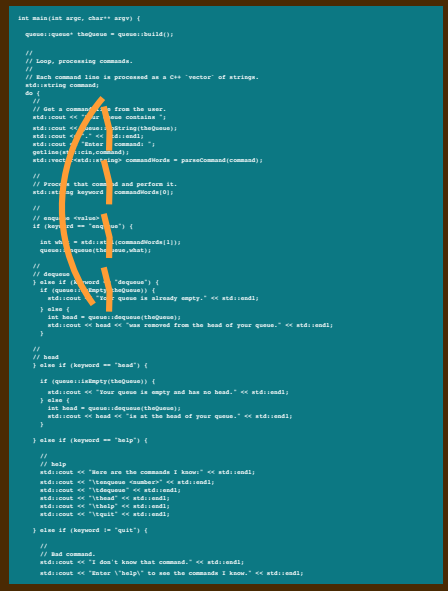
```
int main(int argc, char** argv) {

  queue::queue* theQueue = queue::build();

  //
  // Loop, processing commands.
  //
  // Each command line is processed as a C++ `vector` of strings.
  std::string command;
  do {
    //
    // Get a command line from the user.
    std::cout << "Your queue contains ";
    std::cout << queue::toString(theQueue);
    std::cout << "." << std::endl;
    std::cout << "Enter command: ";
    getline(std::cin,command);
    std::vector<std::string> commandWords = parseCommand(command);

    //
    // Process that command and perform it.
    std::string keyword = commandWords[0];

    //
    // enqueue <value>
    if (keyword == "enqueue") {

      int what = std::stoi(commandWords[1]);
      queue::enqueue(theQueue,what);

    //
    // dequeue
    } else if (keyword == "dequeue") {
      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is already empty." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << " was removed from the head of your queue." << std::endl;
      }

    //
    // head
    } else if (keyword == "head") {

      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is empty and has no head." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << " is at the head of your queue." << std::endl;
      }

    } else if (keyword == "help") {

      //
      // help
      std::cout << "Here are the commands I know:" << std::endl;
      std::cout << "\tenqueue <number>" << std::endl;
      std::cout << "\tdequeue" << std::endl;
      std::cout << "\thead" << std::endl;
      std::cout << "\thelp" << std::endl;
      std::cout << "\tquit" << std::endl;

    } else if (keyword != "quit") {

      //
      // Bad command.
      std::cout << "I don't know that command." << std::endl;
      std::cout << "Enter \"help\" to see the commands I know." << std::endl;
```

- A ***thread*** is a code executing entity:
  - It has its own instruction pointer into the code.
  - It has its own call stack.
  - It has its own copy of local variables.
  - *It "threads" its way through the code, executing it line-by-line.*

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

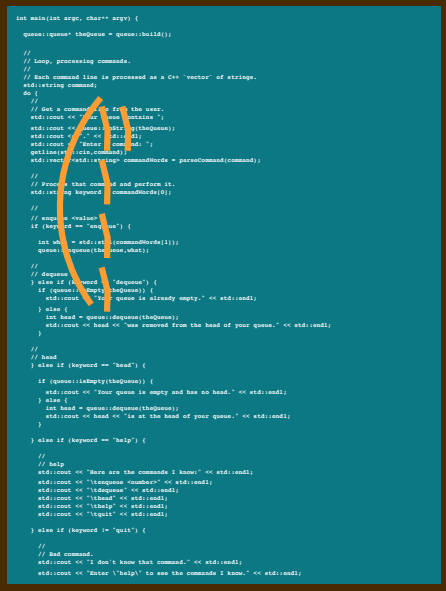


- A ***thread*** is a code executing entity:
  - It has its own instruction pointer into the code.
  - It has its own call stack.
  - It has its own copy of local variables.
  - *It "threads" its way through the code, executing it line-by-line.*

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

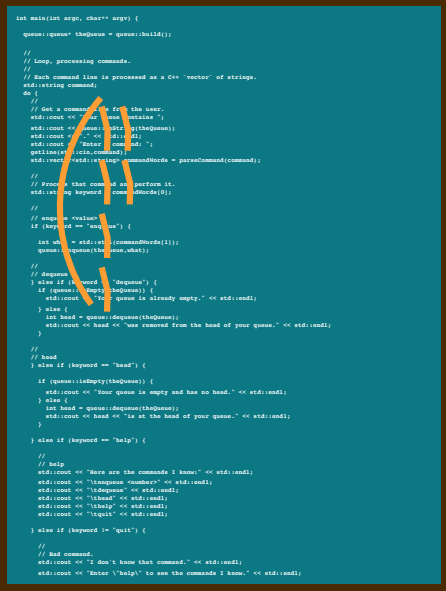


- A ***thread*** is a code executing entity:
  - It has its own instruction pointer into the code.
  - It has its own call stack.
  - It has its own copy of local variables.
  - *It "threads" its way through the code, executing it line-by-line.*

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

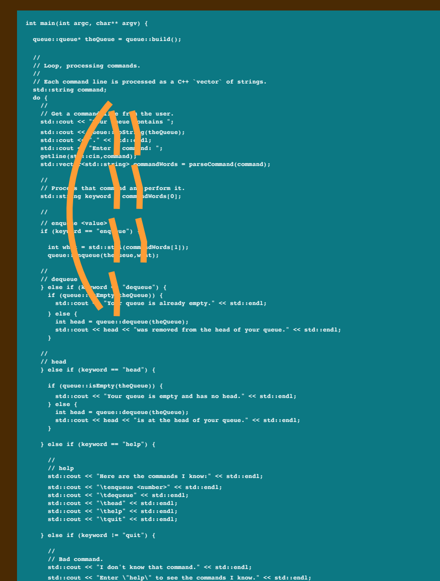


- A ***thread*** is a code executing entity:
  - It has its own instruction pointer into the code.
  - It has its own call stack.
  - It has its own copy of local variables.
  - *It "threads" its way through the code, executing it line-by-line.*

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

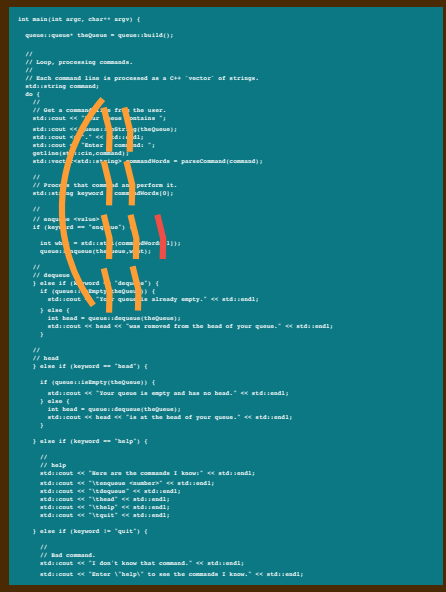


- A ***thread*** is a code executing entity:
  - The libraries allow us to *create another thread* of execution.
  -

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

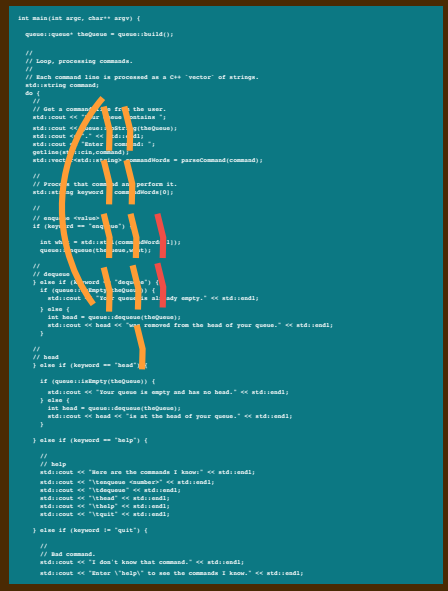


- A ***thread*** is a code executing entity:
  - The libraries allow us to *create another thread* of execution.
  - It also *threads its way through the code line by line*, independently of the *other thread*.

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

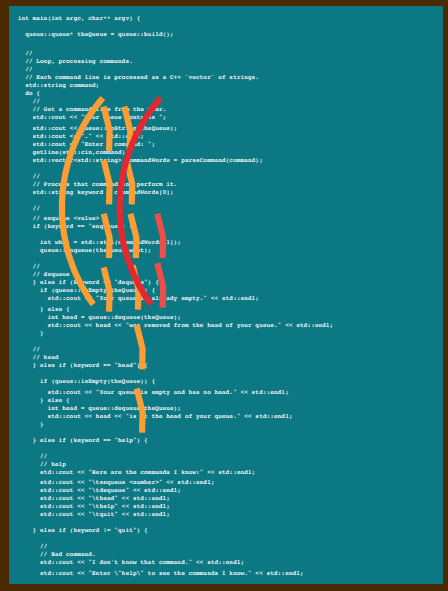


- A ***thread*** is a code executing entity:
  - The libraries allow us to *create another thread* of execution.
  - It also *threads its way through the code line by line*, independently of the *other thread*.

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

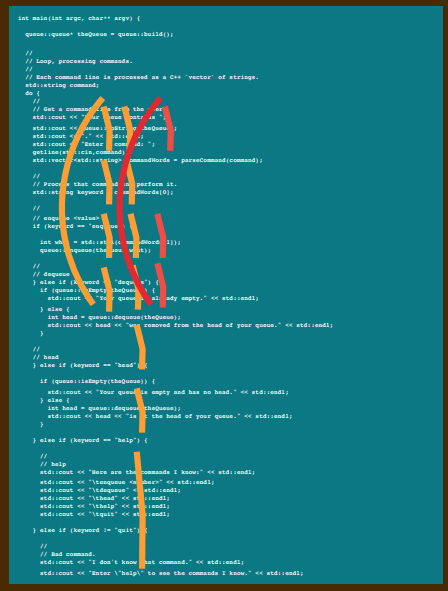


- A ***thread*** is a code executing entity:
  - The libraries allow us to *create another thread* of execution.
  - It also *threads its way through the code line by line*, independently of the *other thread*.

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

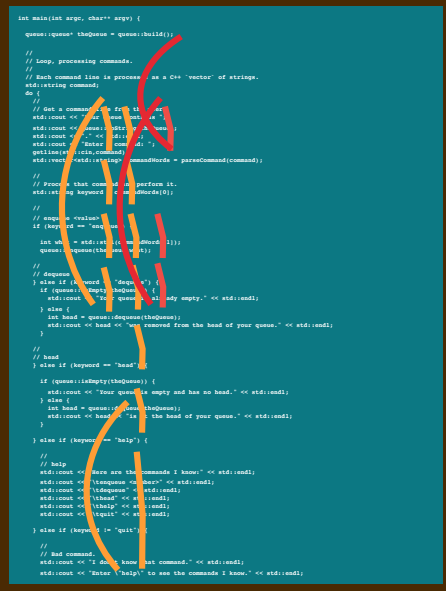


- A ***thread*** is a code executing entity:
  - The libraries allow us to *create another thread* of execution.
  - It also *threads its way through the code line by line*, independently of the *other thread*.

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

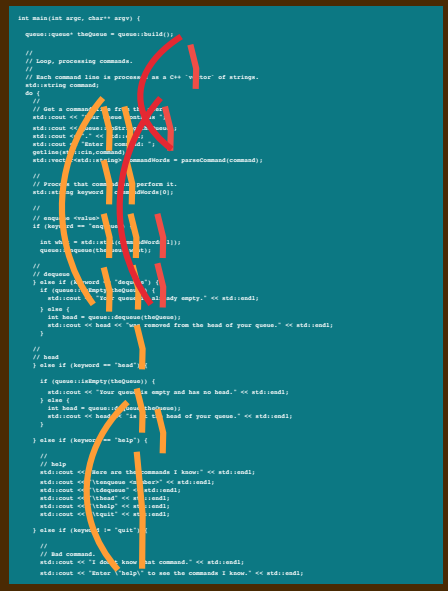


- A ***thread*** is a code executing entity:
  - The libraries allow us to *create another thread* of execution.
  - It also *threads its way through the code line by line*, independently of the *other thread*.

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

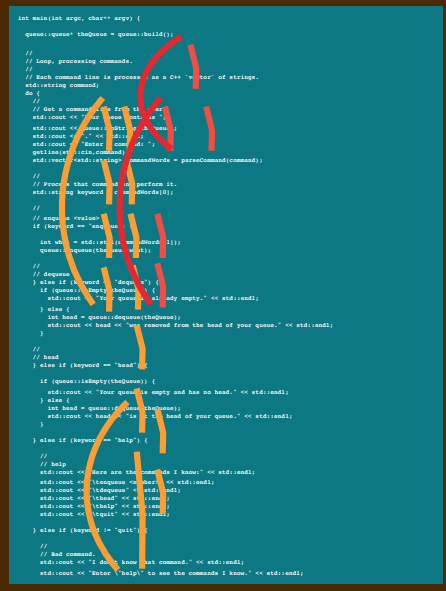


- A ***thread*** is a code executing entity:
  - The libraries allow us to *create another thread* of execution.
  - It also *threads its way through the code line by line*, independently of the *other thread*.

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc



- A ***thread*** is a code executing entity:
  - The libraries allow us to *create another thread* of execution.
  - It also *threads its way through the code line by line*, independently of the *other thread*.

CREATING TWO THREADS TO WALK AND CHEWGUM

- The **samples** folder contains C++ source **walkchew.cc** to perform two tasks.
- It uses the *POSIX Threads library* to create two threads:
  - One repeatedly outputs `step` #*xxxxxx*
  - One repeatedly outputs `chomp` #*yyyyyy*
    - ➡ Each with a count *xxxxxx* or *yyyyyy* of what step it's executing.
- The two threads run *concurrently*.
- The OS might run them *in parallel* on two separate cores.

CREATING TWO THREADS TO WALK AND CHEWGUM

```
#include <pthread.h> ...
typedef void *tmain(void *);
void *walk(long *v) {
  (*v) = 0;
  for (long i=100000;i>0;i--,(*v)++) std::cout<<"step #"<< *v <<"\n";
  return NULL;
}
void *chewGum(long *v) {
  long i;
  for (long i=100000;i>0;i--,(*v)++) std::cout<<"chomp #"<< *v <<"\n";
  return NULL;
}
int main(int argc, char **argv) {
  pthread_t id1, id2;
  long a1, a2, r1, r2;
  pthread_create(&id1,NULL,(tmain *)walk,   (void *)&a1);
  pthread_create(&id2,NULL,(tmain *)chewGum,(void *)&a2);
  pthread_join(id1,(void **)&r1);
  pthread_join(id2,(void **)&r2);
  std::cout << "Completed steps:" << a1 << " ...chomps:" << a2 << "\n";
  pthread_exit(NULL);
}
```

CREATING TWO THREADS COLLABORATIVELY COUNT

- The **samples** folder contains C++ source **count.cc**.
- It also uses the *PThreads library* to create two threads:
  - One repeatedly increments a value in memory.
  - The other repeatedly increments *the same location* in memory.
- These two collaborating threads also run *concurrently*.

TWO THREADS COLLABORATE ON COUNTING...

```
#include <pthread.h> ...
typedef void *tmain(void *);
long increment(long *v) {
 long i;
 for (i=0; i<100000; i++) {
    long tmp = (*v);
    (*v) = tmp + 1;
  }
  return i;
}
int main(int argc, char **argv) {
  pthread_t id1, id2;
  long r1, r2;
  long count = 0;
  pthread_create(&id1,NULL,(tmain *)increment,(void *)&count);
  pthread_create(&id2,NULL,(tmain *)increment,(void *)&count);
  pthread_join(id1,(void **)&r1);
  pthread_join(id2,(void **)&r2);
  std::cout << "Thread 1 incremented " << r1 << " times." << std::endl;
  std::cout << "Thread 2 incremented " << r2 << " times." << std::endl;
  std::cout << "The final count is " << count << "." << std::endl;
  pthread_exit(NULL);
}
```

WHAT HAPPENS?

```
> g++ -o count -lpthread count.cc
> ./count
```

????

WHAT HAPPENS? TUNE IN WEDNESDAY...

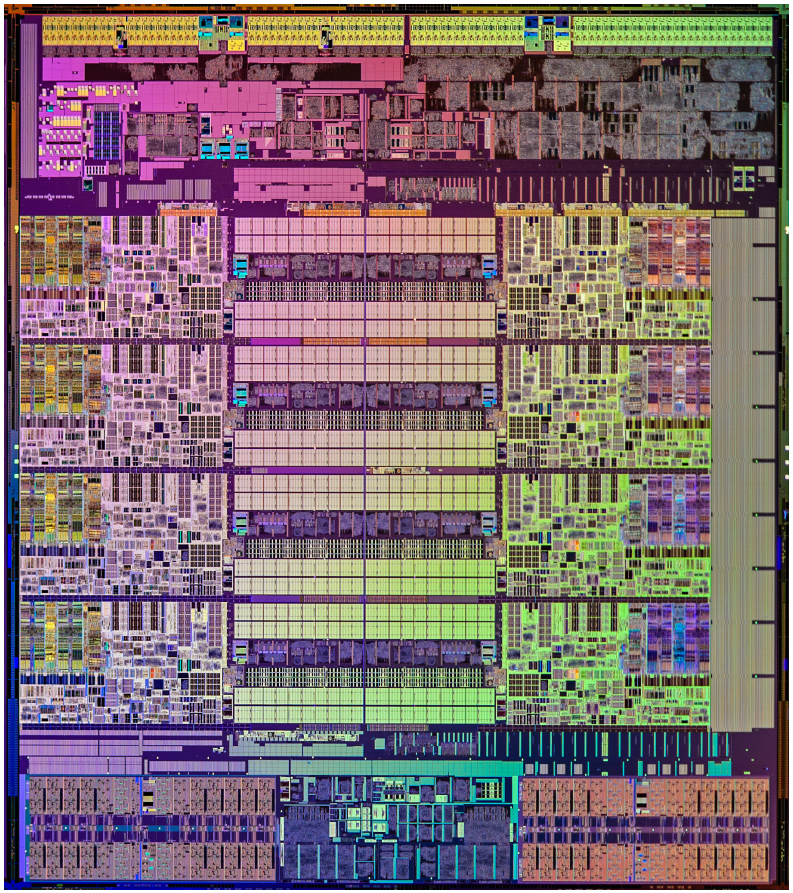
```
> g++ -o count -lpthread count.cc
> ./count
```

????

My Laptop

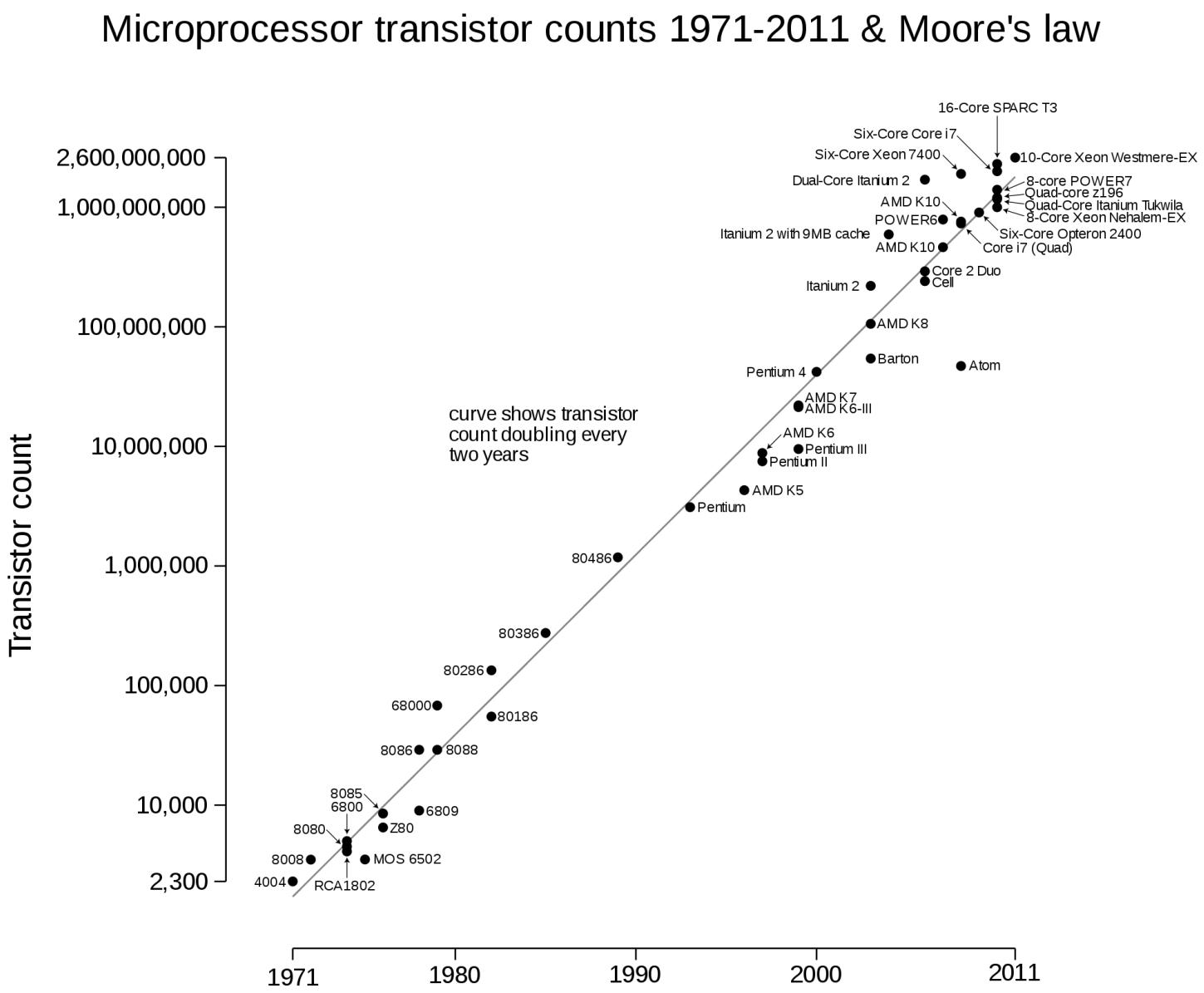
From 2013, based on an Intel Core i7

- runs OSX 10.11.6, based on Mach OS
- 16 GB of memory, 2.8 GHz clock
- 64-bit architecture, 64-bit addresses
- 1.3 billion transistors
- 181 mm^2
- 22 nm feature size
- 2 cores
- Picture: similar family, 8 cores



Computer Performance History

Processor technology has skyrocketed in those 40 years



Slight Digression: Parallelism

My history (cont'd): parallel computers in the late 80s

- BBN Butterfly 64-node computer (Livermore)
- MasPar MP-2 with 16384 4-bit processors—let's think of this as 2048 32-bit processors

These were kitchen appliance-sized machines and cost $1M+.

Today's computers have several processors on a chip

- not unusual to buy computer with 4-core chip; there are 16-64 core chips available for ~4-16x the price of a consumer processor
- graphics processors (GPUs) have 500-2000 "streaming" processors to compute the display of 3D & video

So there are 80s supercomputers on a single chip, and under $15K!

Slight Digression: Parallelism

My history (cont'd): parallel computers in the late 80s

- BBN Butterfly 64-node computer (Livermore)
- MasPar MP-2 with 16384 4-bit processors—let's think of this as 2048 32-bit processors

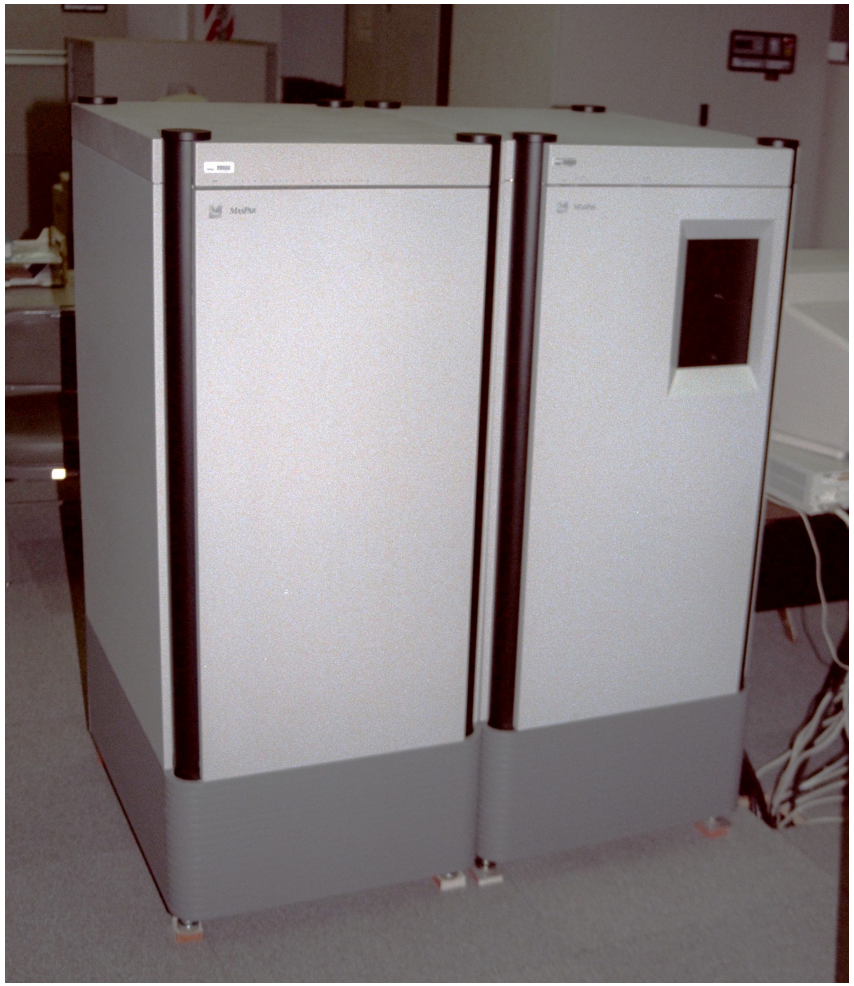
These were kitchen appliance-sized machines and cost $1M+.

Today's computers have several processors on a chip

- not unusual to buy computer with 4-core chip; there are 16-64 core chips available for ~4-16x the price of a consumer processor
- graphics processors (GPUs) have 500-2000 "streaming" processors to compute the display of 3D & video

So there are 80s supercomputers on a single chip, and under $15K!

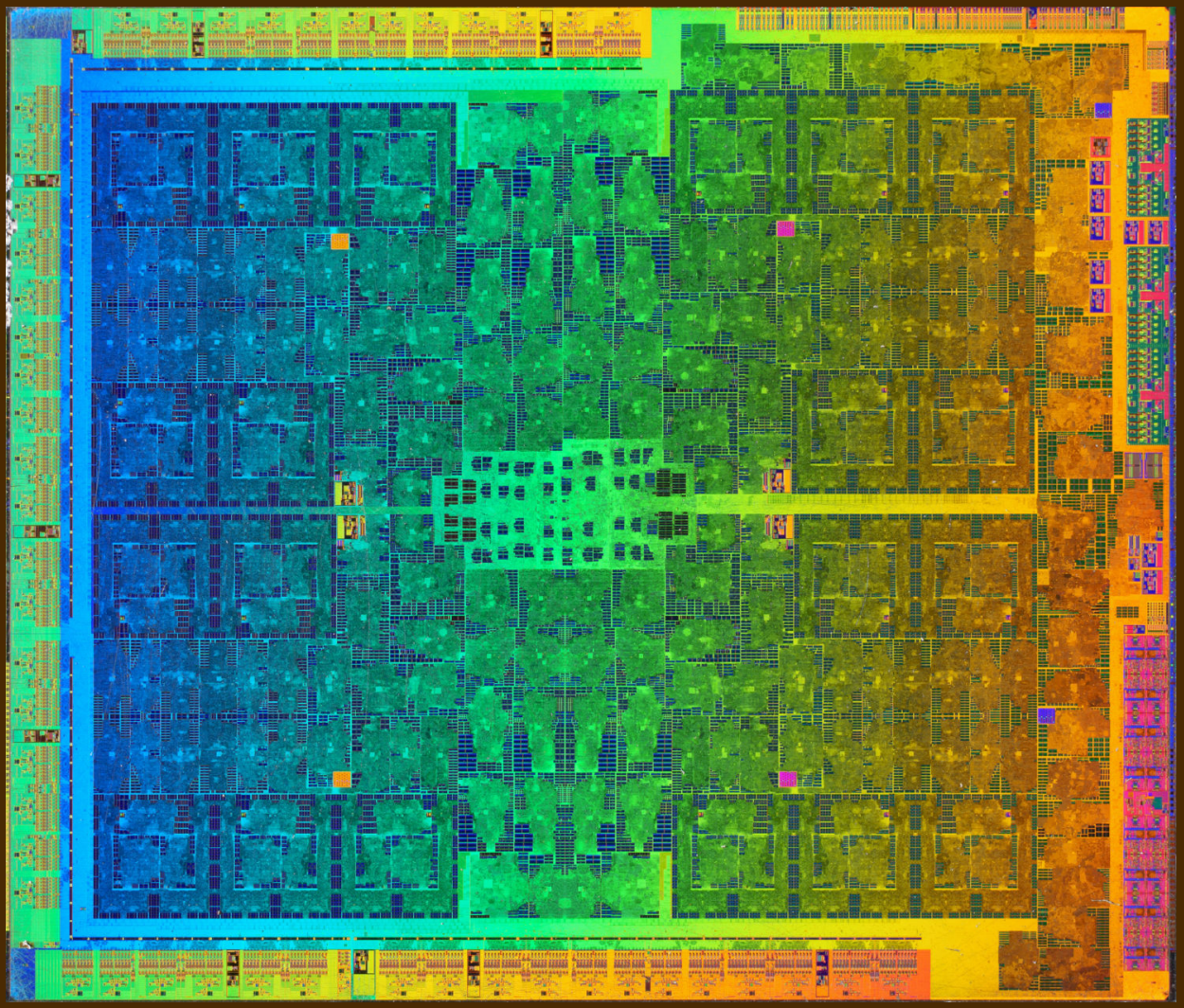
Slight Digression: Parallelism



- MasPar MP-2 with 16384 4-bit processors
  - ➡Let's think of this as 2048 32-bit processors

TODAY'S COMPUTERS HAVE A SEPARATE GPU FOR DISPLAY

- Graphics processors (GPUs) have 100s of "streaming" processors:



NVidia
GTX 1080 GP 104
with its 2560 "cores"

- *https://wccftech.com/nvidia-gtx-1080-gp104-die-shot/*