Linked Lists

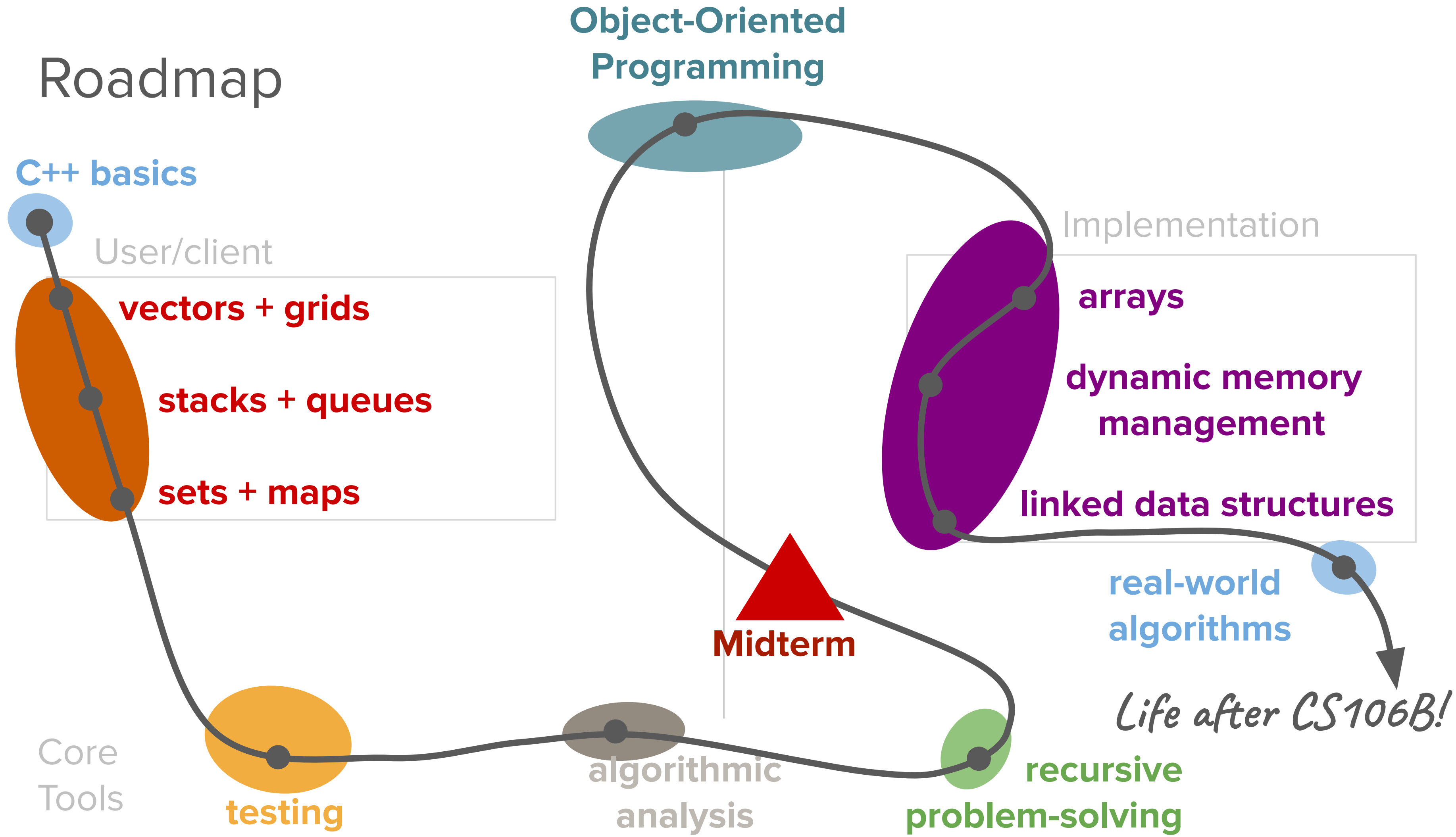
What's something annoying/frustrating about arrays?

Feel free to mention something that's come up in A4
or something that's confusing from lecture.

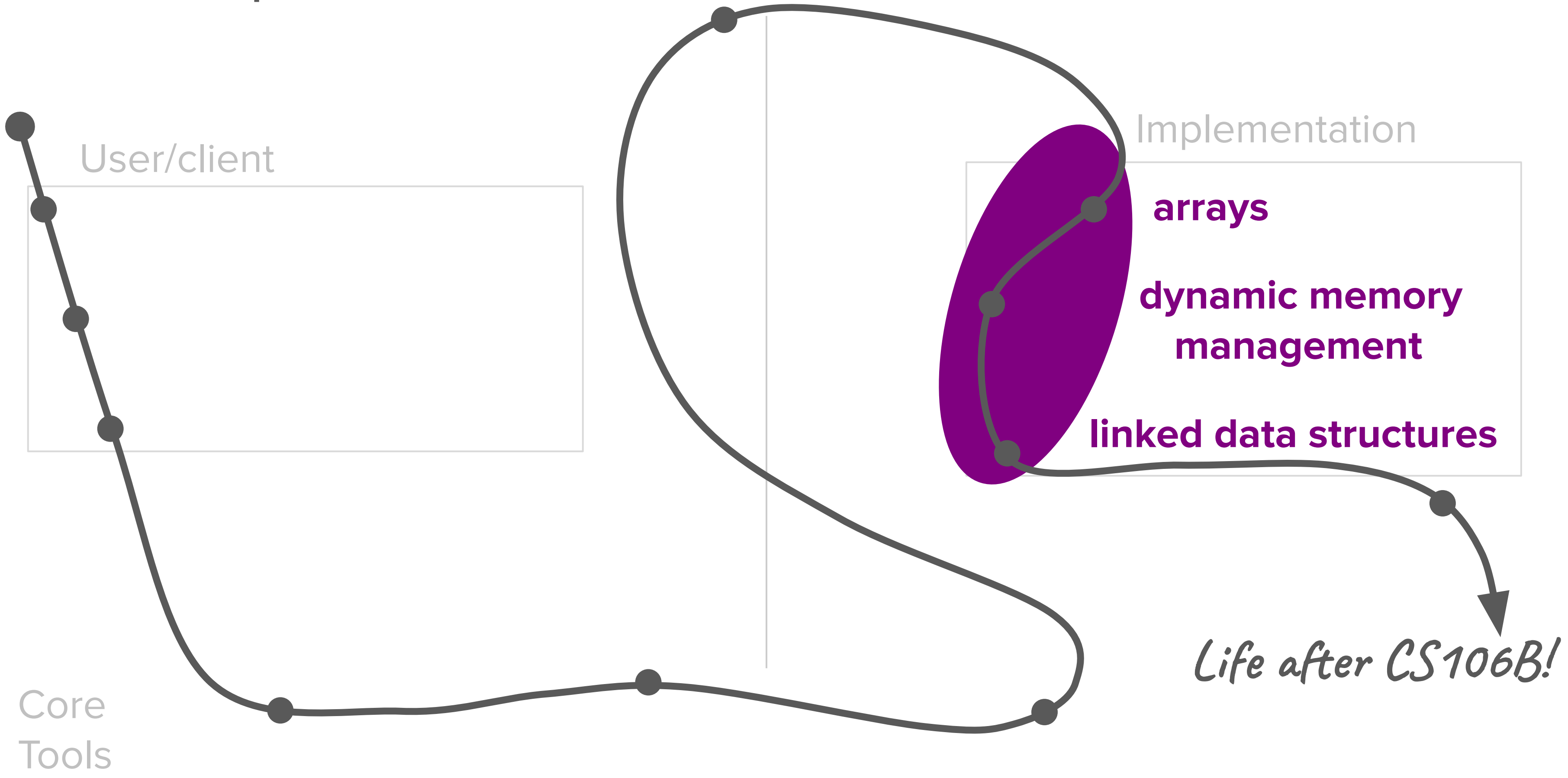
(pollev.com/cs106bpoll)

Something that's annoying/frustrating about working with arrays?

Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Midterm
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Roadmap


User/client
Implementation
arrays
dynamic memory management
linked data structures
Core Tools
Life after CS106B!

Today’s question

How can we use pointers to organize non-contiguous memory on the heap?

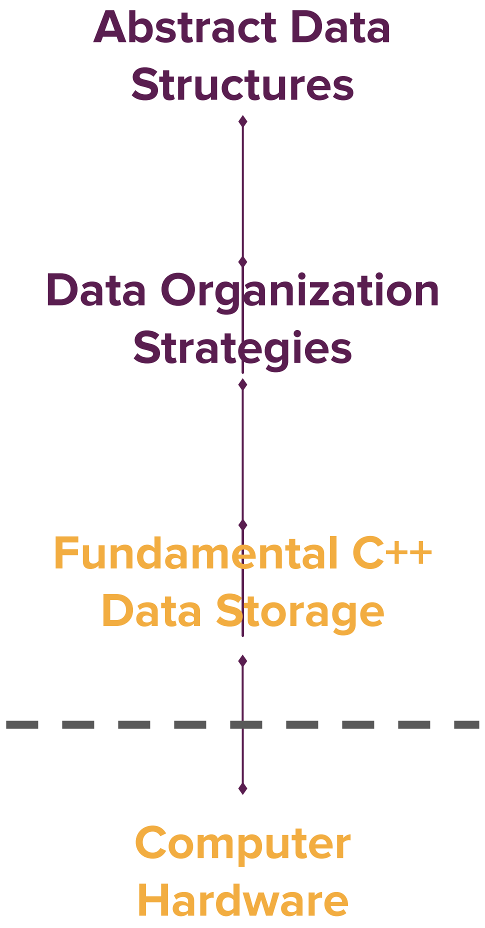
Today's topics

1. Review
2. What is a linked list?
3. How do we use linked lists in a class?
4. How do we manipulate linked lists?

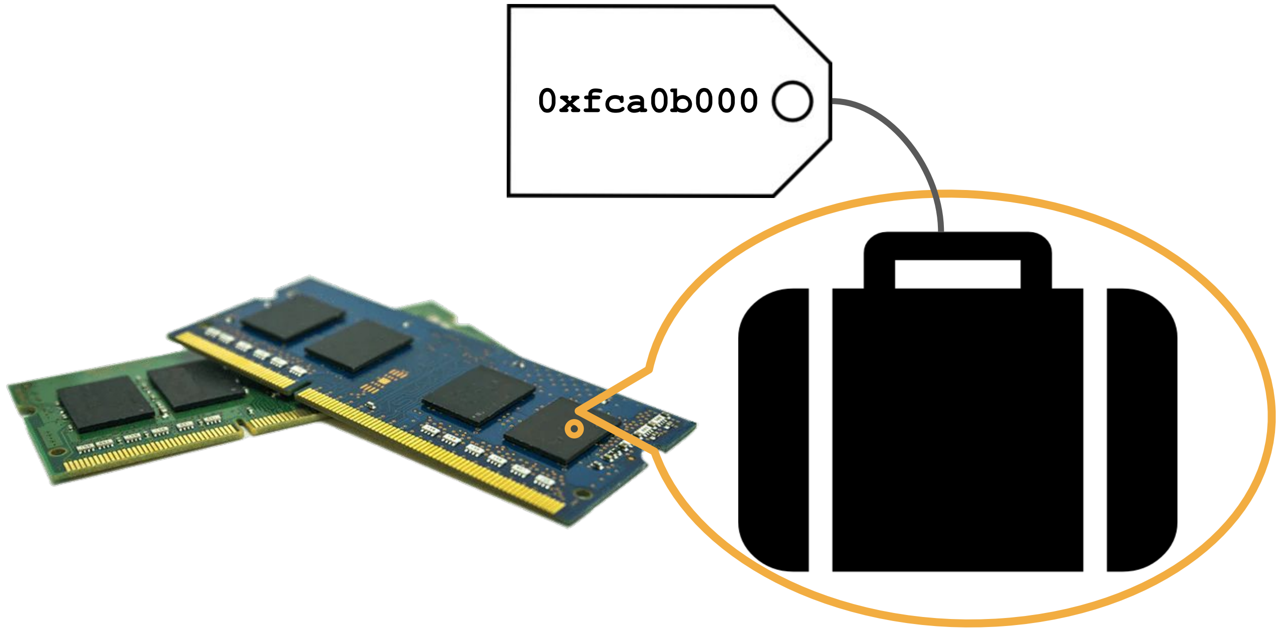
Review

[memory and pointers]

Levels of abstraction



How is computer memory organized?



Pointers and Memory

- Every variable you create has an address in memory on your computer (either on the stack or the heap).

How is computer memory organized?

Stack

Static memory allocation

Automatic memory management

Persistence is out of your control!

You need to know size needed at compile time!

Hard to share a single large object (copy instead)

Heap

Dynamic memory allocation

You manage the memory

You manage persistence!

You can figure out the size needed at runtime!

You can share a single large object between classes (with pointers!).

How is computer memory organized?

Stack

Static memory allocation

Automatic memory management

Persistence is out of your control!

You need to know size needed at compile time!

Hard to share a single large object (copy instead)

Heap

Dynamic memory allocation

You manage the memory

You manage persistence!

You can figure out the size needed at runtime!

You can share a single large object between classes (with pointers!).

Pointers and Memory

- Every variable you create has an address in memory on your computer (either on the stack or the heap).

- A pointer is just a type of variable that stores a memory address!

Pointers and Memory

- Every variable you create has an address in memory on your computer (either on the stack or the heap).

- A pointer is just a type of variable that stores a memory address!
    - You specify the type of the variable that it points to so that C++ knows how much space the value its pointing to is taking up (e.g. **`string*`** or **`int*`** or **`Vector*`**).

Pointers and Memory

- Every variable you create has an address in memory on your computer (either on the stack or the heap).

- A pointer is just a type of variable that stores a memory address!
  - You specify the type of the variable that it points to so that C++ knows how much space the value its pointing to is taking up (e.g. **`string*`** or **`int*`** or **`Vector*`**).
  - But remember that pointers and what they point to (e.g. **`string`** vs. **`string*`**) are two completely different data types!

Pointers and Memory

- Every variable you create has an address in memory on your computer (either on the stack or the heap)

- A pointer is just a type of variable that stores a memory address!

- When you **dynamically allocate** variables on the heap, you must use the keyword **`new`** (or **`new[]`** for arrays) and must store the address in a pointer to keep track of it.
    - E.g. `int* number = new int;`
    - E.g. `int* numArr = new int[5];`

Pointers and Memory

- Every variable you create has an address in memory on your computer (either on the stack or the heap)

- A pointer is just a type of variable that stores a memory address!

- When you **dynamically allocate** variables on the heap, you must use the keyword **`new`** (or **`new[]`** for arrays) and must store the address in a pointer to keep track of it.
    - E.g. `int* number = new int;`
    - E.g. `int* numArr = new int[5];`

Dynamically allocated variables are the only reason we'll use pointers in this class!

Pointers and Memory

- Every variable you create has an address in memory on your computer (either on the stack or the heap)

- A pointer is just a type of variable that stores a memory address!

- When you **dynamically allocate** variables on the heap, you must use the keyword **`new`** (or **`new[]`** for arrays) and must store the address in a pointer to keep track of it.

- To get the value located at the memory address stored in a pointer, you must **dereference** the pointer using the **`*`** operator (e.g. **`cout << *number << endl;`**).

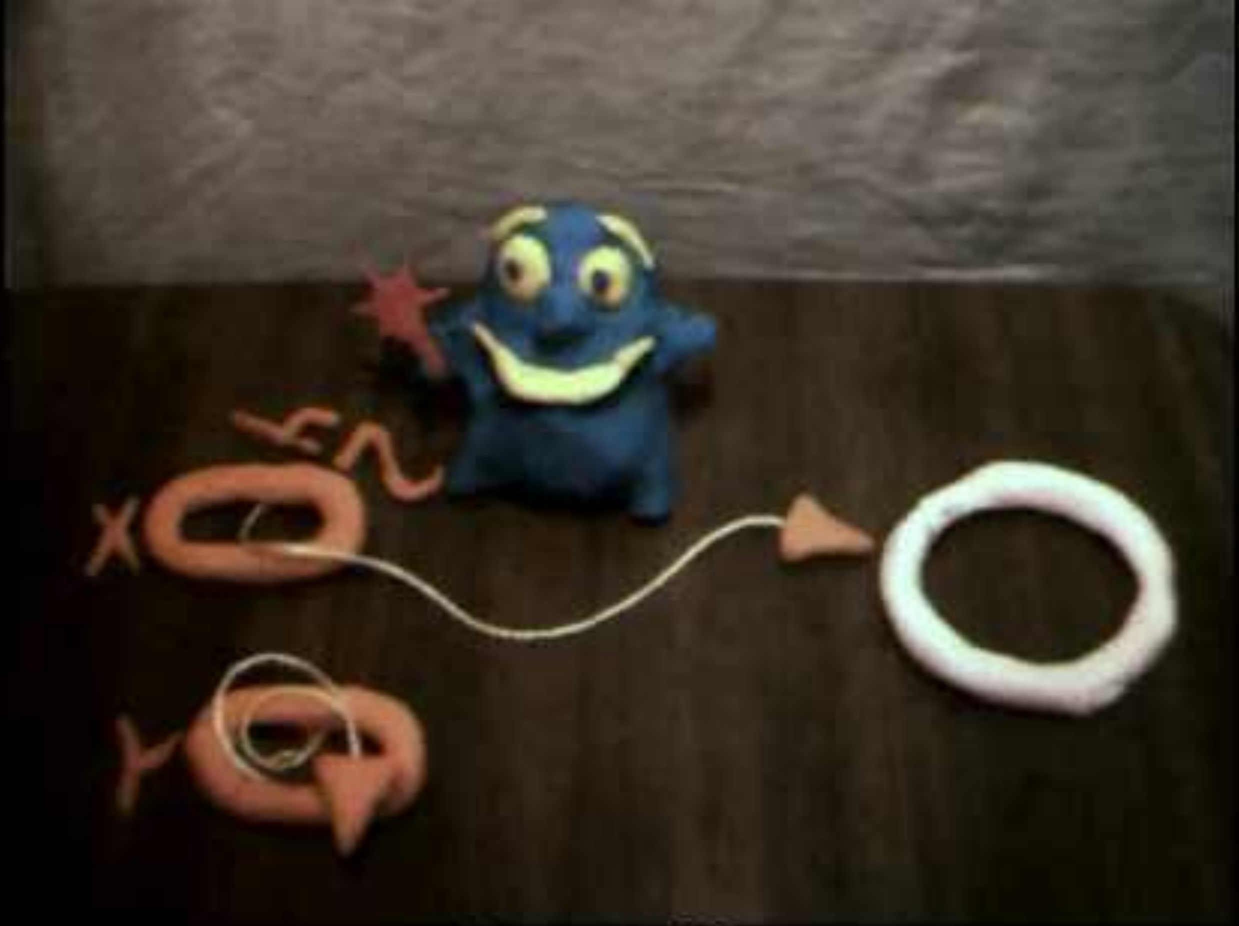
Pointers and Memory

- Every variable you create has an address in memory on your computer (either on the stack or the heap)

- A pointer is just a type of variable that stores a memory address!

- When you **dynamically allocate** variables on the heap, you must use the keyword **`new`** (or **`new[]`** for arrays) and must store the address in a pointer to keep track of it.

- To get the value located at the memory address stored in a pointer, you must **dereference** the pointer using the **`*`** operator (e.g. **`cout << *number << endl;`**).



```
*x = 42;
```

Today: Using pointers in practice

Today: Using pointers in practice

How can we use pointers to organize non-contiguous memory on the heap?

Today: Using pointers in practice

How can we use pointers to organize non-contiguous memory on the heap?

Not arrays!

Levels of abstraction

What is the interface for the user?

How is our data organized?

What stores our data?
(arrays, linked lists)

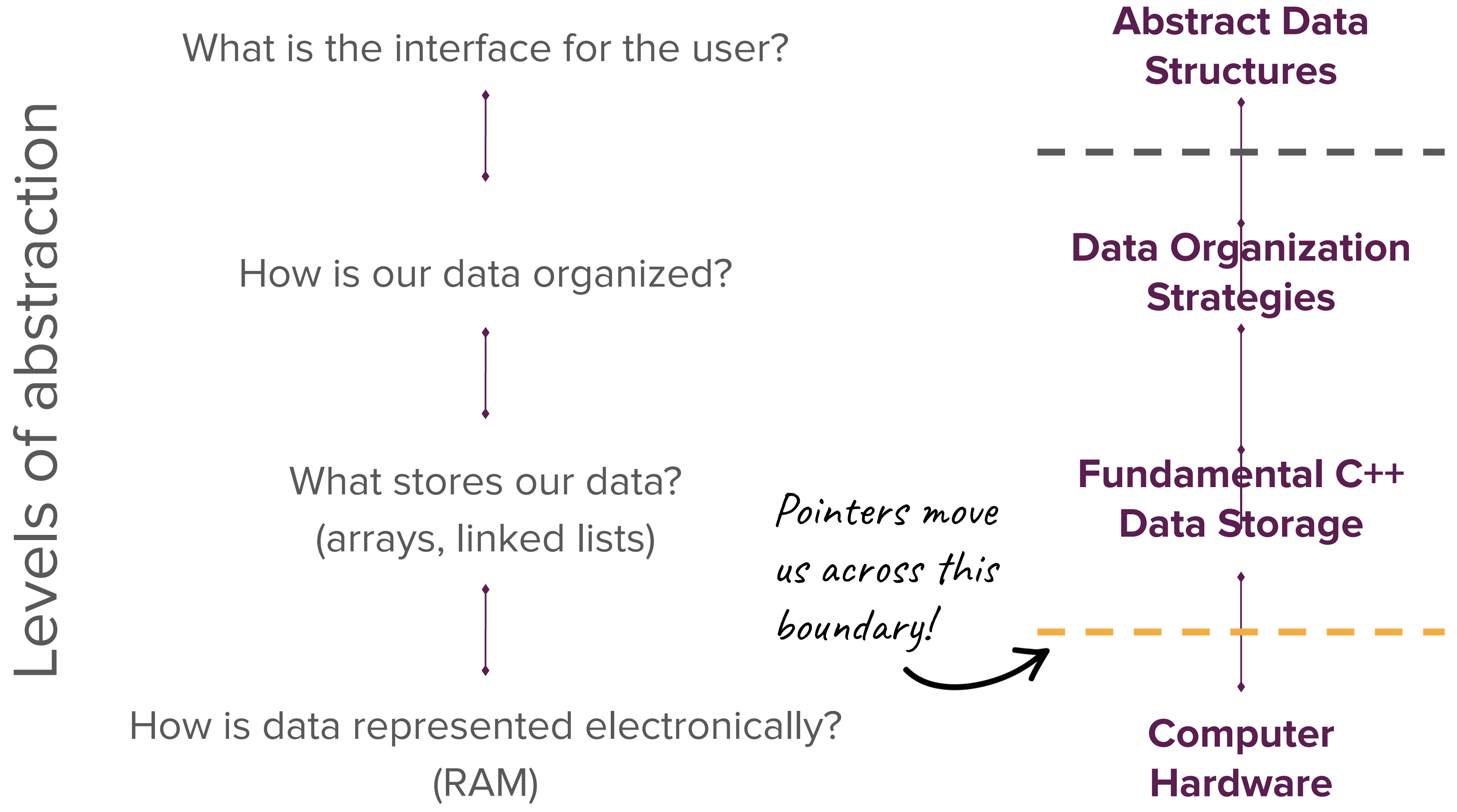
How is data represented electronically?
(RAM)

Abstract Data Structures

Data Organization Strategies

Fundamental C++ Data Storage

Computer Hardware


Levels of abstraction
What is the interface for the user?
How is our data organized?
What stores our data?
(arrays, linked lists)
How is data represented electronically?
(RAM)
Abstract Data Structures
Data Organization Strategies
Fundamental C++ Data Storage
Computer Hardware
Pointers move us across this boundary!

Levels of abstraction

What is the interface for the user?

How is our data organized?

What stores our data?
(**arrays**, **linked lists**)

These are built on top of pointers!

How is data represented electronically?
(RAM)

Abstract Data Structures

Data Organization Strategies

Fundamental C++ Data Storage

Computer Hardware

Levels of abstraction

What is the interface for the user?

Abstract Data Structures

How is our data organized?

Data Organization Strategies

What stores our data?
(arrays, **linked lists**)

Our focus for today!

Fundamental C++ Data Storage

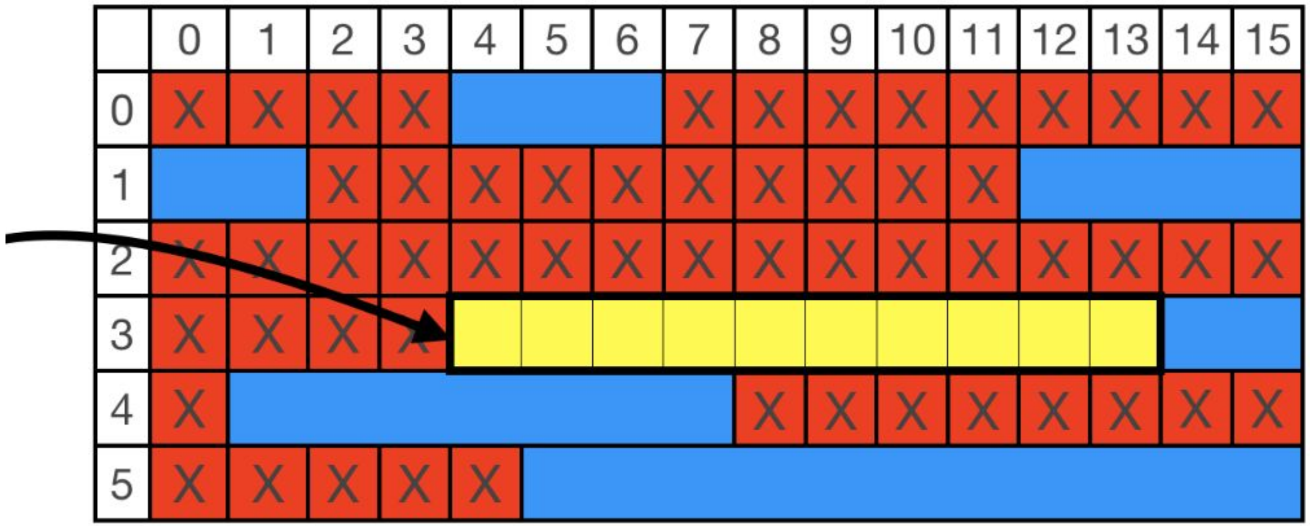
How is data represented electronically?
(RAM)

Computer Hardware

What’s wrong with arrays?

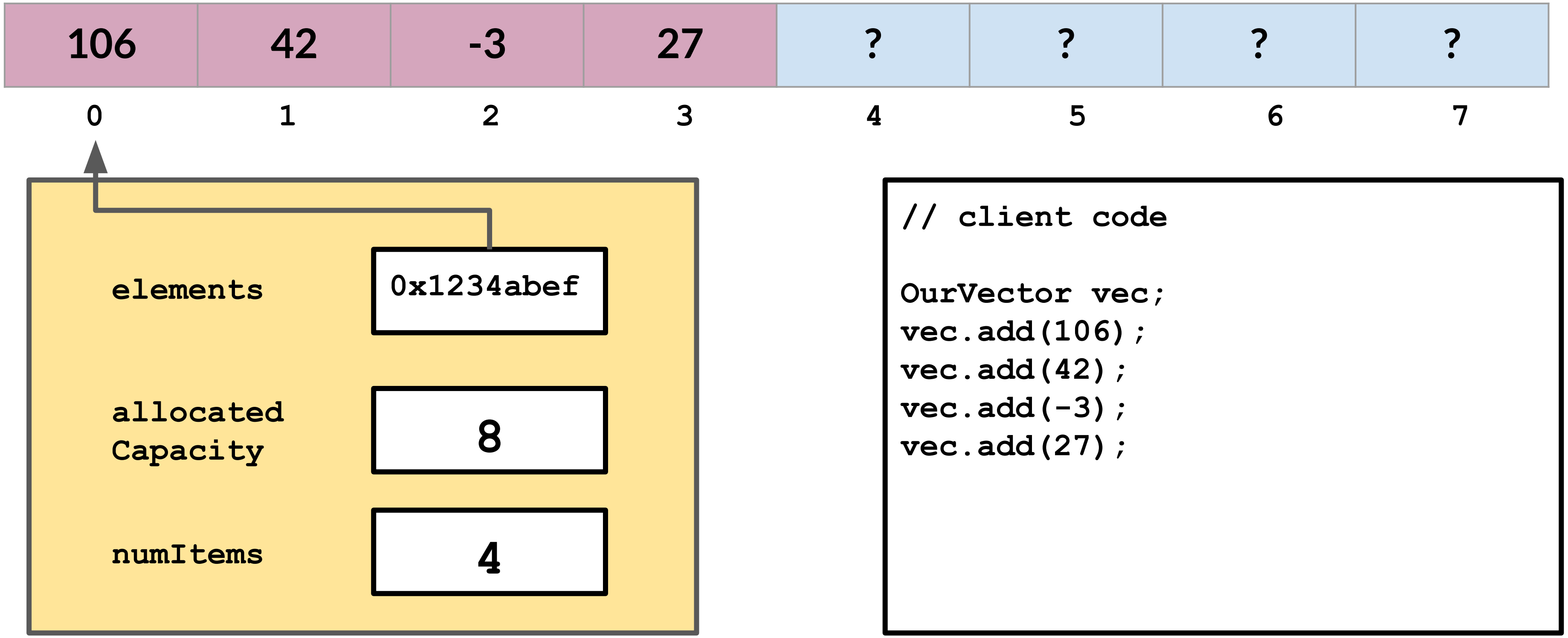
```
int* tenInts = new int[10];
```

The OS will find a contiguous array for 10 integers and give you that memory back



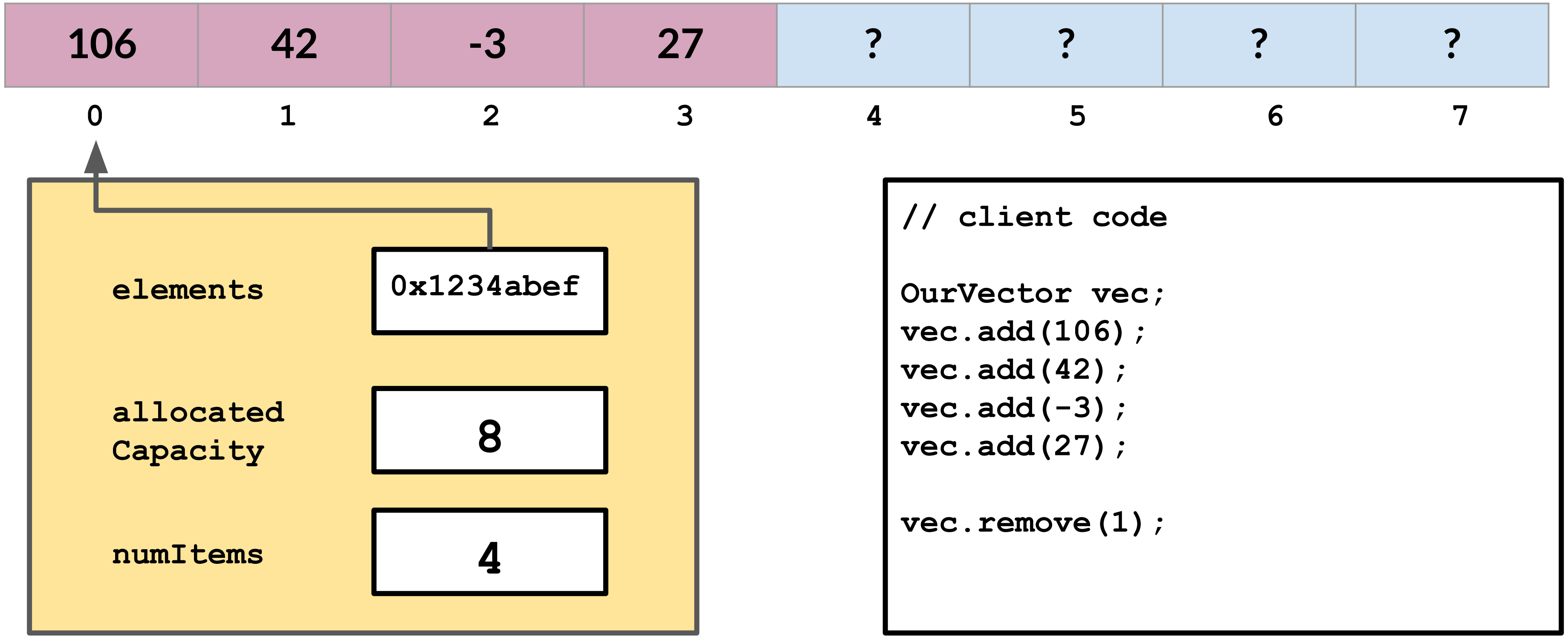
Credit: Neel Kishnani, Chris Gregg

The `remove()` operation



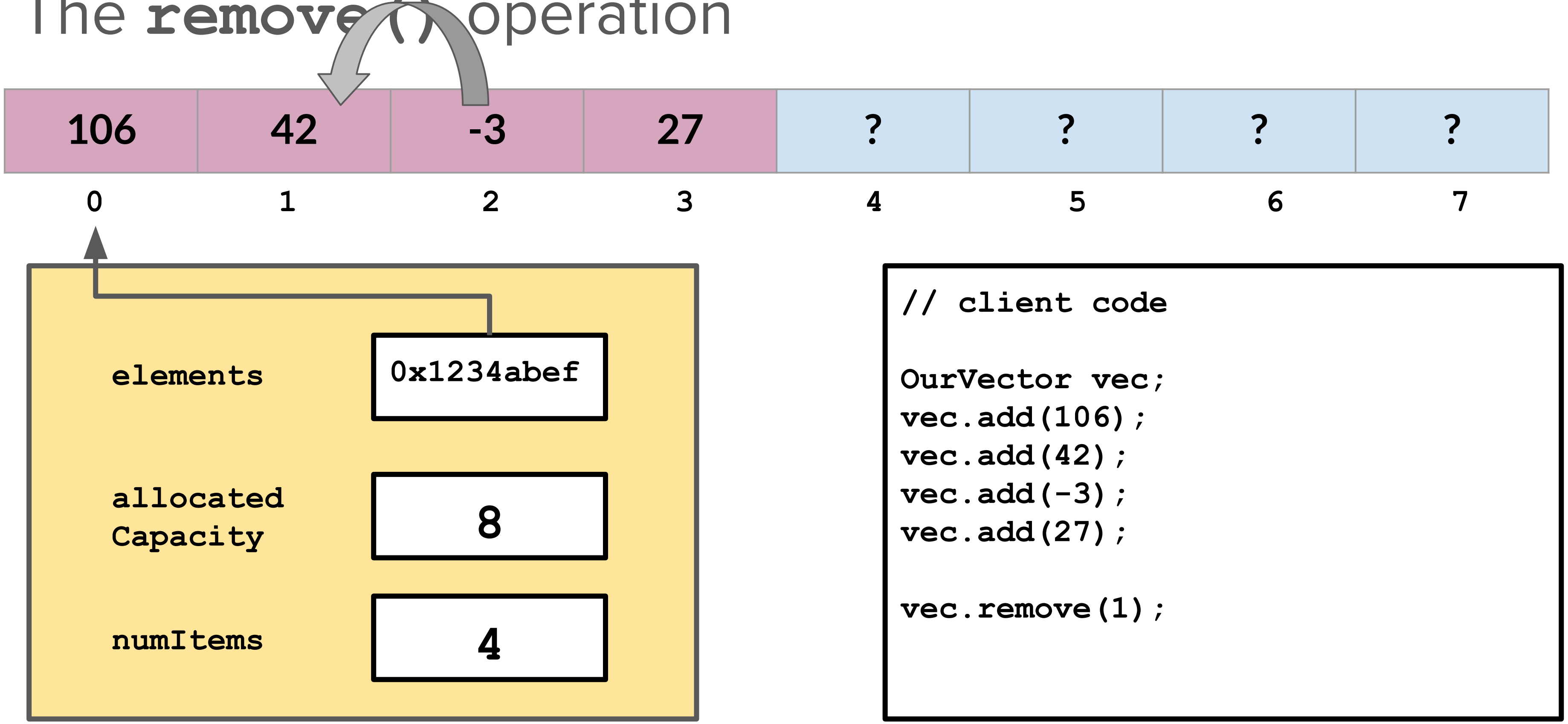
```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);
```

The **`remove()`** operation



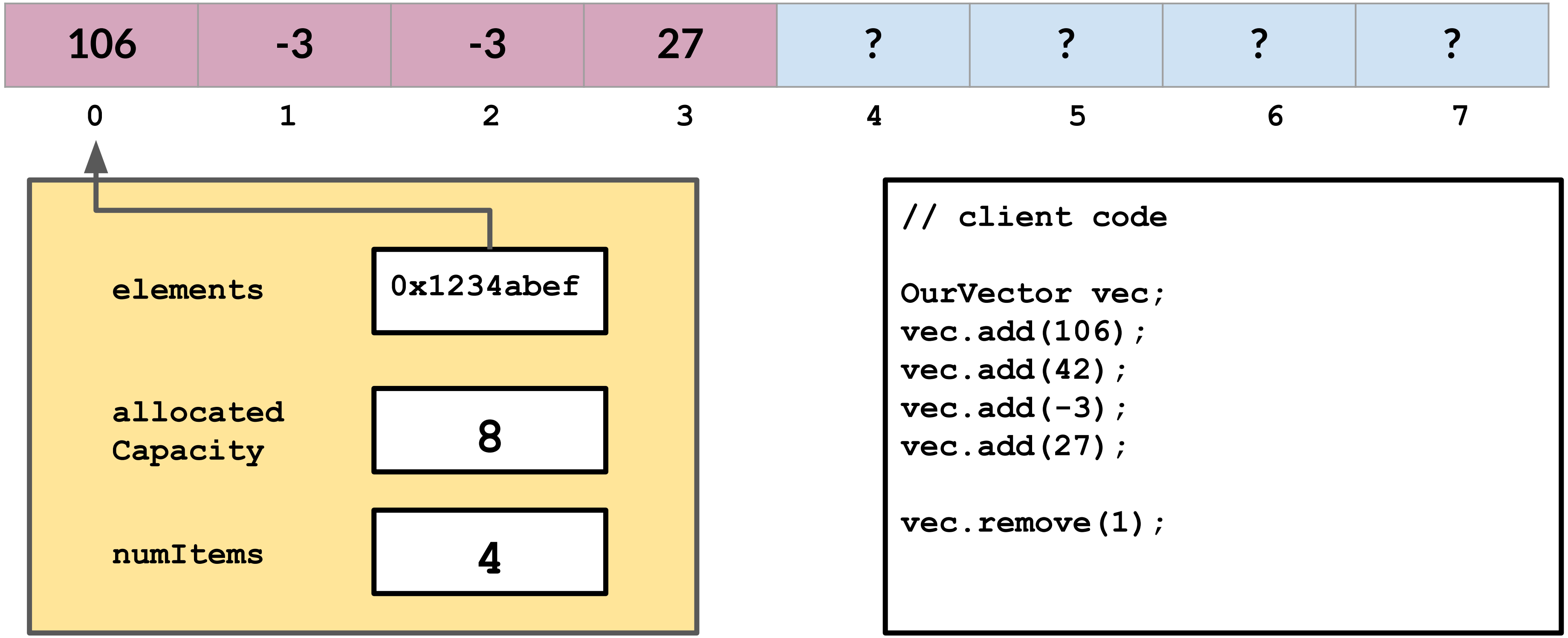
```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
```

The **`remove()`** operation



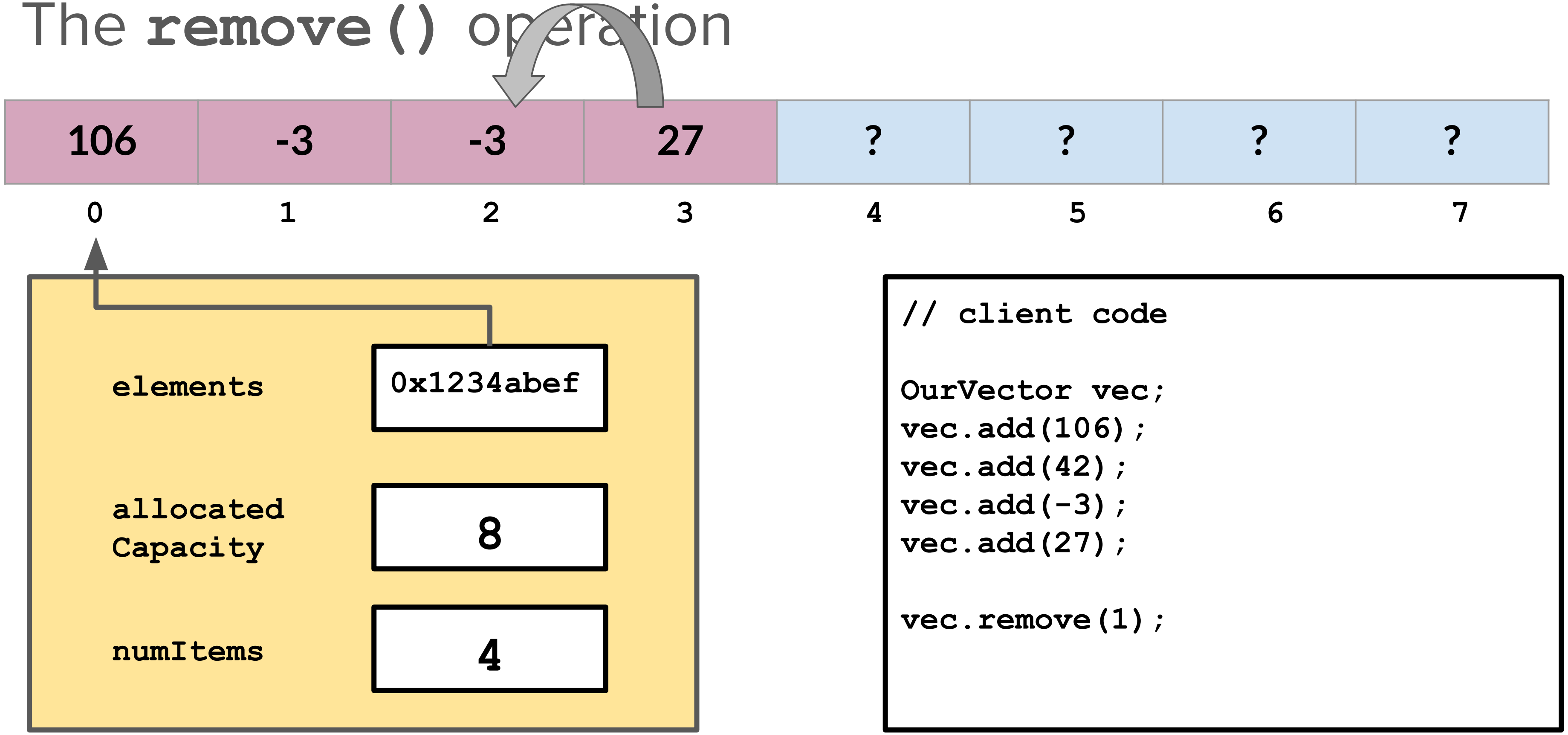
```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
```

The **`remove()`** operation



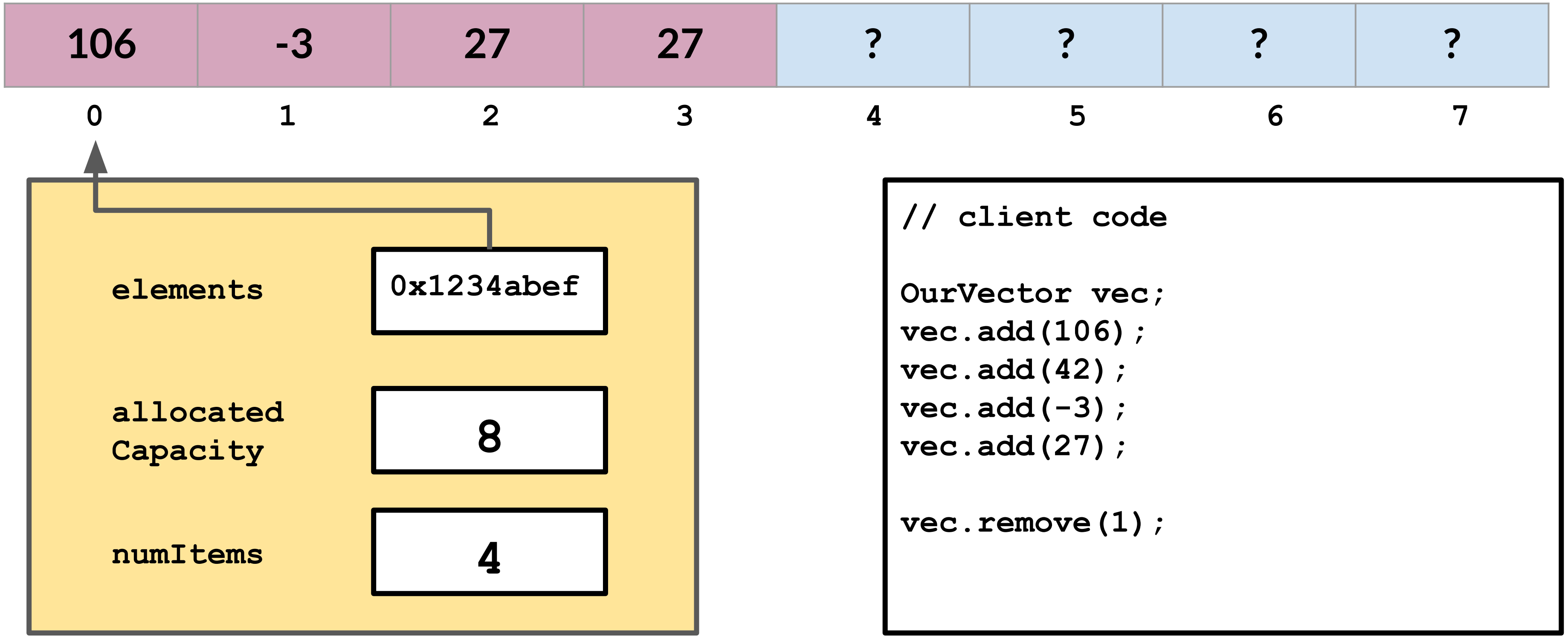
```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
```

The **`remove()`** operation



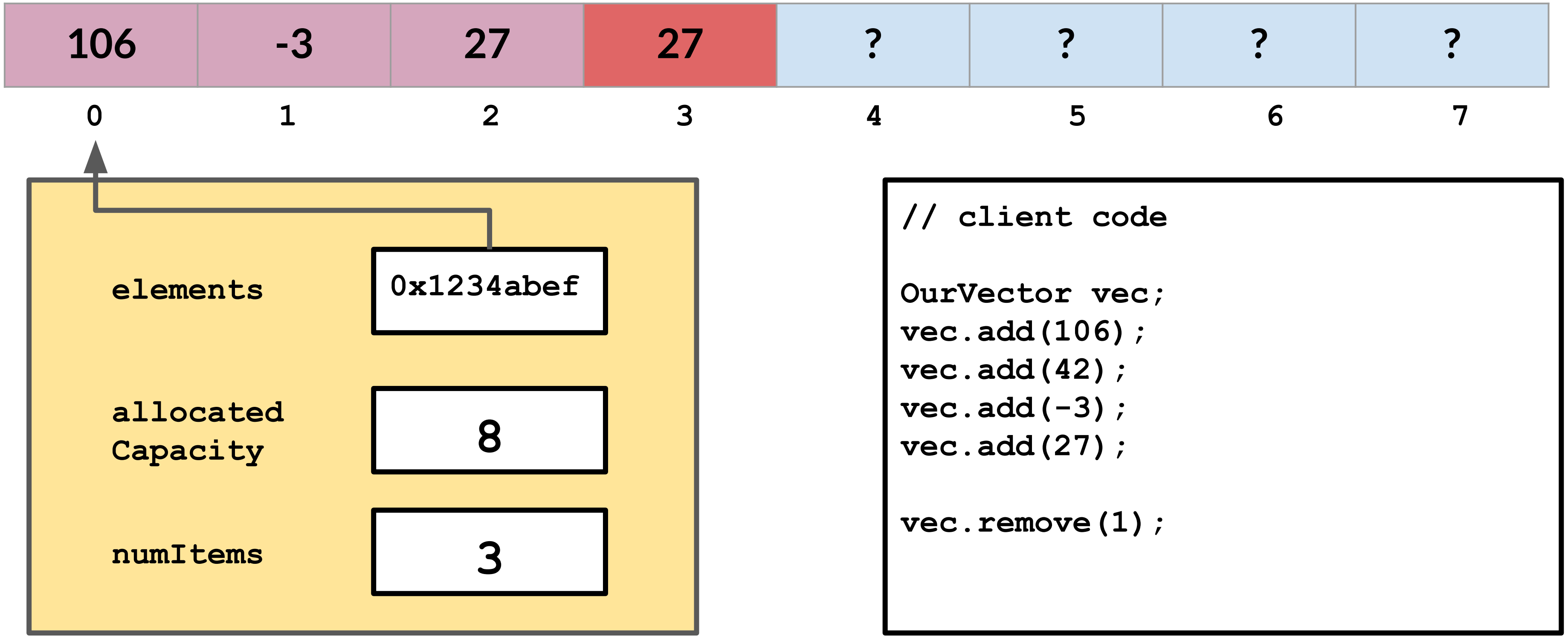
```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
```

The **`remove()`** operation



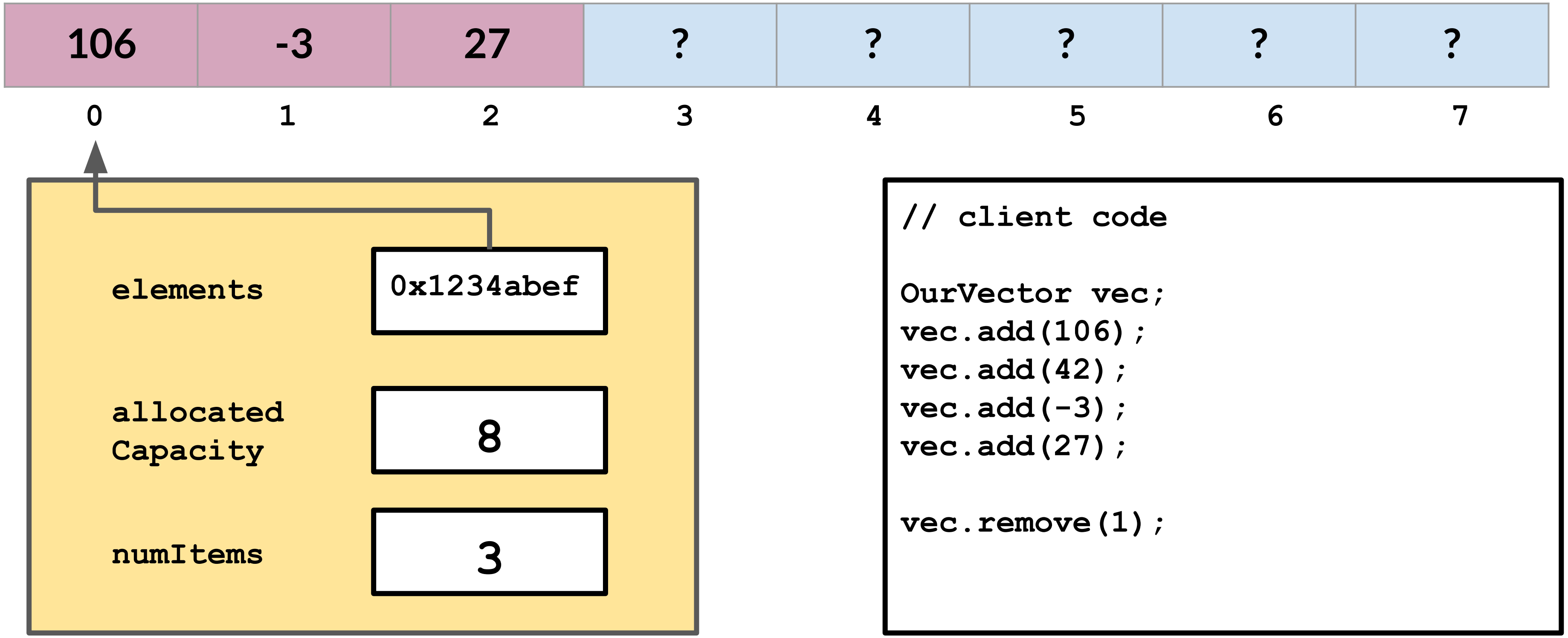
```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
```

The `remove()` operation



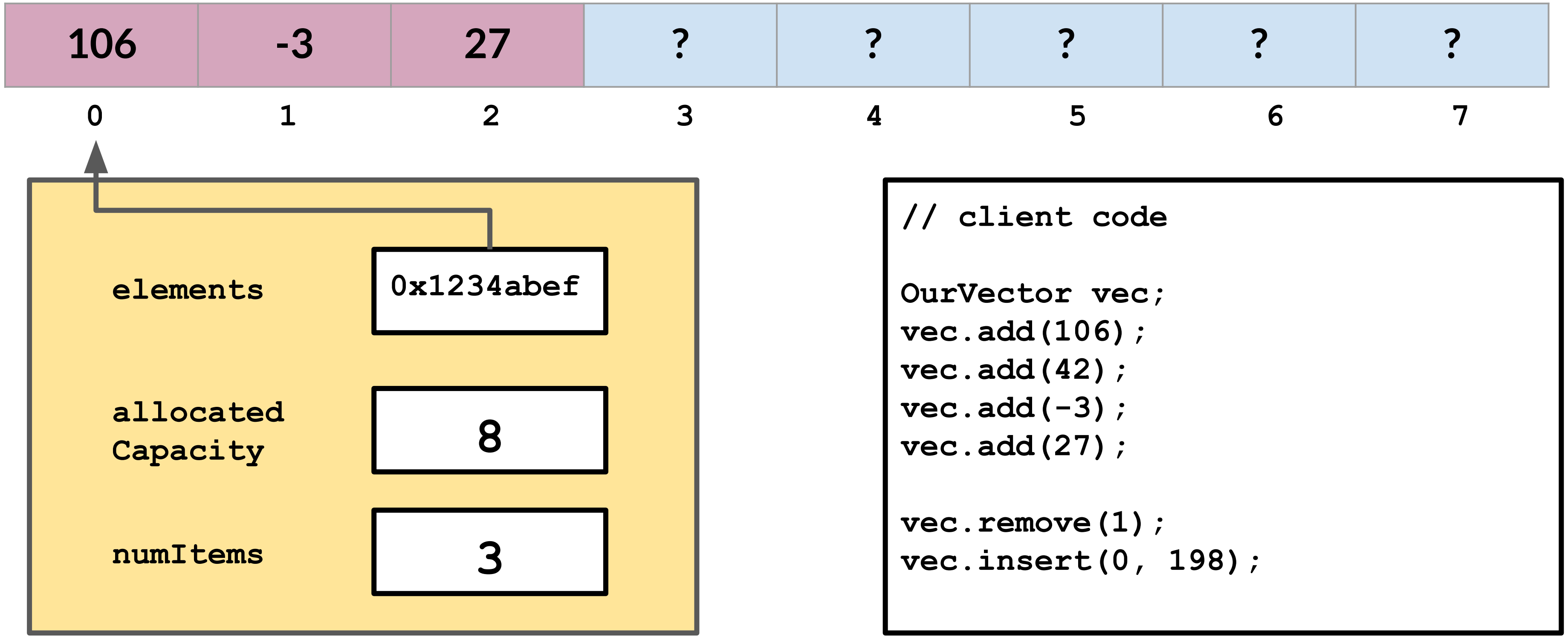
```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
```

The **`insert()`** operation



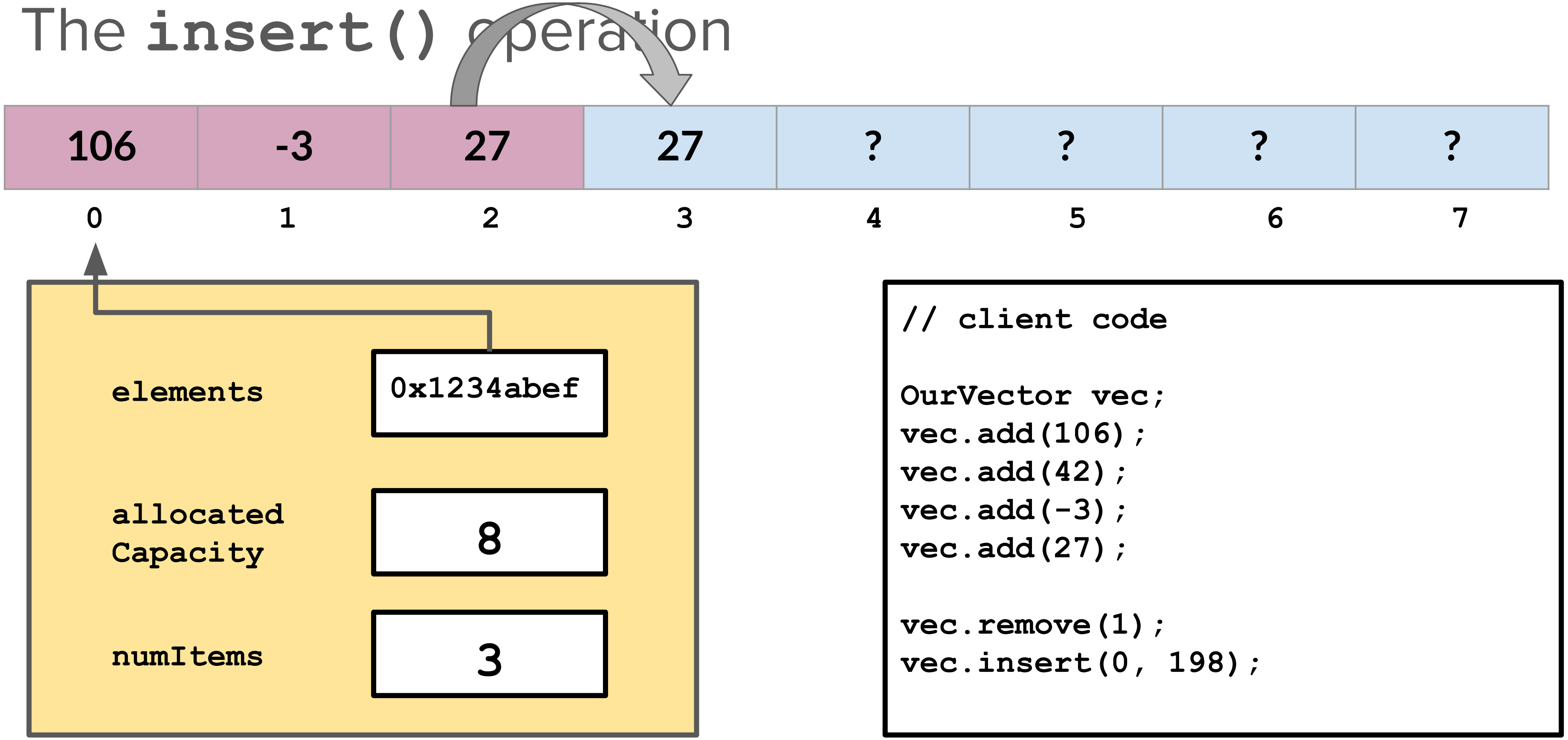
```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
```

The `insert()` operation



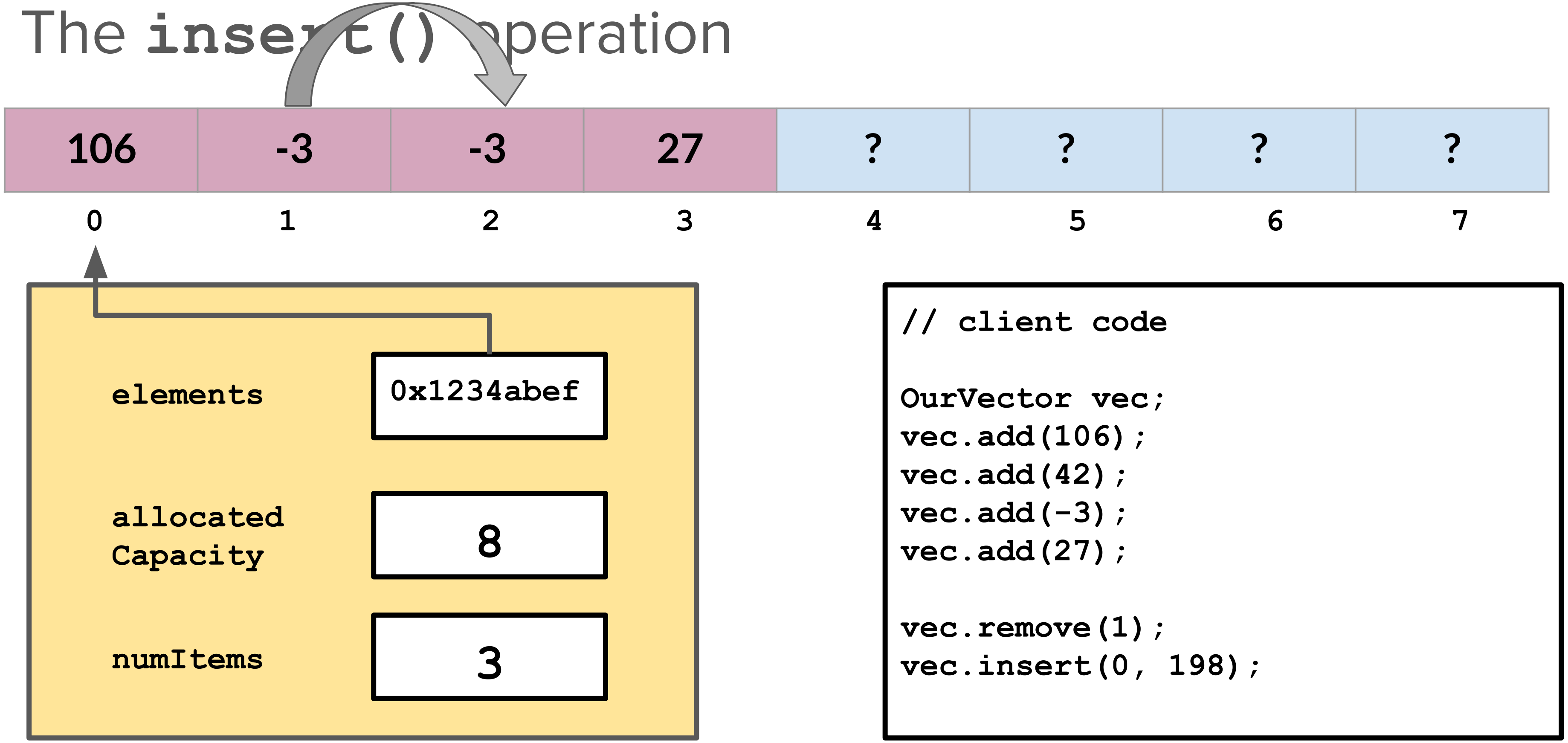
```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
vec.insert(0, 198);
```

The **`insert()`** operation



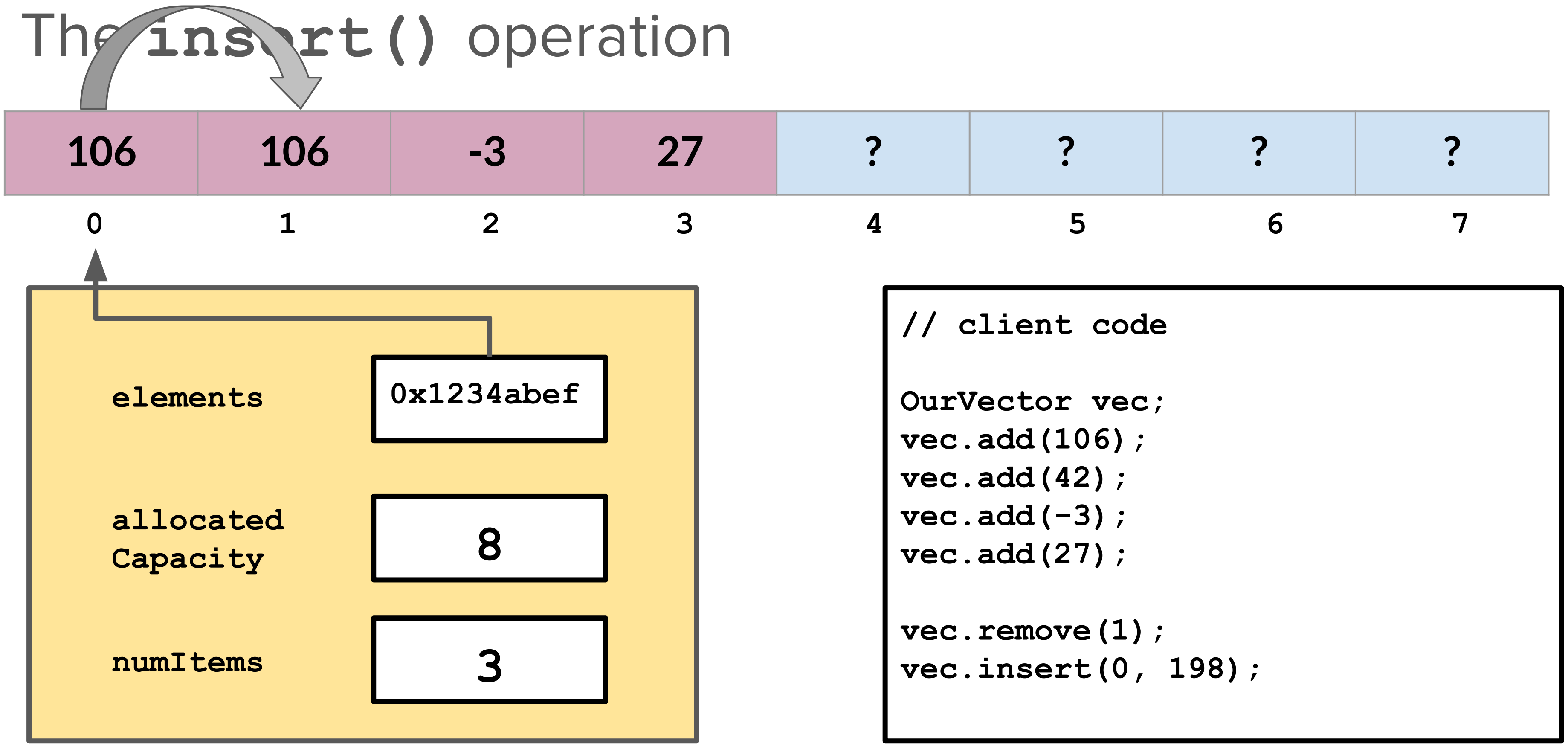
```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
vec.insert(0, 198);
```

The **insert()** operation



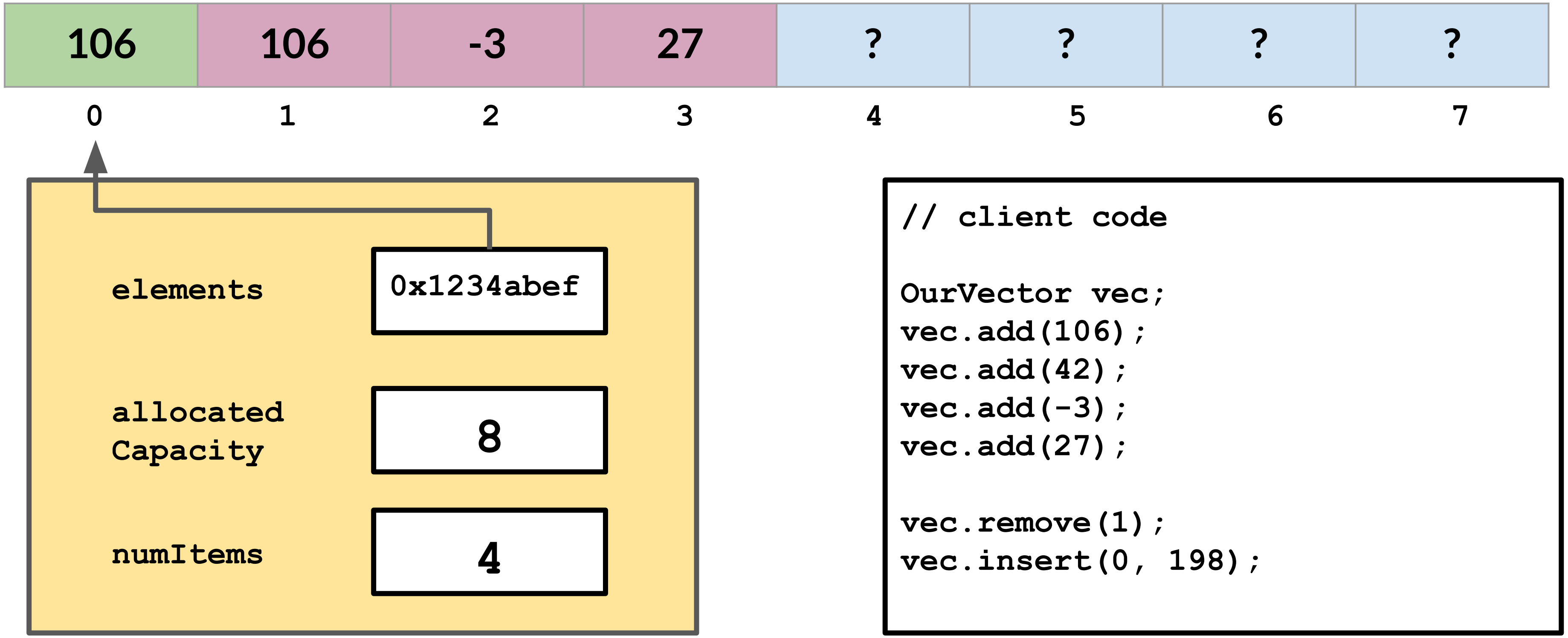
```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
vec.insert(0, 198);
```

The `insert()` operation



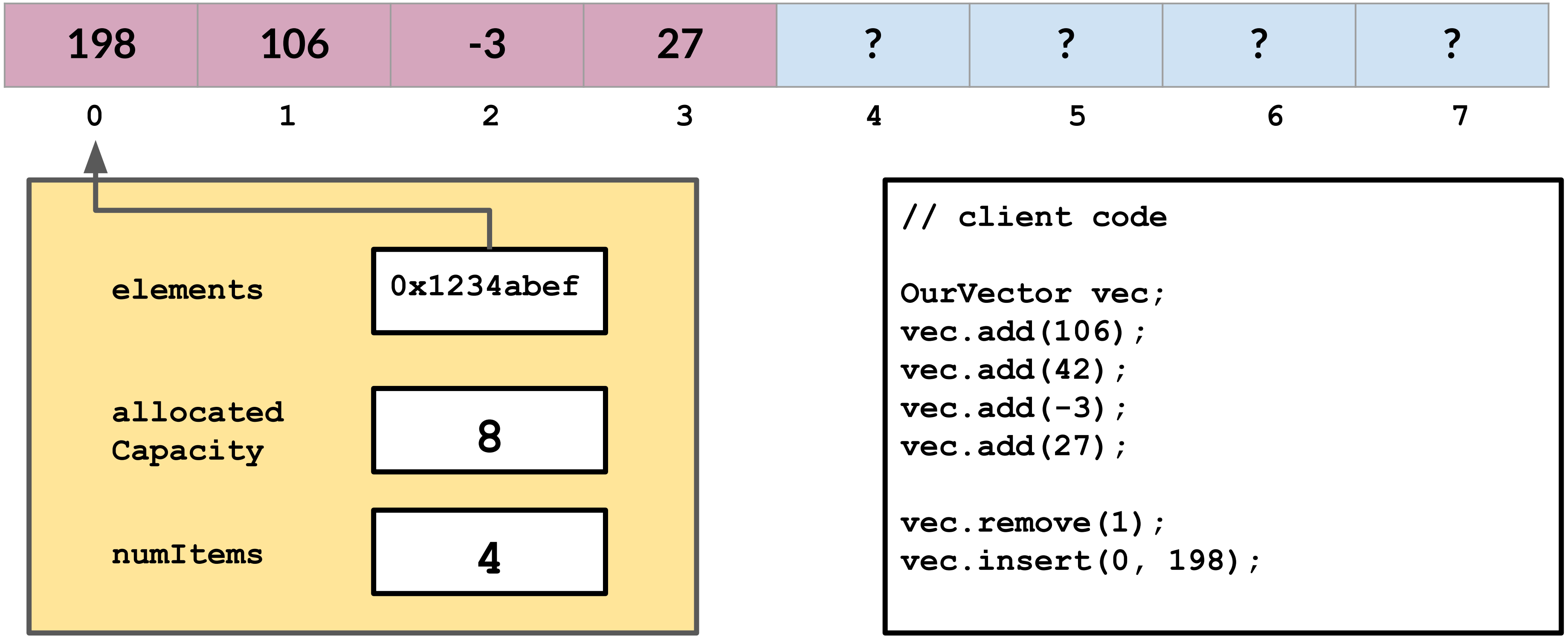
```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
vec.insert(0, 198);
```

The `insert()` operation



```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
vec.insert(0, 198);
```

The `insert()` operation



```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
vec.insert(0, 198);
```

A Day in the Life of a Growable Array

- In essence, when we run out of space in our array, we want to allocate a new array that is bigger than our old array so we can store the new data and keep growing. These "growable arrays" follow a five-step expansion that mirrors the hermit crab model (with poetic license).
  - Grow the array until we run out of space (how can we tell if we've run out of space?)
    - Create a new, larger array. Usually we choose to **double** the current size.
    - Copy the old array elements to the new array.
    - Delete (free) the old array.
    - Point the old array variable to the new array.
    - Update the associated capacity variable for the array.

Can we do better?

- A way to store elements as a sequence even if they’re not physically next to each other on the computer memory
    - So we can easily insert new elements into the list
    - So we can easily remove elements from the list
    - So we can easily resize the list
    - (So we don’t have to mass copy elements and shift them over or shift them into a new block of memory)

Can we do better?

- Nope. Class for the rest of the quarter is cancelled; computing as we know it has been a standstill since 1954.

(just kidding)

What is a linked list?

What is a linked list?

- A linked list is a **chain of nodes**.

What is a linked list?

- A linked list is a **chain of nodes**.

- Each **node** contains two pieces of information:
  - Some piece of data that is stored in the sequence
  - A link to the next node in the list

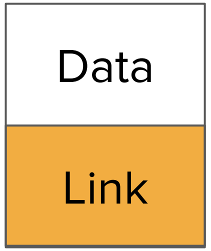
What is a linked list?

- A linked list is a **chain of nodes**.

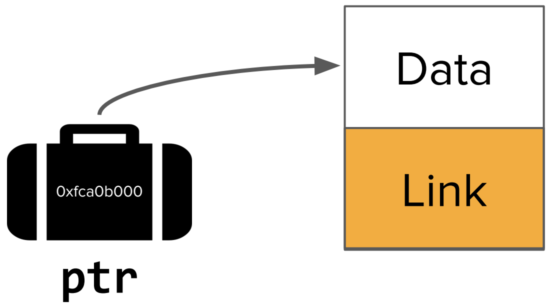
- Each **node** contains two pieces of information:
    - Some piece of data that is stored in the sequence
    - A link to the next node in the list

- We can traverse the list by starting at the first node and repeatedly following its link.

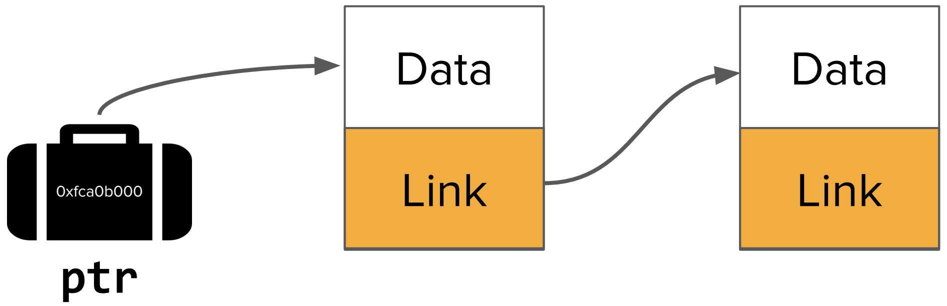
Node



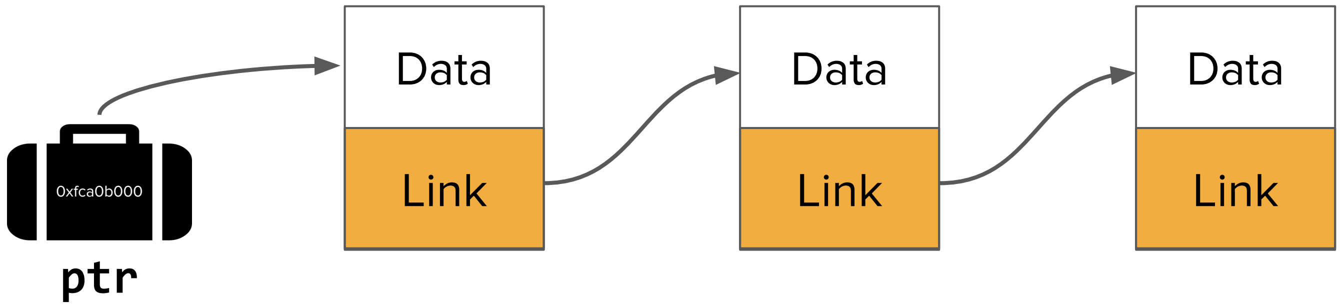
Pointer to a node



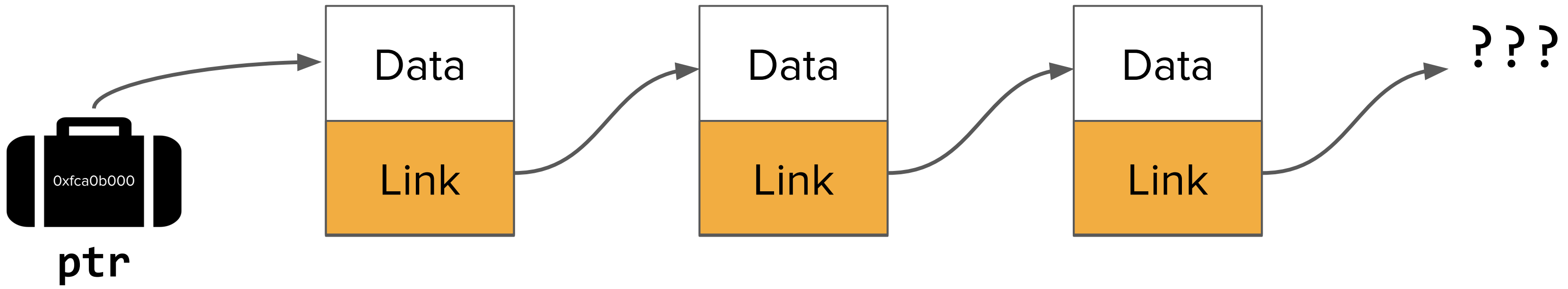
Pointer to a node that points to a node



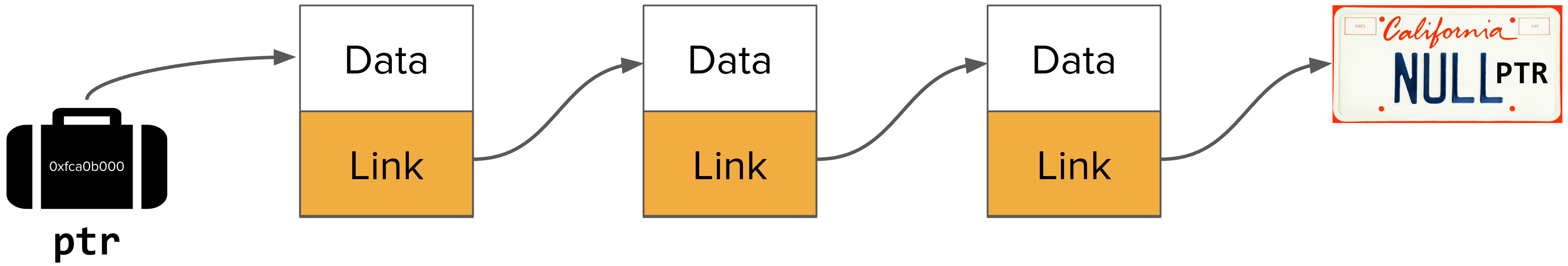
Pointer to a node that points to a node that points to a node



Pointer to a node that points to a node that points to a node



A linked list!



Why use linked lists?

- More flexible than arrays!
  - Since they’re not contiguous, they’re easier to rearrange.

- We can efficiently splice new elements into the list or remove existing elements anywhere in the list. (We’ll see how shortly!)

- We never have to do a massive copy step.

- But linked lists still have many tradeoffs and are not always the best data structure!

Linked lists in C++

The **Node** struct

```
struct Node {
    string data;
    Node* next;
}
```

The **Node** struct

```
struct Node {
    string data;
    Node* next;
}
```

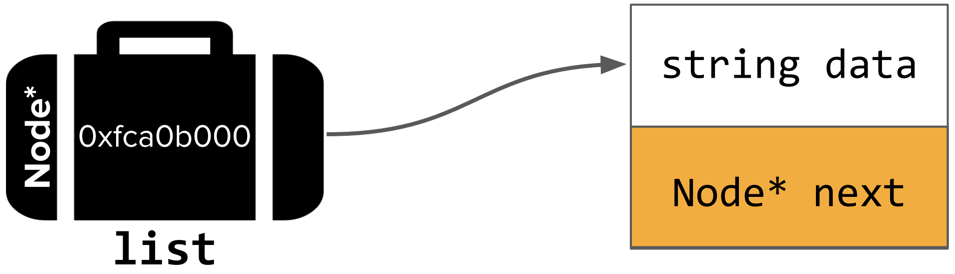
- The structure is defined recursively! (both the Node and the linked list itself)

The **Node** struct

```
struct Node {
    string data;
    Node* next;
}
```

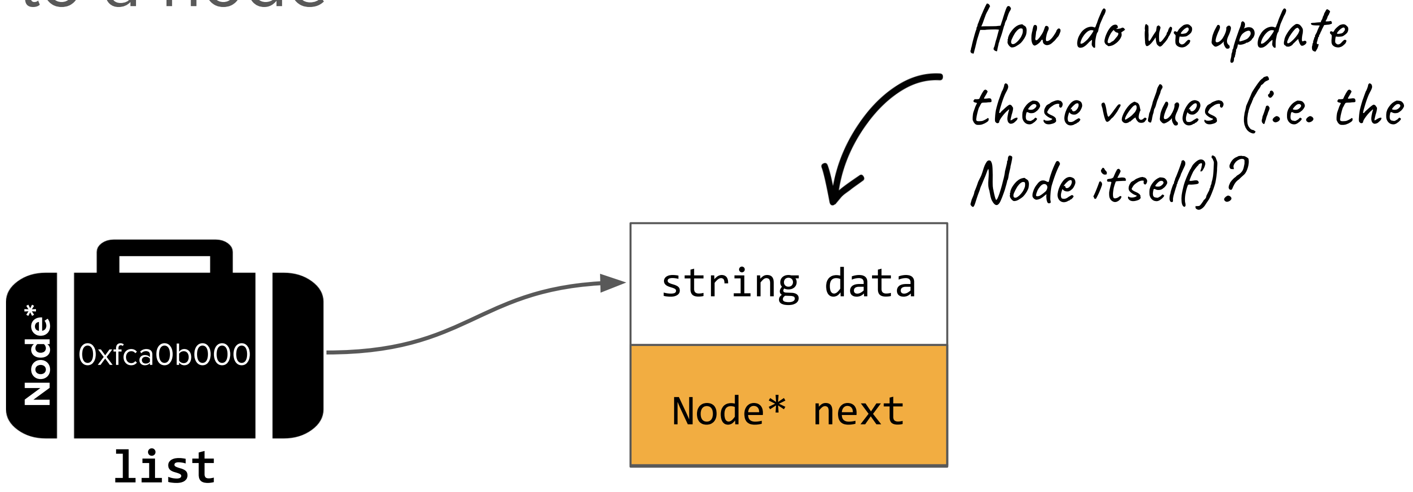
- The structure is defined recursively! (both the Node and the linked list itself)
- The compiler can handle the fact that in the definition of the **Node** there is a **Node*** because it knows it is simply a pointer.
    - (It would be impossible to recursively define the **Node** with an actual **Node** object inside the struct.)

Pointer to a node



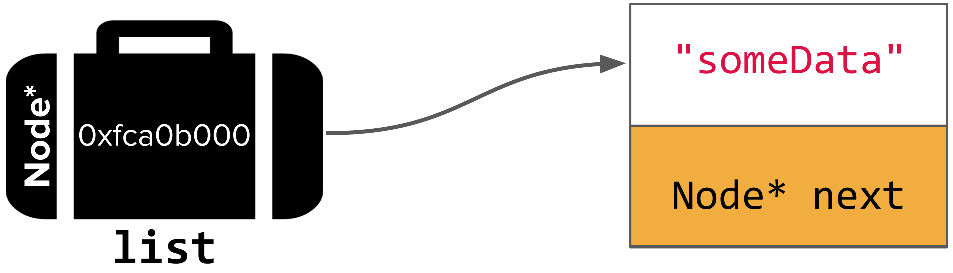
```
Node* list = new Node;
```

Pointer to a node



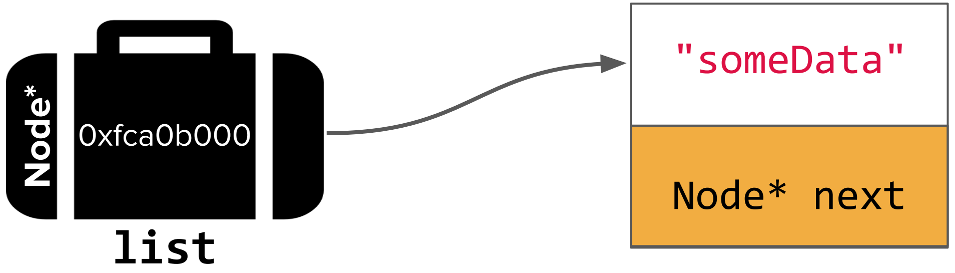
```
Node* list = new Node;
```

Pointer to a node



```
Node* list = new Node;
(*list).data = "someData";
```

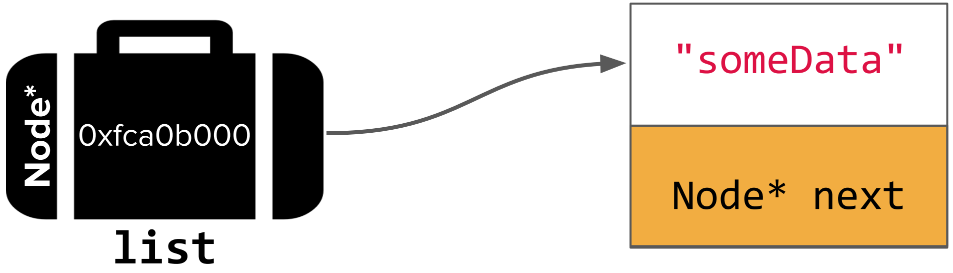
Pointer to a node



```
Node* list = new Node;
(*list).data = "someData";
```

Use * *to dereference the pointer to get the Node struct.*

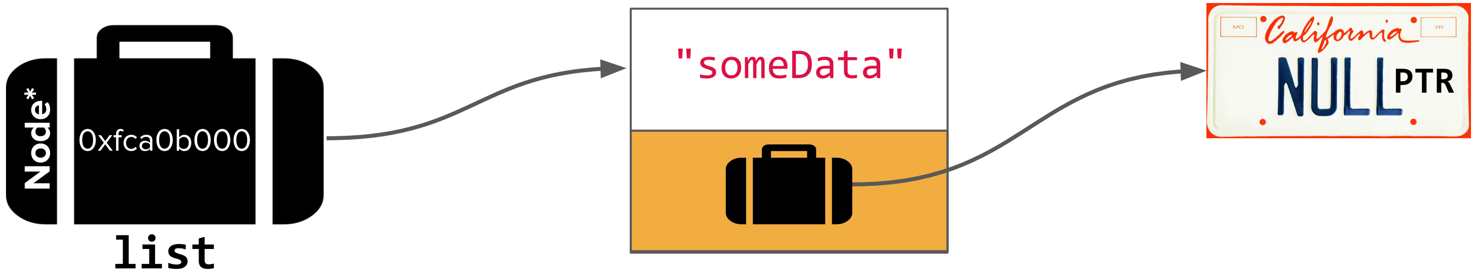
Pointer to a node



```
Node* list = new Node;
(*list).data = "someData";
```

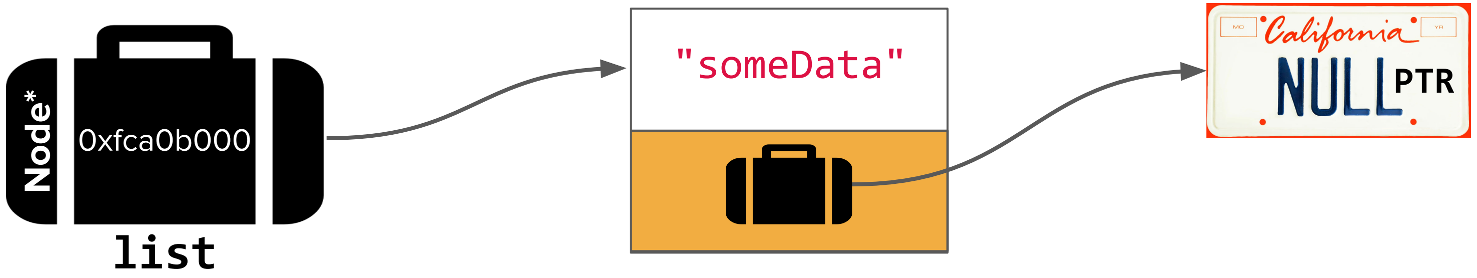
Use dot (.) notation to update the data field of the struct.

Pointer to a node



```
Node* list = new Node;
(*list).data = "someData";
(*list).next = nullptr;
```

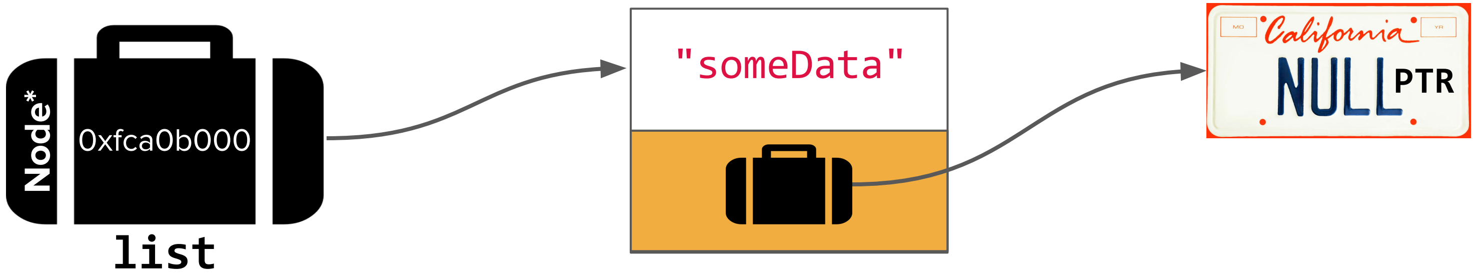
Pointer to a node



```
Node* list = new Node;
(*list).data = "someData";
(*list).next = nullptr;
```

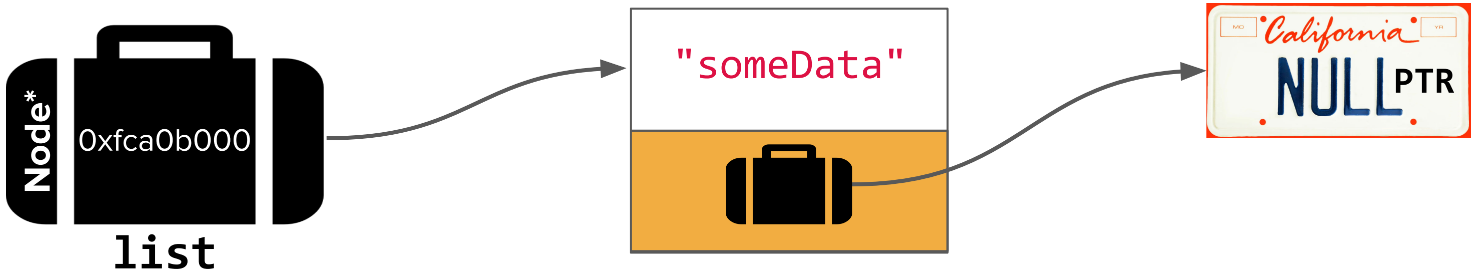
There's an easier way!

Pointer to a node



```
Node* list = new Node;
list->data = "someData";
list->next = nullptr;
```

Pointer to a node



```
Node* list = new Node;
list->data = "someData";
list->next = nullptr;
```

The arrow notation (->) dereferences AND accesses the field for pointers that point to structs specifically.

Announcements

Announcements

- Final project proposals were due **yesterday.** We will try to have feedback to you by Thursday or Friday.
  - In the meantime, make sure to take a look at the project timeline to stay on track!
  - Next milestone: Sunday Aug 7

- Assignment 4 is due tomorrow (with 24 hour grace period).
- Assignment 5 is out tomorrow!
  - Good use of the debugger is essential in this assignment. Use the techniques in the warm-up to help you uncover those tricky memory bugs!

How do we use linked lists in a class?

Common linked lists operations

- **Traversal**
  - How do we walk through all elements in the linked list?
- **Rewiring**
  - How do we rearrange the elements in a linked list?
- **Insertion**
  - How do we add an element to a linked list?
- **Deletion**
  - How do we remove an element from a linked list?

Implementing a Stack

Note: You could do this with an array! This is just for the sake of getting practice with linked lists.

Stack as a linked list

- We’ll keep a pointer **`Node* top`** that points to the “top” element in our stack.
    - This member var will get initialized to **`nullptr`** when our stack is empty!

Stack as a linked list

- We’ll keep a pointer **`Node* top`** that points to the “top” element in our stack.
  - This member var will get initialized to **`nullptr`** when our stack is empty!

- Our linked list nodes will be connected from the top to the bottom of our stack.

Stack as a linked list

- We'll keep a pointer **`Node* top`** that points to the "top" element in our stack.
  - This member var will get initialized to **`nullptr`** when our stack is empty!

- Our linked list nodes will be connected from the top to the bottom of our stack.

- Our stack will specifically hold integers, so our **`Node`** struct will hold an **`int`** type for our **`data`** field:

```
struct Node {
    int data;
    Node* next;
}
```

Three Stack operations

- **`push()`**
- **`pop()`**
- Destructor

Three Stack operations

- `push()`
- `pop()`
- Destructor

Common linked lists operations

- **Traversal**
  - How do we walk through all elements in the linked list?
- **Rewiring**
  - How do we rearrange the elements in a linked list?
- **Insertion (at the front)**
  - How do we add an element to a linked list?
- **Deletion**
  - How do we remove an element from a linked list?

push()

- Suppose we have the following Stack we want to push to:

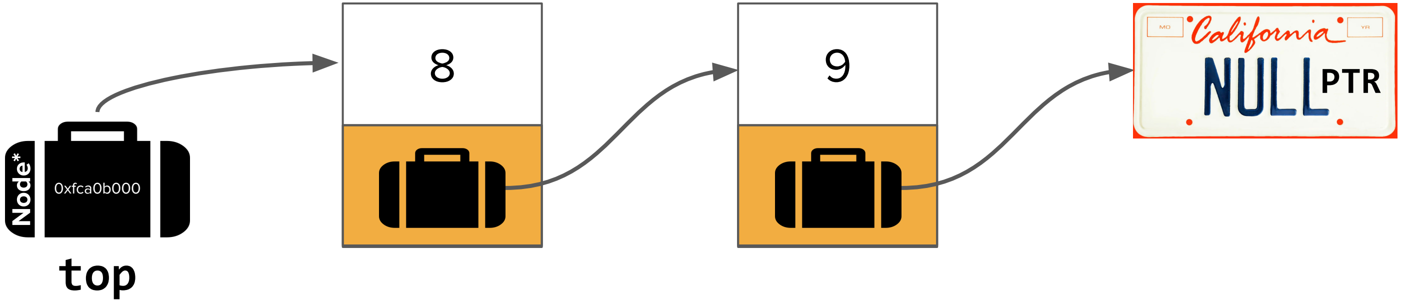
```
Stack myStack = {9, 8}; // 8 is at the "top" of the stack
myStack.push(7); // we want the result to be {9, 8, 7}
```

push()

- Suppose we have the following Stack we want to push to:

```
Stack myStack = {9, 8}; // 8 is at the "top" of the stack
myStack.push(7); // we want the result to be {9, 8, 7}
```

How our linked list starts:

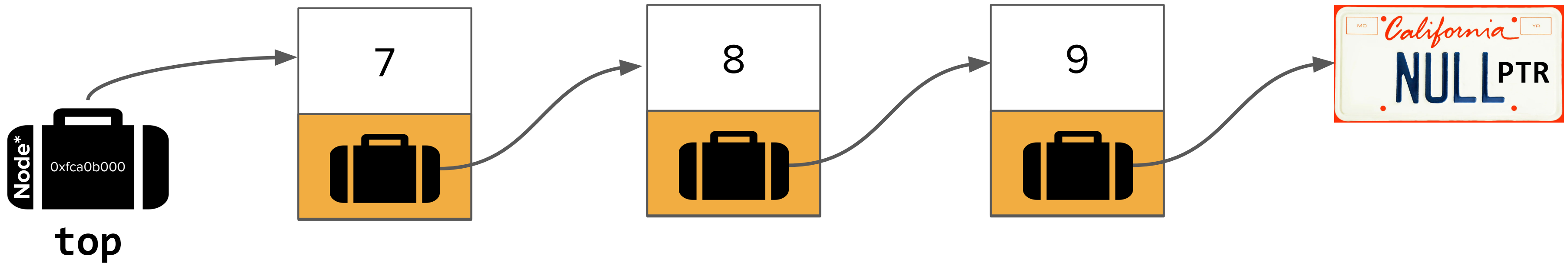


push()

- Suppose we have the following Stack we want to push to:

```
Stack myStack = {9, 8}; // 8 is at the "top" of the stack
myStack.push(7); // we want the result to be {9, 8, 7}
```

Goal:

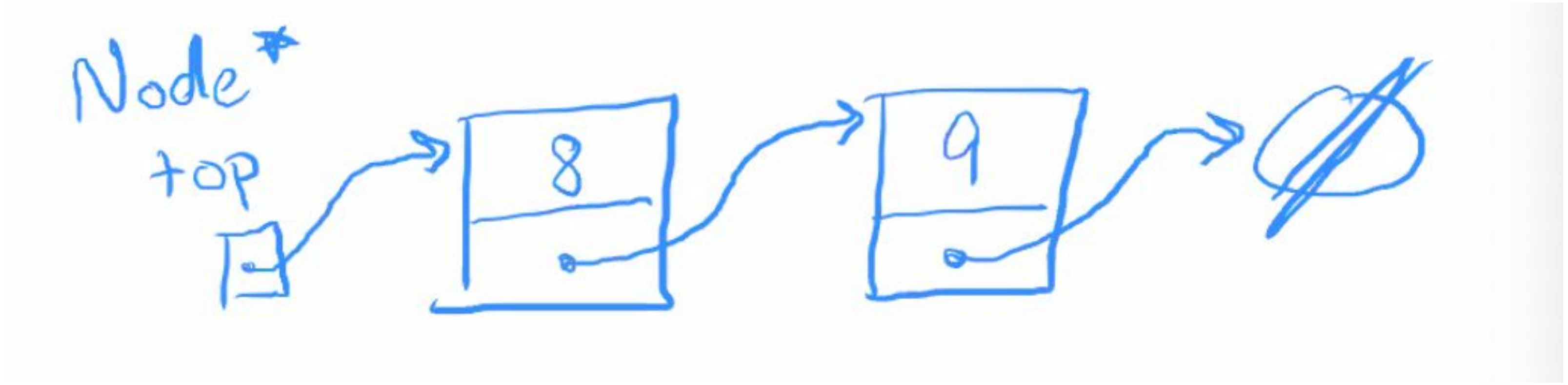


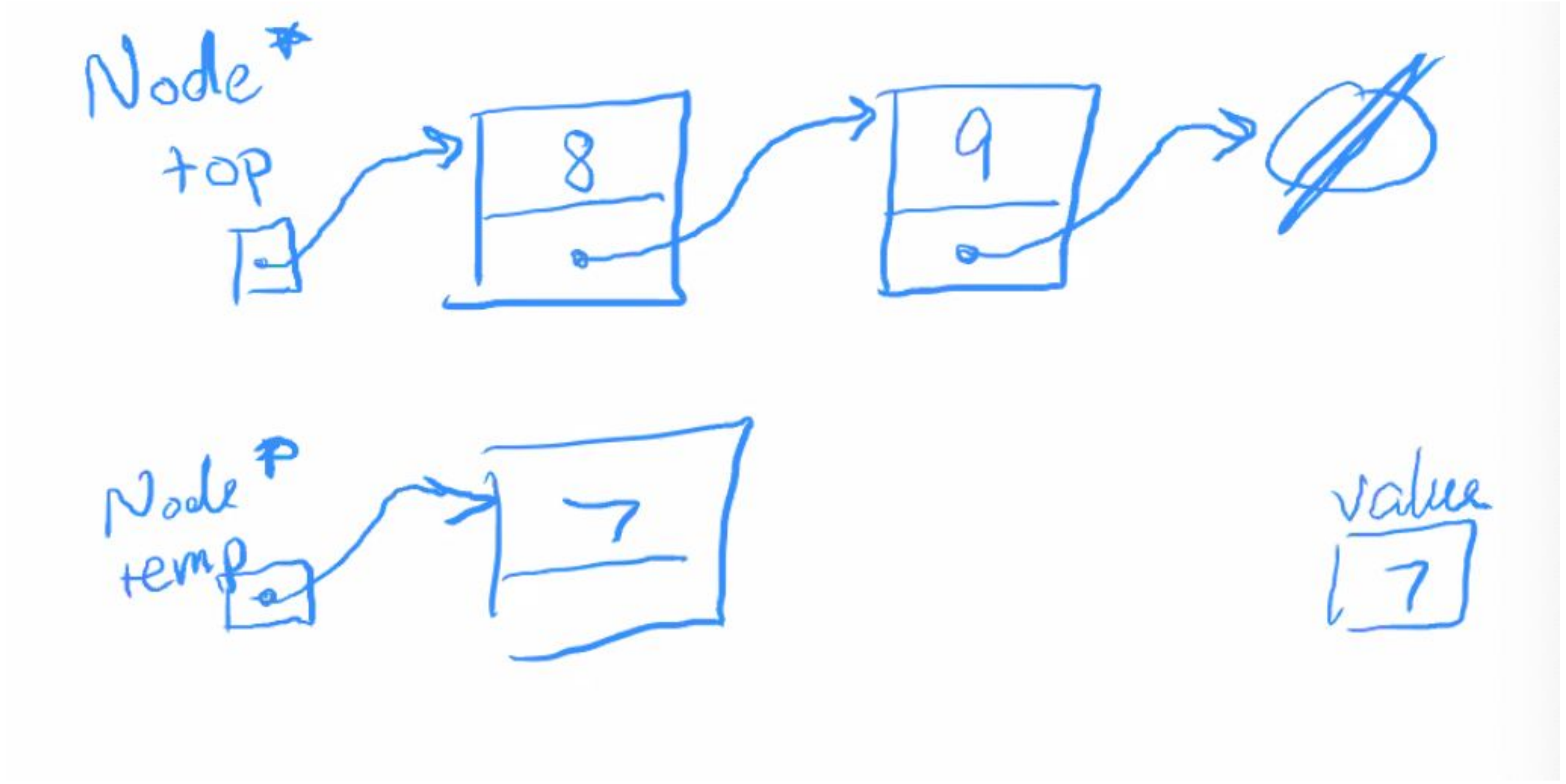
Let’s code **`push()`**!

Live Activity Summary

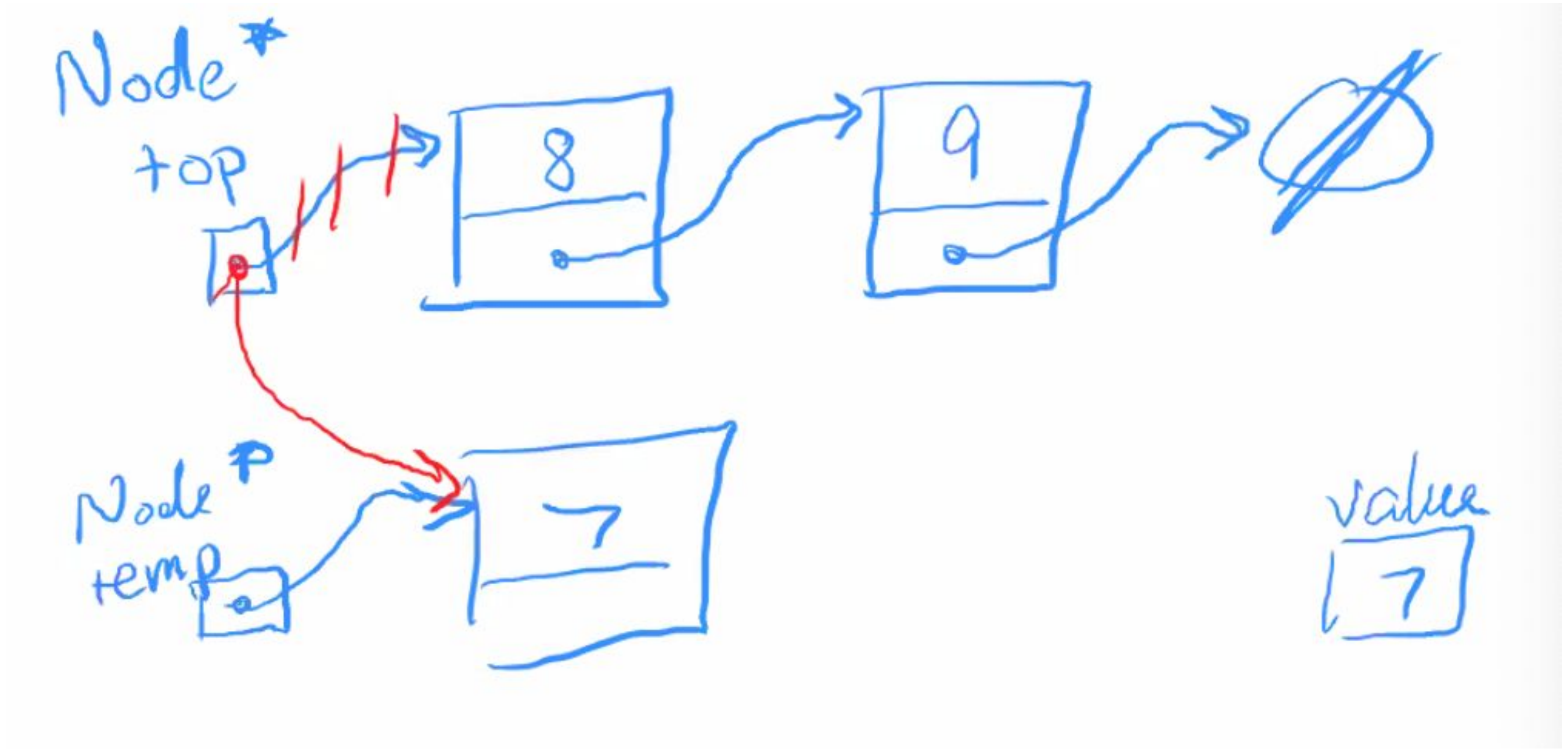
- We strongly recommend watching the live recording of the coding activity, as the code and explanations contextualize the following diagrams

Initial State (beginning of **push()** function)

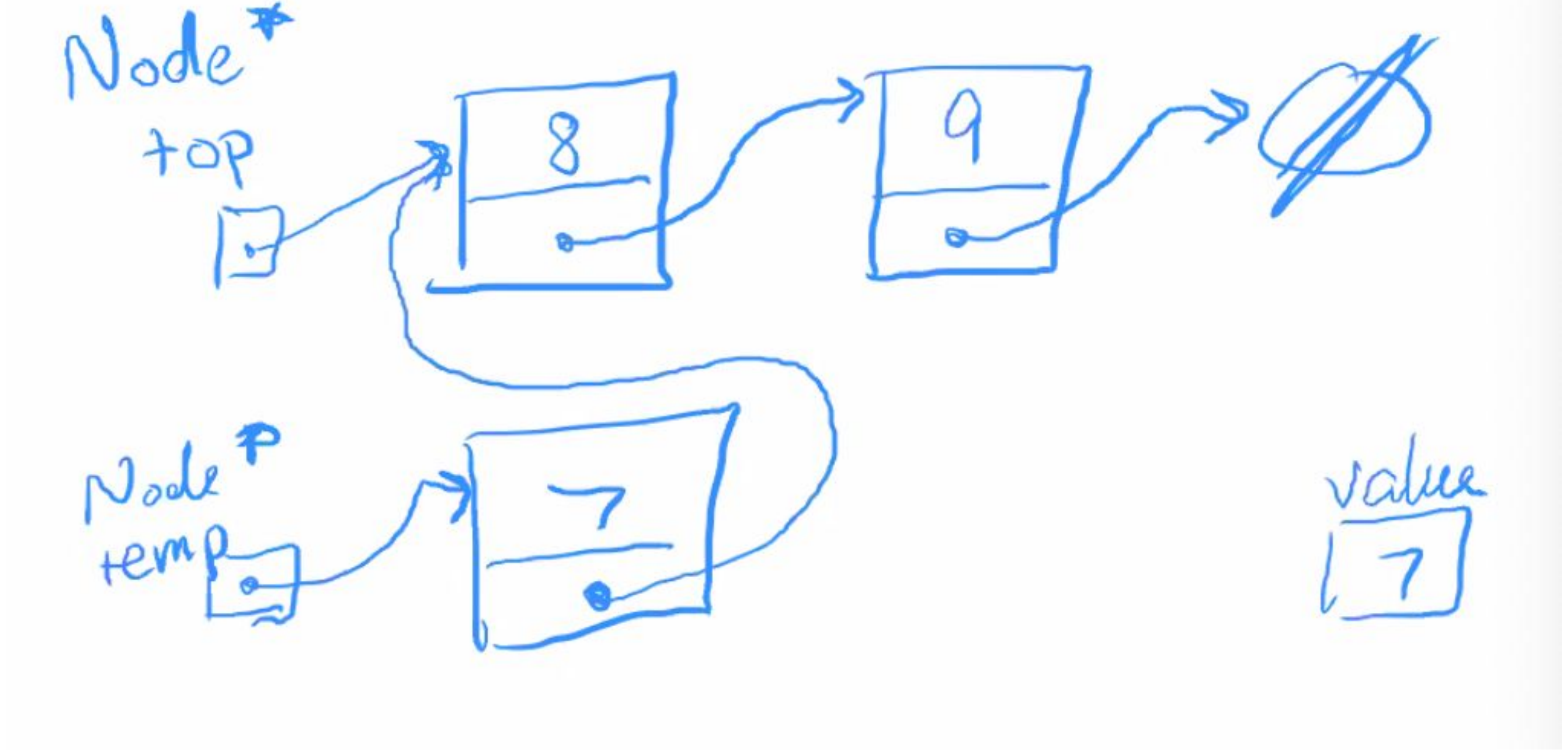




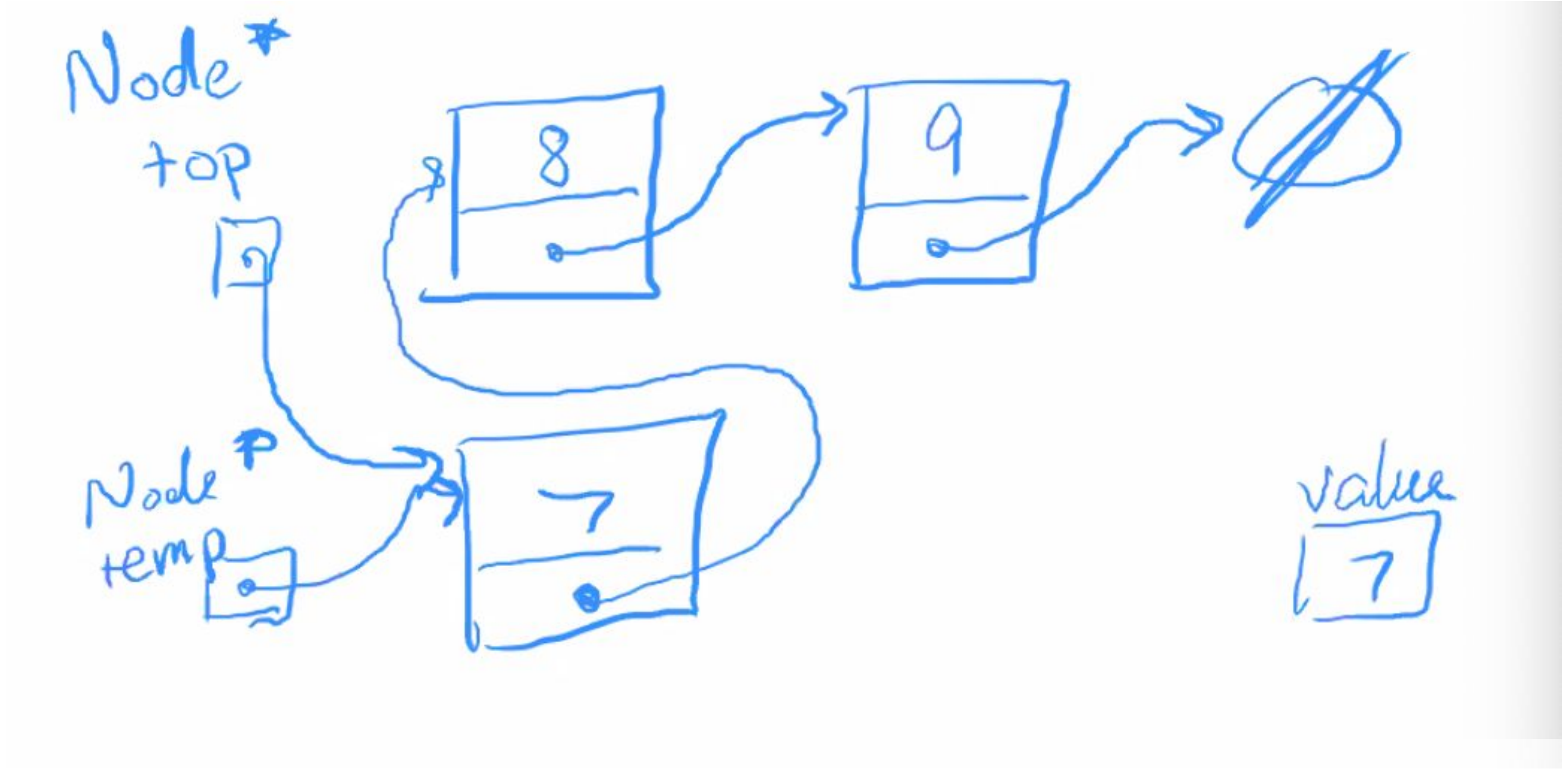
```
Node *temp = new Node;
temp->data = 7;
```



```
Node *temp = new Node;
temp->data = 7;
top = temp; // INCORRECT
```



```
Node *temp = new Node;
temp->data = 7;
temp->next = top;
```



```
Node *temp = new Node;
temp->data = 7;
temp->next = top;
top = temp;
```

Three Stack operations

- **`push()`**
- **`pop()`**
- Destructor

Common linked lists operations

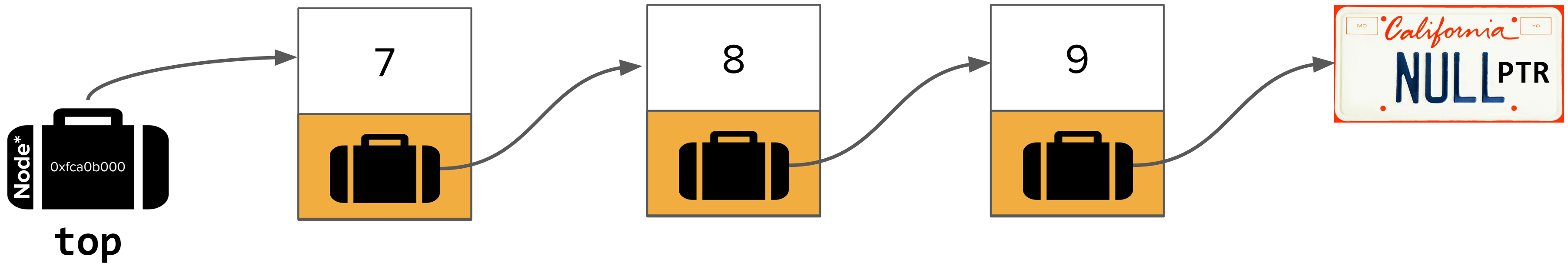
- **Traversal**
  - How do we walk through all elements in the linked list?
- **Rewiring**
  - How do we rearrange the elements in a linked list?
- **Insertion**
  - How do we add an element to a linked list?
- **Deletion**
  - How do we remove an element from a linked list?

pop()

- Now we want to remove the top value:

```
...
myStack.pop(); // we want the result to be {9, 8}
```

Starting state of the list:

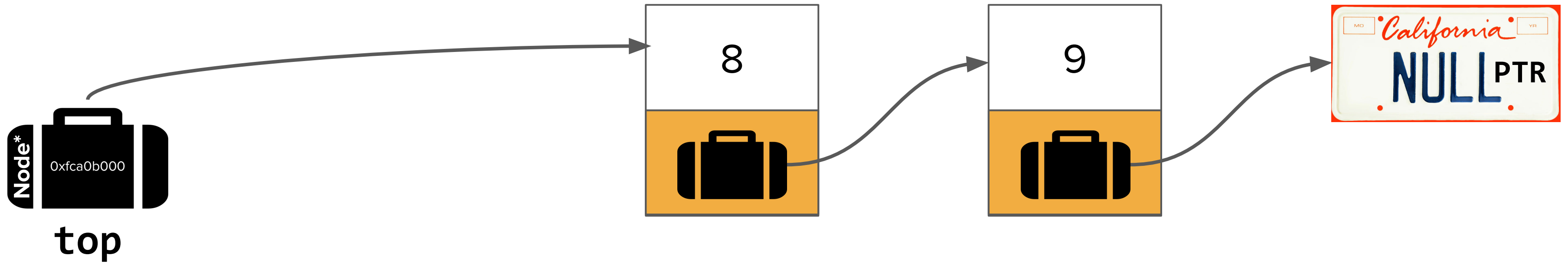


pop()

- Now we want to remove the top value:

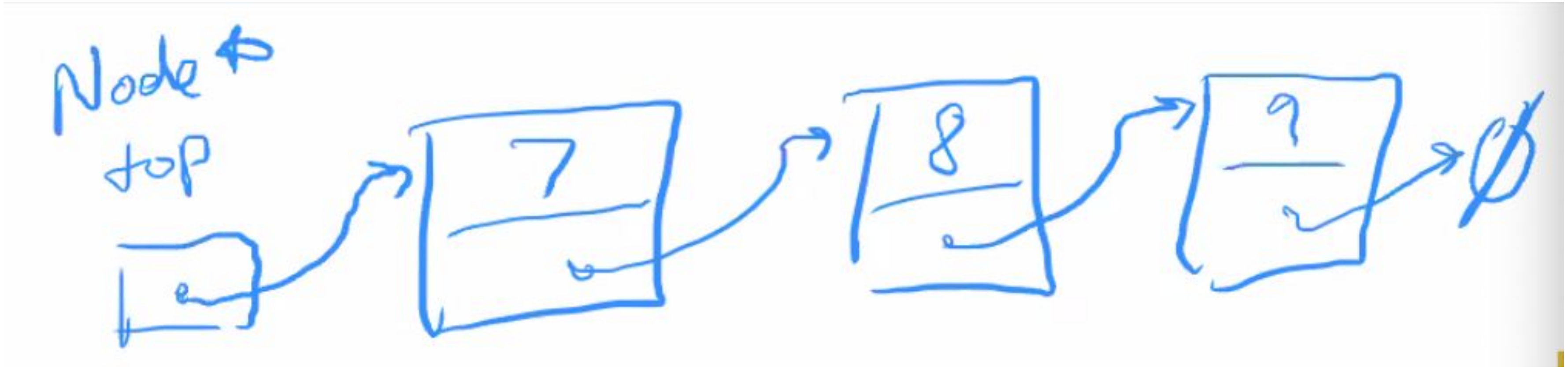
```
...
myStack.pop(); // we want the result to be {9, 8}
```

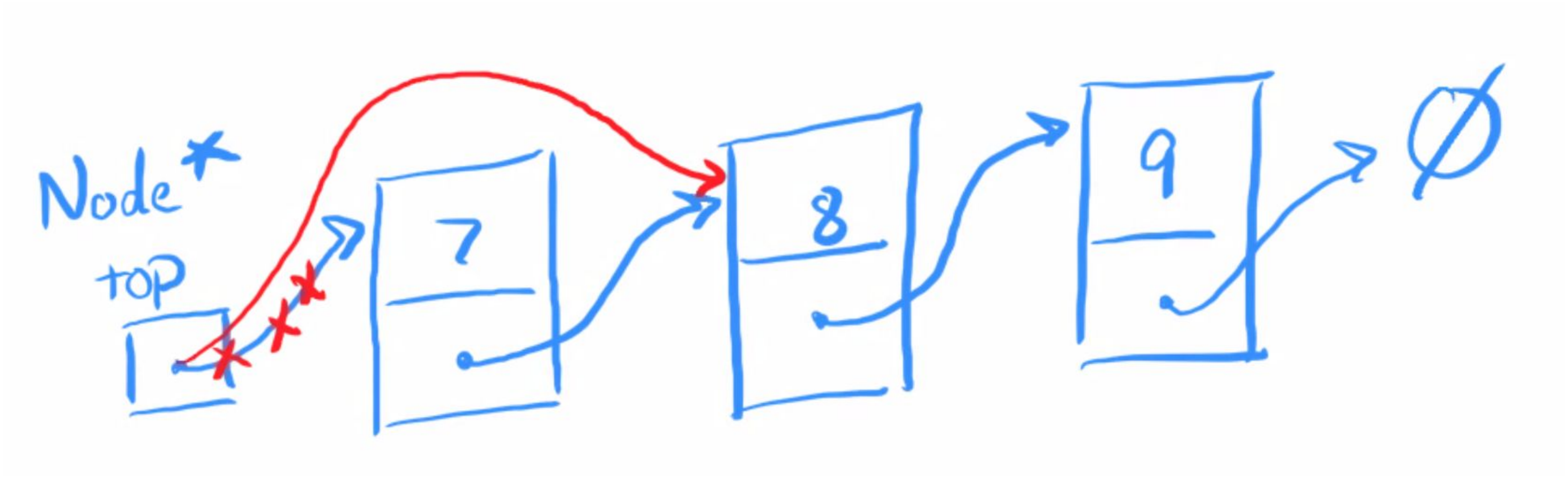
Goal:



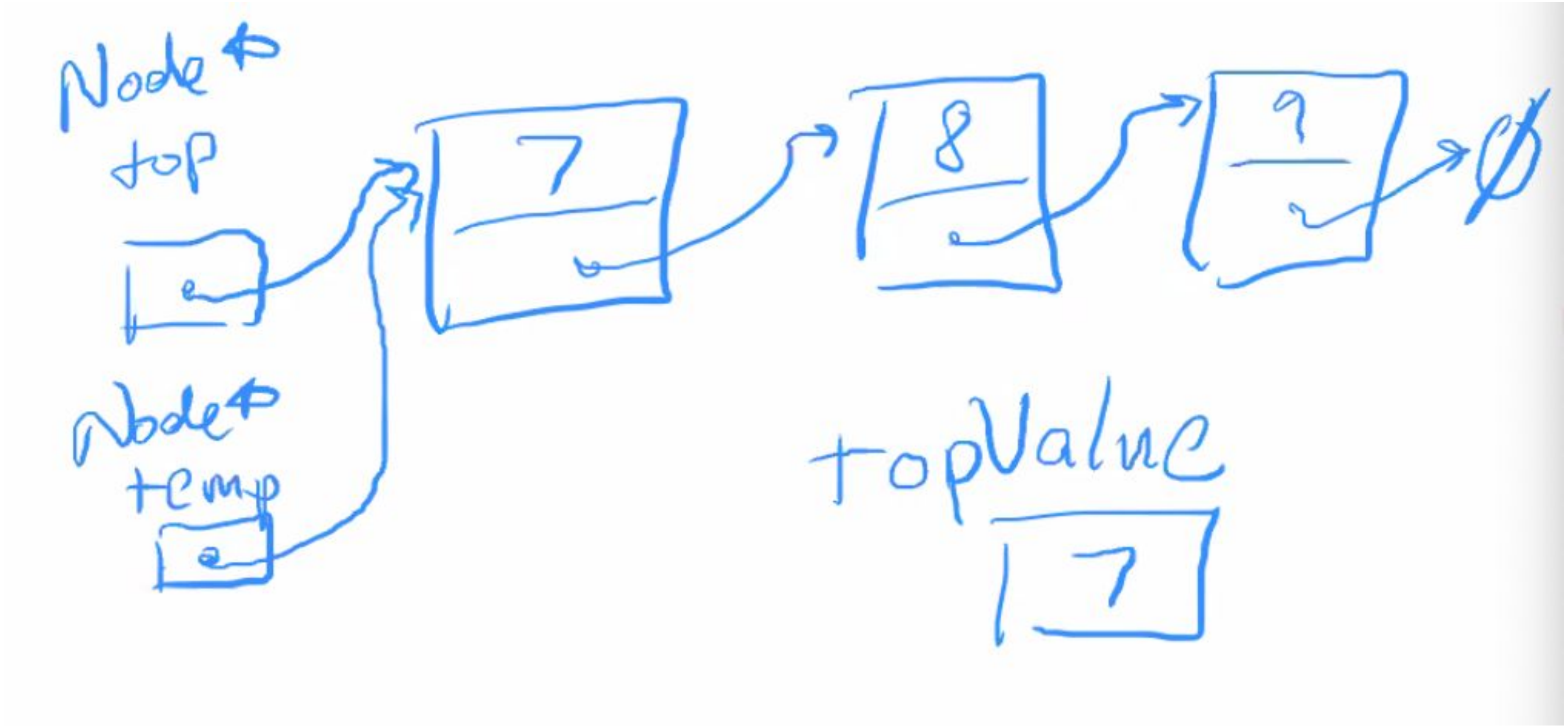
Let’s code **`pop()`**!

Initial State (beginning of **pop( )** function)

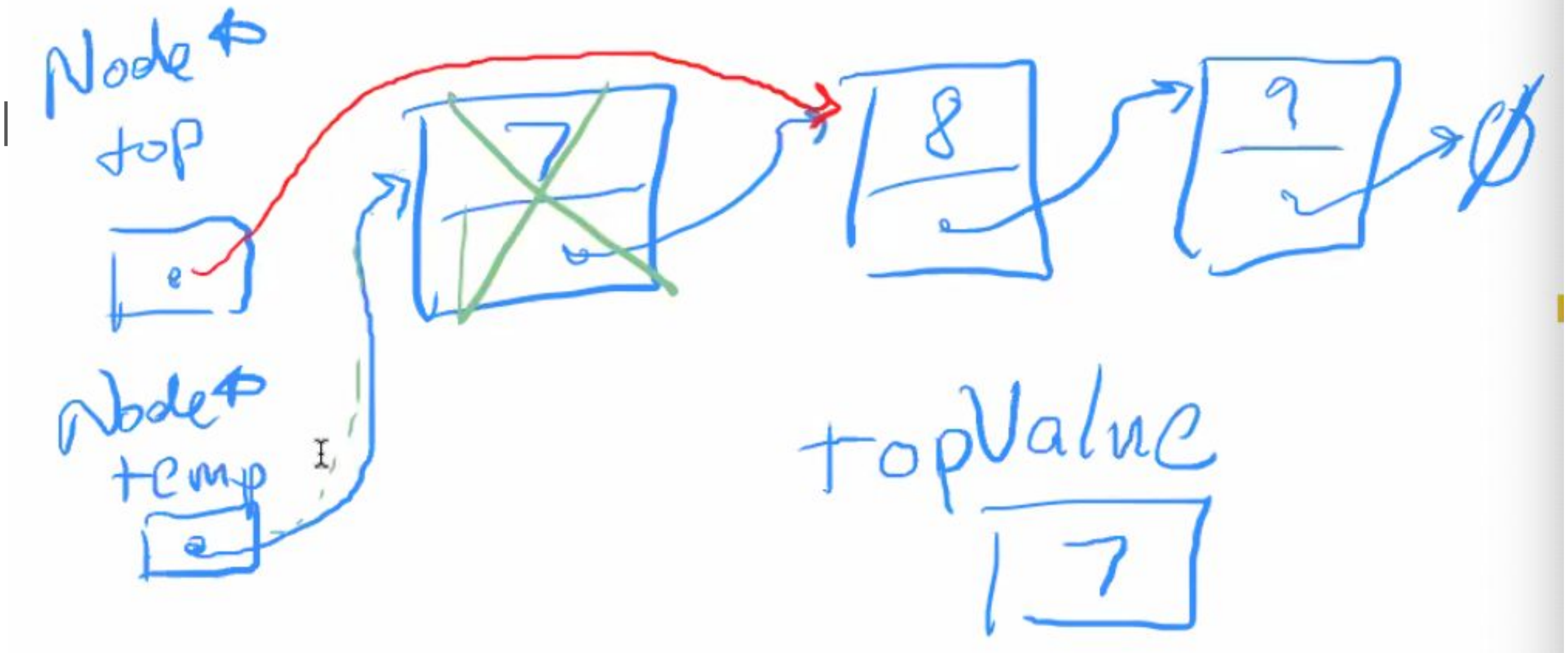




```
top = top->next; // INCORRECT
```



```
Node* temp = top;
```



```
Node* temp = top;
top = top->next;
delete temp;
```

Attendance ticket:

https://tinyurl.com/willthiscodework

Please don't send this link to students who are not here. It's on your honor!

Three Stack operations

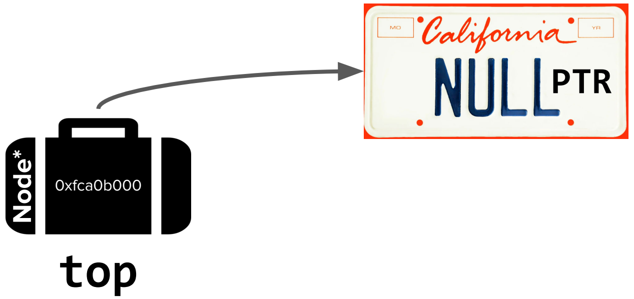
- `push()`
- `pop()`
- **Destructor**

Common linked lists operations

- **Traversal**
  - How do we walk through all elements in the linked list?
- **Rewiring**
  - How do we rearrange the elements in a linked list?
- **Insertion**
  - How do we add an element to a linked list?
- **Deletion**
  - How do we remove an element from a linked list?

Destructor

- We have to make sure we delete all of the **`Node`**s.
- The **`top`** pointer should be **`nullptr`** when we're done.



Let’s code the destructor!

IntStack takeaways

- Linked lists are chains of Node structs, which are connected by pointers.
    - Since the memory is not contiguous, they allow for fast rewiring between nodes (without moving all the other Nodes like an array might).
- Common traversal strategy
    - While loop with a pointer that starts at the front of your list
    - Inside the while loop, reassign the pointer to the next node
- Common bugs
    - Be careful about the order in which you delete and rewire pointers!
    - It’s easy to end up with dangling pointers or memory leaks (memory that hasn’t been deallocated but that you not longer have a pointer to)

How do we manipulate linked lists?

Linked list utility functions

- We’ve now seen linked lists in the context of classes, where we used a linked list as the data storage underlying an implementation of a Stack.

Linked list utility functions

- We’ve now seen linked lists in the context of classes, where we used a linked list as the data storage underlying an implementation of a Stack.

- However, linked lists are not limited only to use within classes. In fact, the next assignment will ask you to implement "standalone" linked list functions that operate on provided linked lists, outside the context of a class.

Linked list utility functions

- We’ve now seen linked lists in the context of classes, where we used a linked list as the data storage underlying an implementation of a Stack.
- However, linked lists are not limited only to use within classes. In fact, the next assignment will ask you to implement "standalone" linked list functions that operate on provided linked lists, outside the context of a class.
- This is the paradigm that we will work under for the several functions. In doing so, we'll gain a little more flexibility to get practice with many different linked list operations and build our linked list toolbox!

Common linked lists operations

- **Traversal**
  - How do we walk through all elements in the linked list?
- **Rewiring**
  - How do we rearrange the elements in a linked list?
- **Insertion**
  - How do we add an element to a linked list?
- **Deletion**
  - How do we remove an element from a linked list?

Linked List Traversal

Traversal utility functions

- Freeing a linked list
- Printing a linked list
- Measuring the length of a list

Traversal utility functions

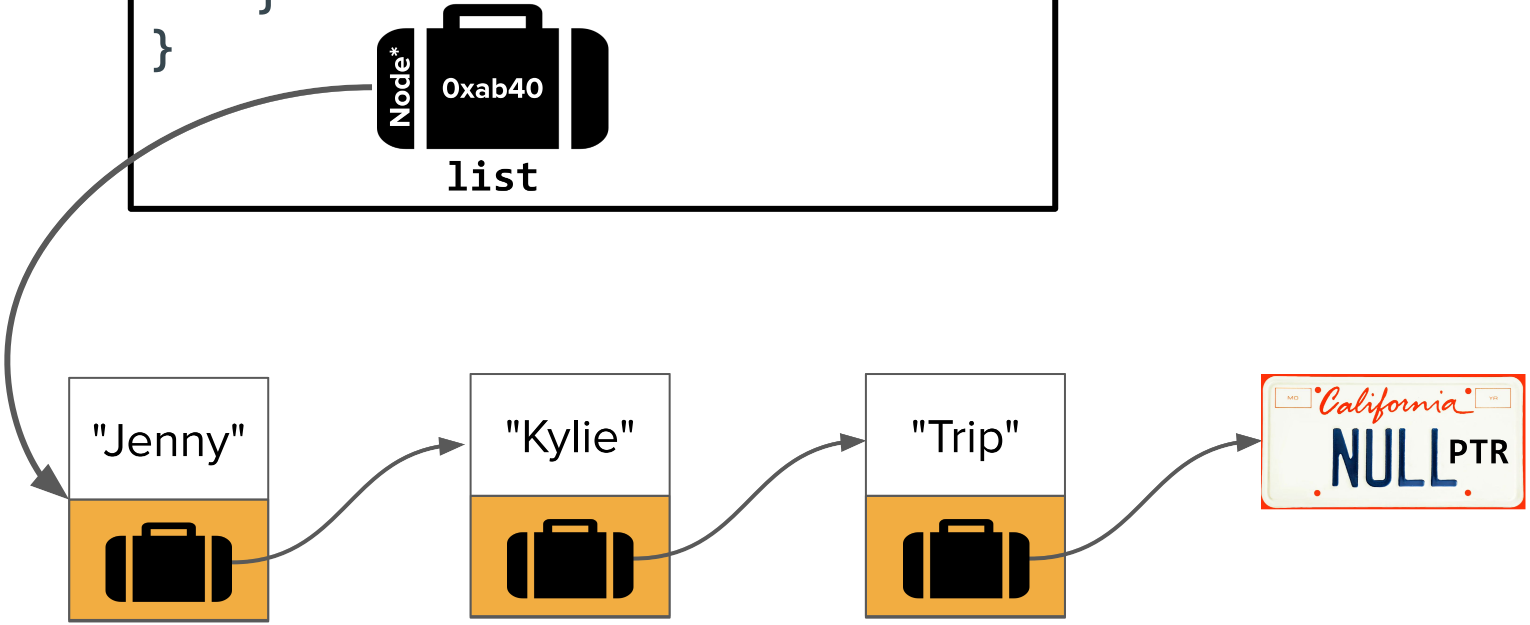
- **Freeing a linked list**
    - Very similar to the destructor we just saw!

- Printing a linked list

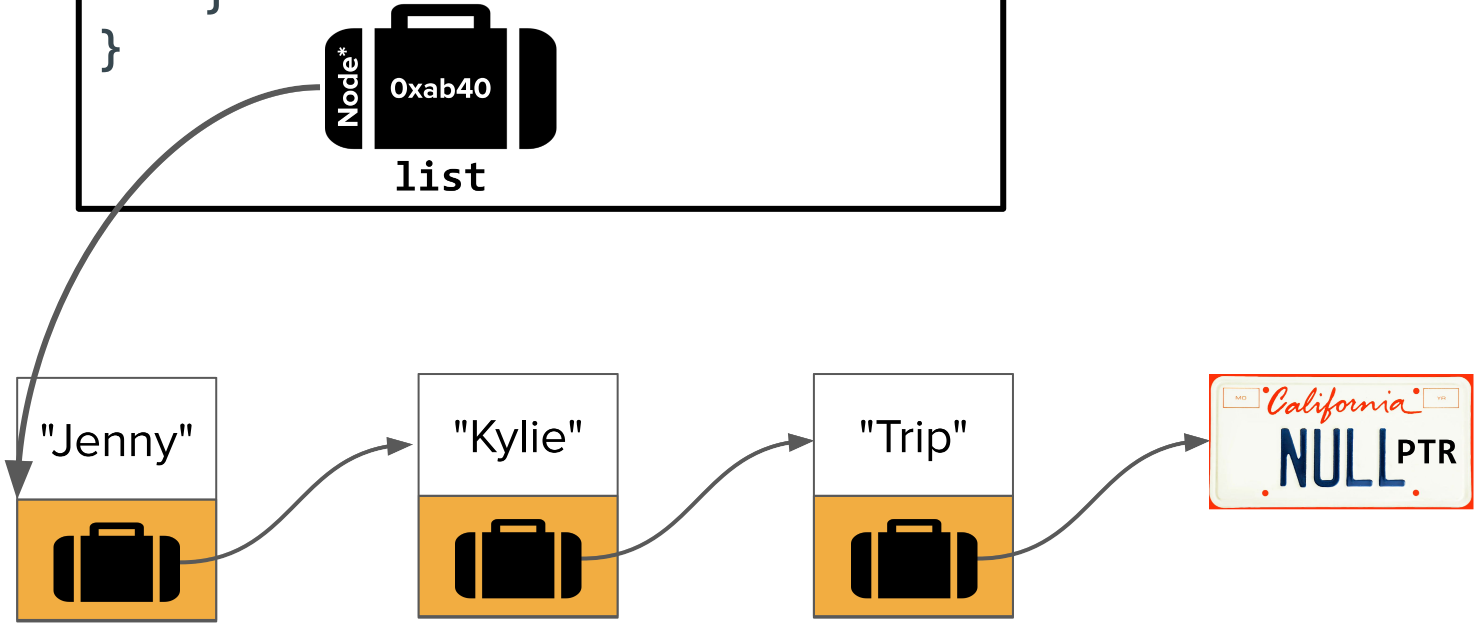
- Measuring the length of a list

Freeing linked lists, the wrong way

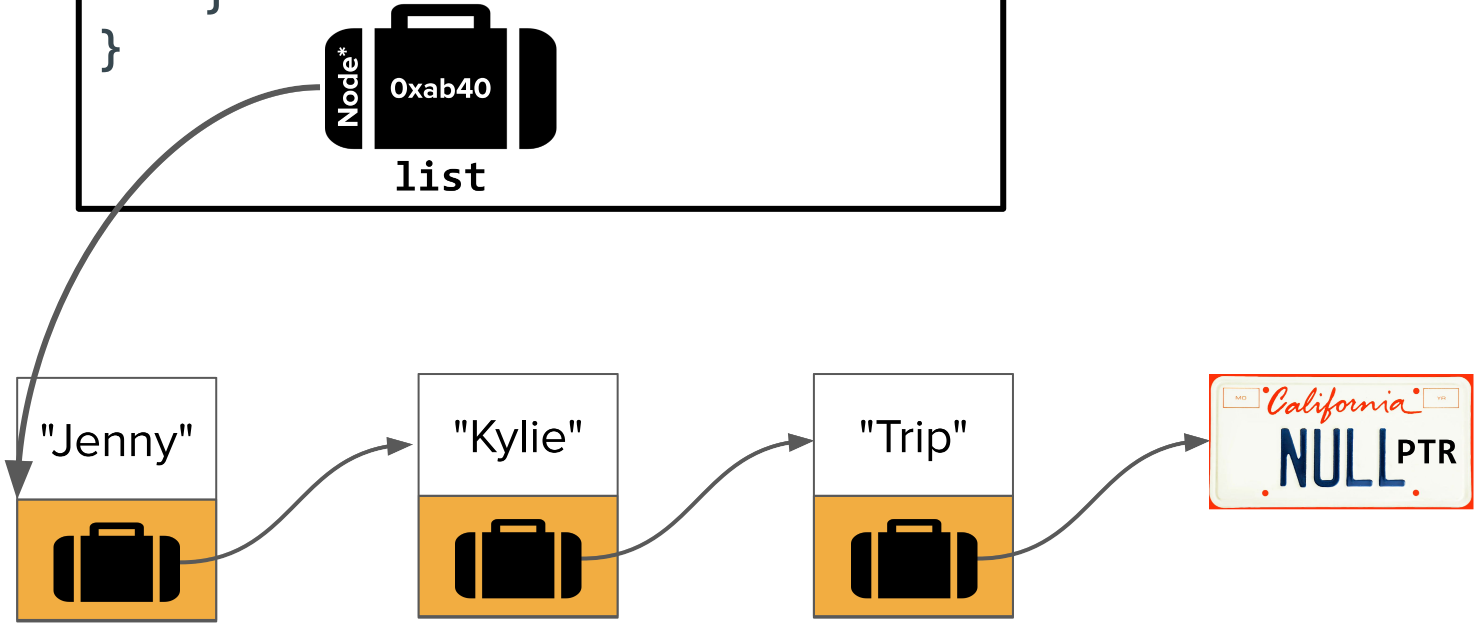
```
void freeList(Node* list) {
    /* WRONG WRONG WRONG WRONG WRONG */
    while (list != nullptr) {
        delete list;
        list = list->next;
    }
}
```



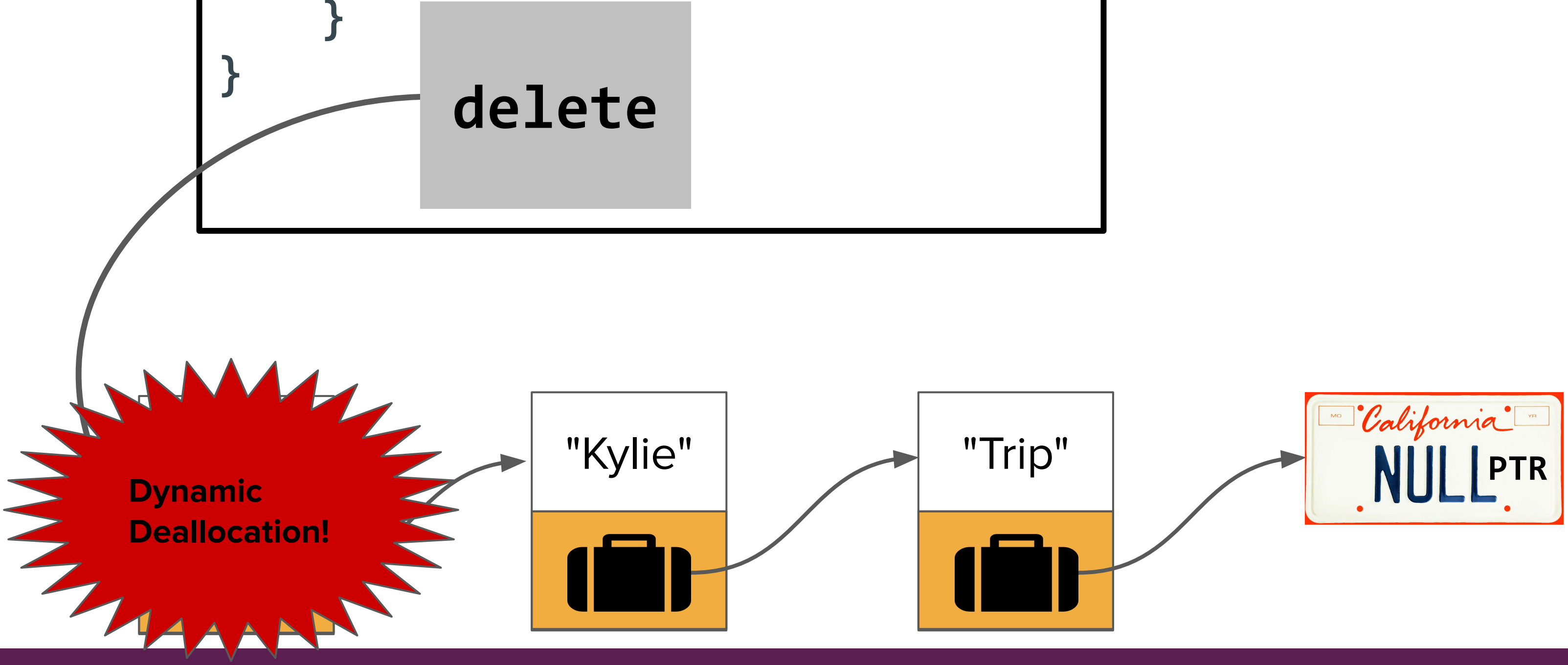
```
void freeList(Node* list) {
    /* WRONG WRONG WRONG WRONG WRONG */
    while (list != nullptr) {
        delete list;
        list = list->next;
    }
}
```



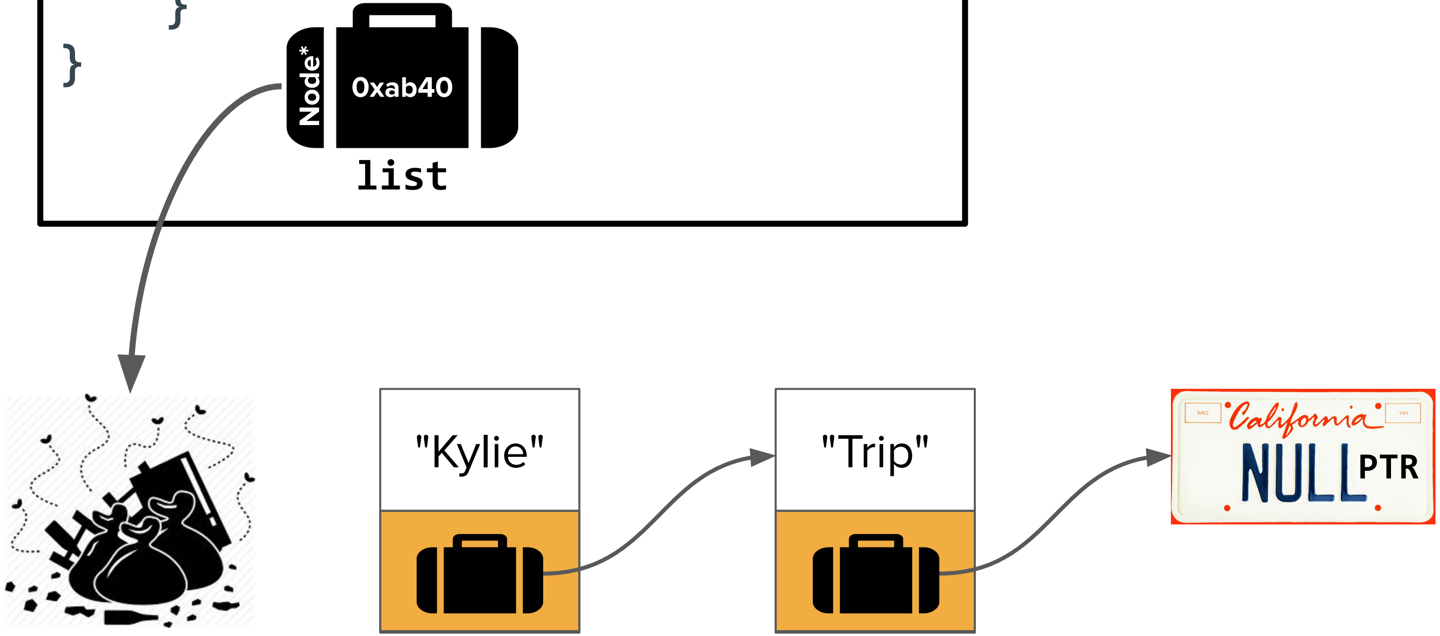
```
void freeList(Node* list) {
    /* WRONG WRONG WRONG WRONG WRONG */
    while (list != nullptr) {
        delete list;
        list = list->next;
    }
}
```



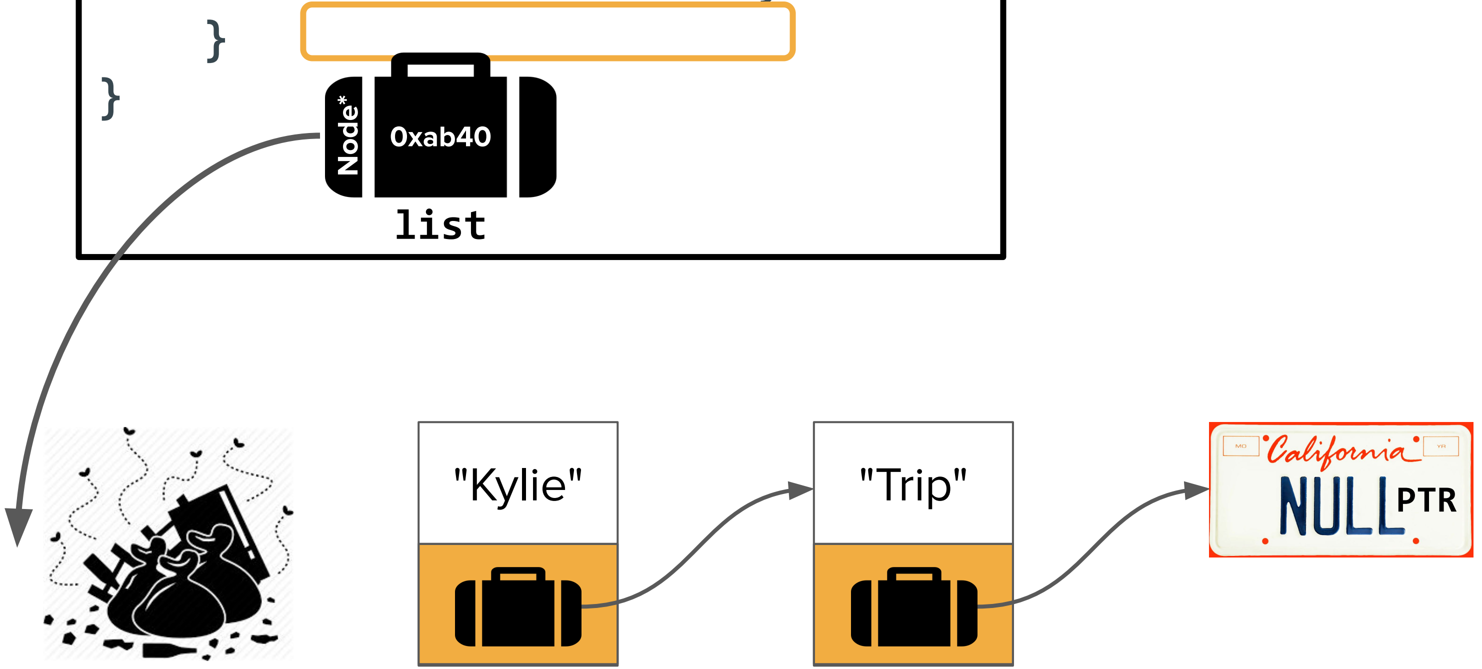
```
void freeList(Node* list) {
    /* WRONG WRONG WRONG WRONG WRONG */
    while (list != nullptr) {
        delete list;
        list = list->next;
    }
}
```



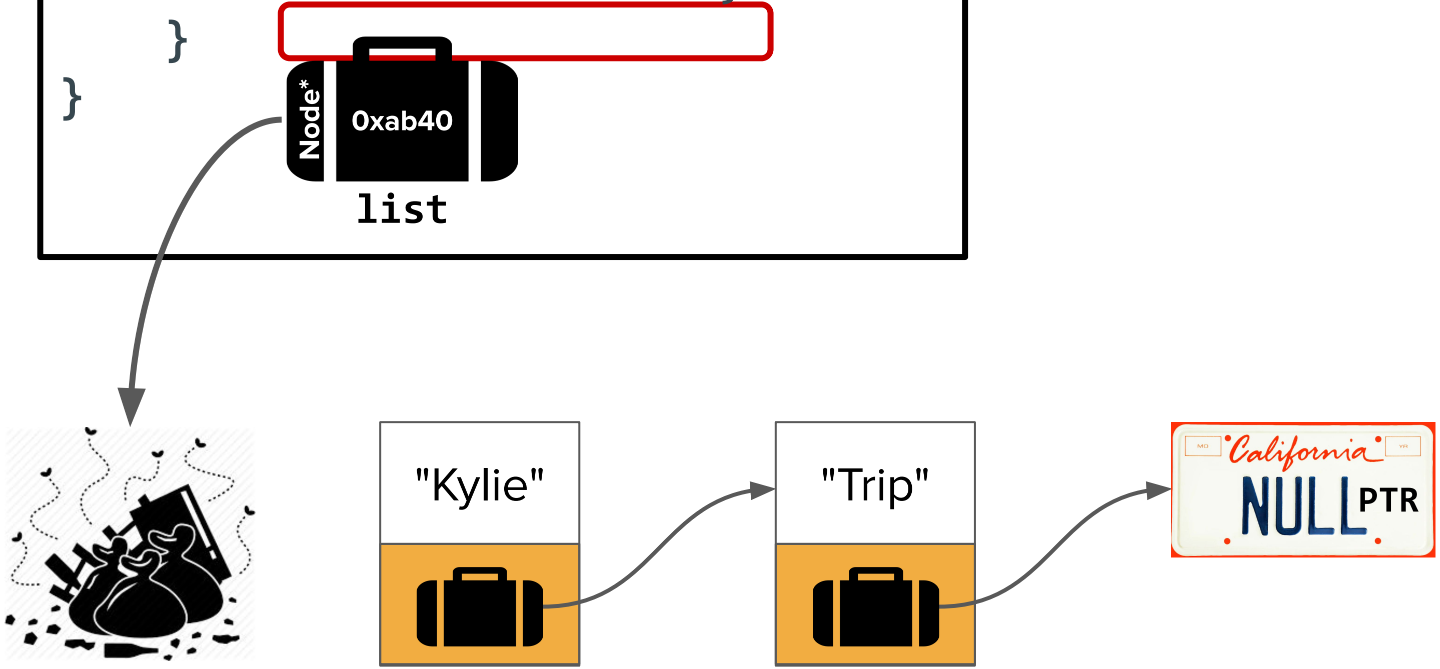
```
void freeList(Node* list) {
    /* WRONG WRONG WRONG WRONG WRONG */
    while (list != nullptr) {
        delete list;
        list = list->next;
    }
}
```



```
void freeList(Node* list) {
    /* WRONG WRONG WRONG WRONG WRONG */
    while (list != nullptr) {
        delete list;
        list = list->next;
    }
}
```

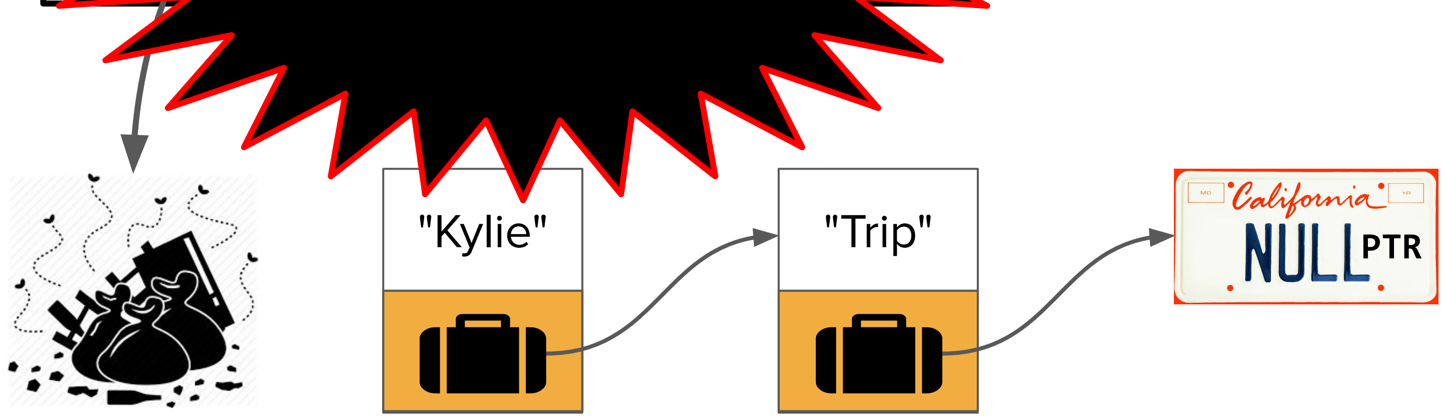


```
void freeList(Node* list) {
    /* WRONG WRONG WRONG WRONG WRONG */
    while (list != nullptr) {
        delete list;
        list = list->next;
    }
}
```



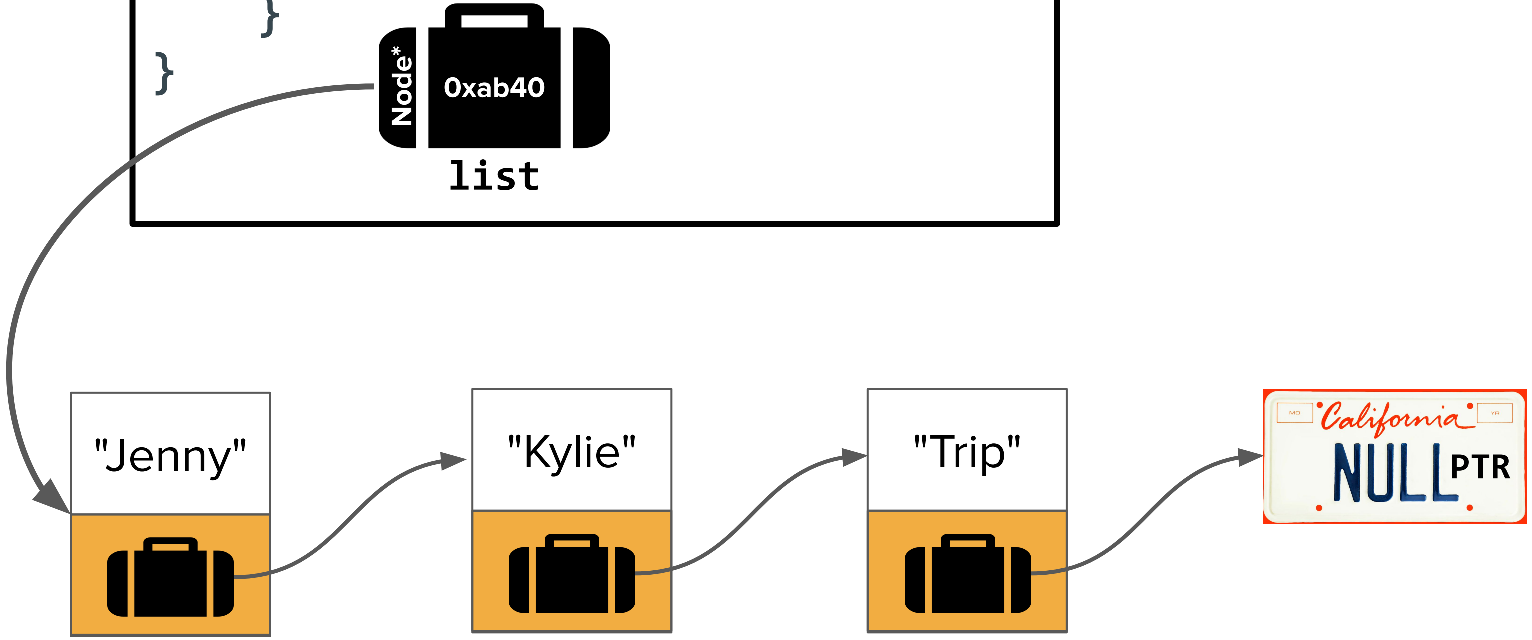
```
void freeList(Node* list) {
    /* WRONG WRONG WRONG WRONG WRONG */
    while (list != nullptr) {
}
```

Undefined Behavior!

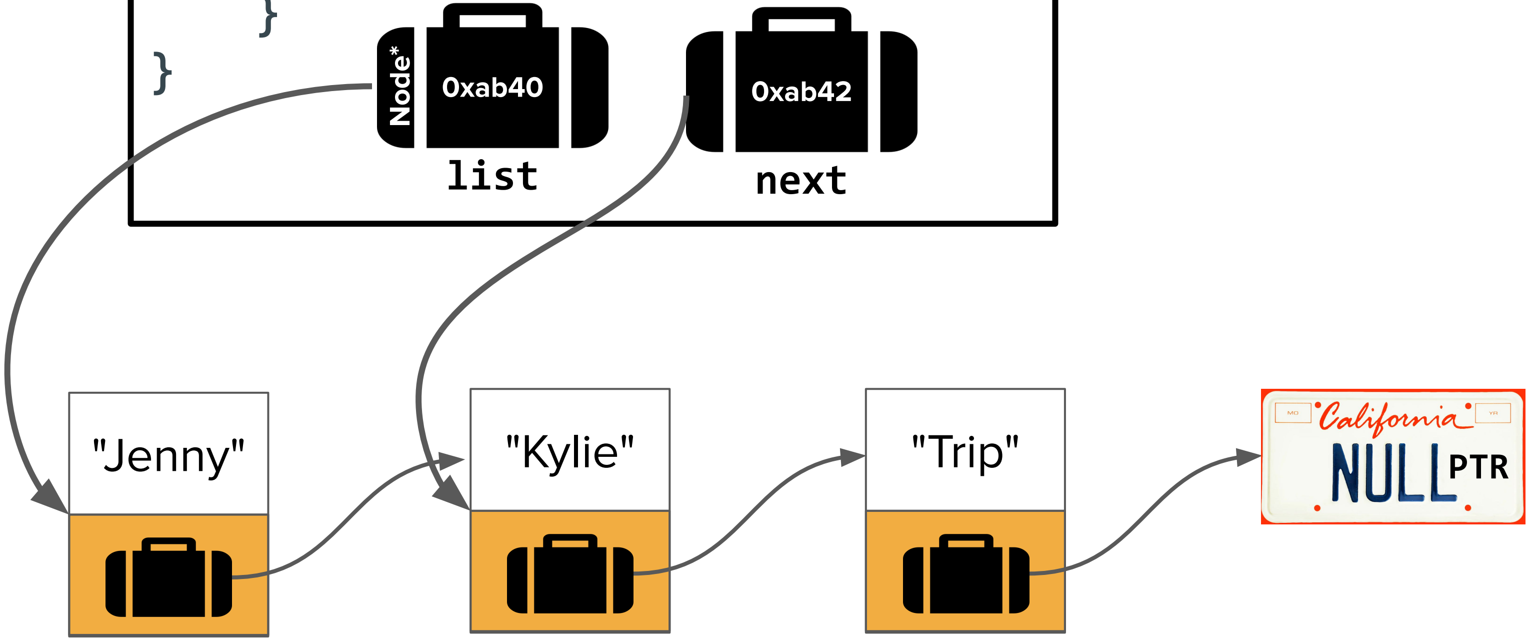


Freeing linked lists, the right way (intuition)

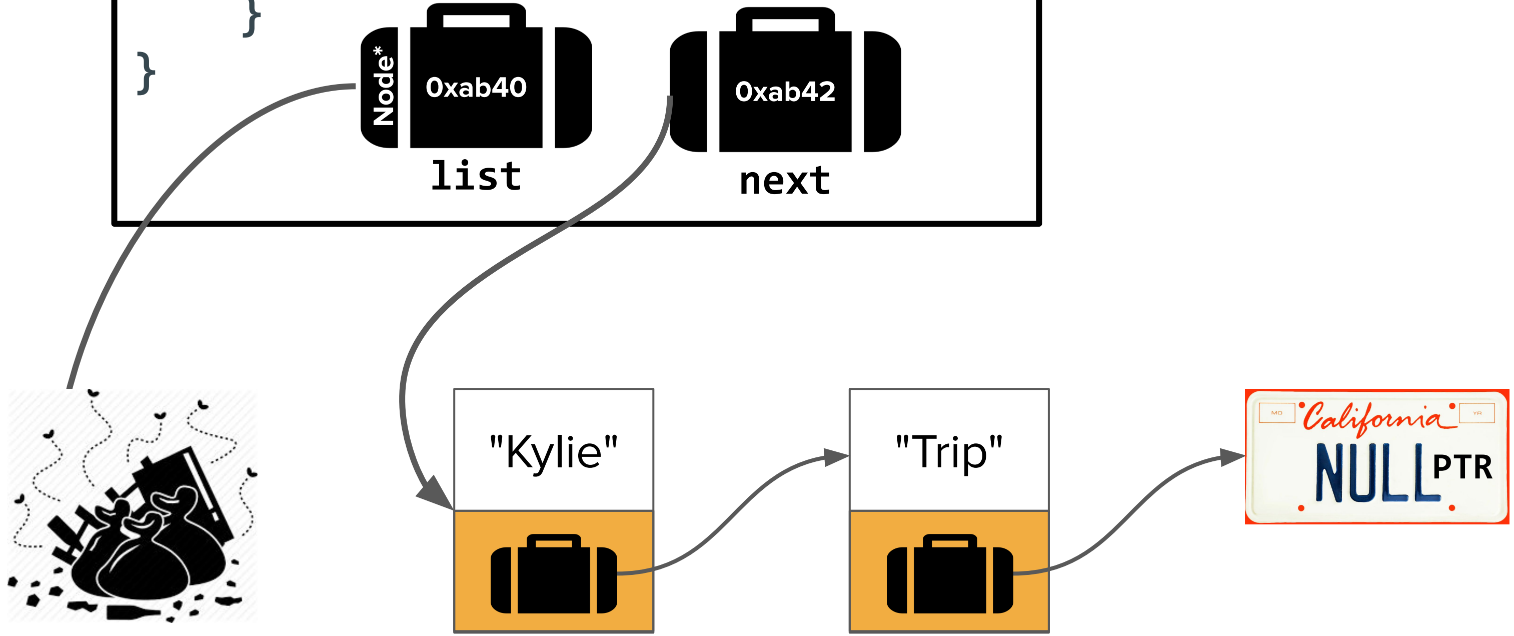
```
void freeList(Node* list) {
    while (list != nullptr) {

        delete list;
        list = list->next;
    }
}
```



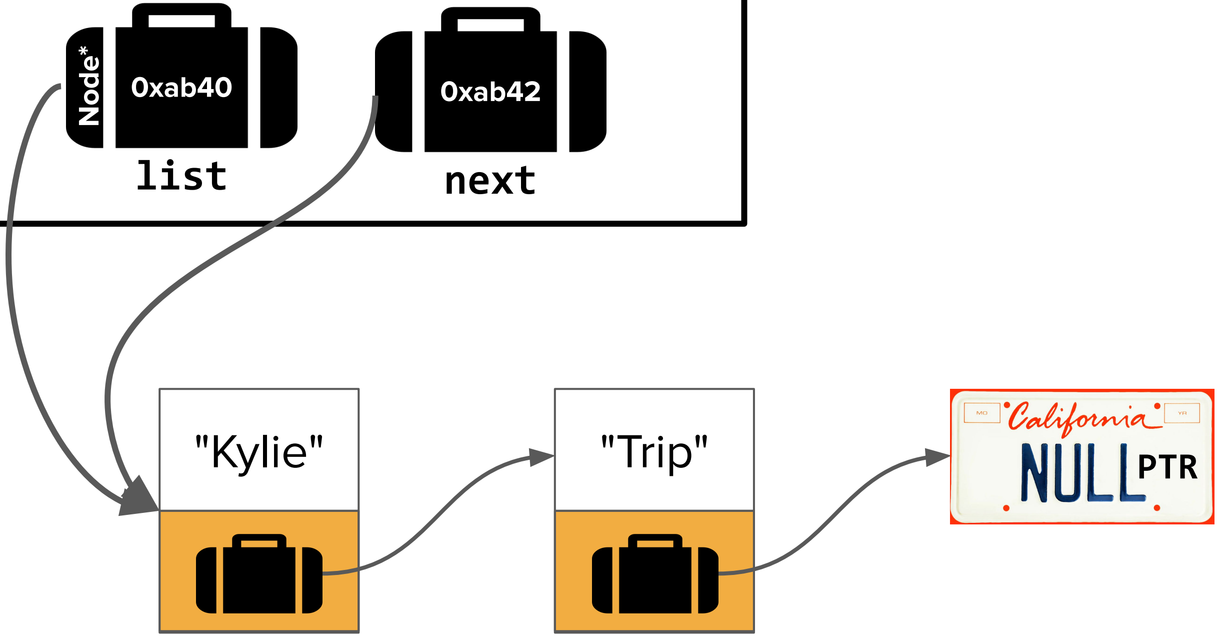
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```

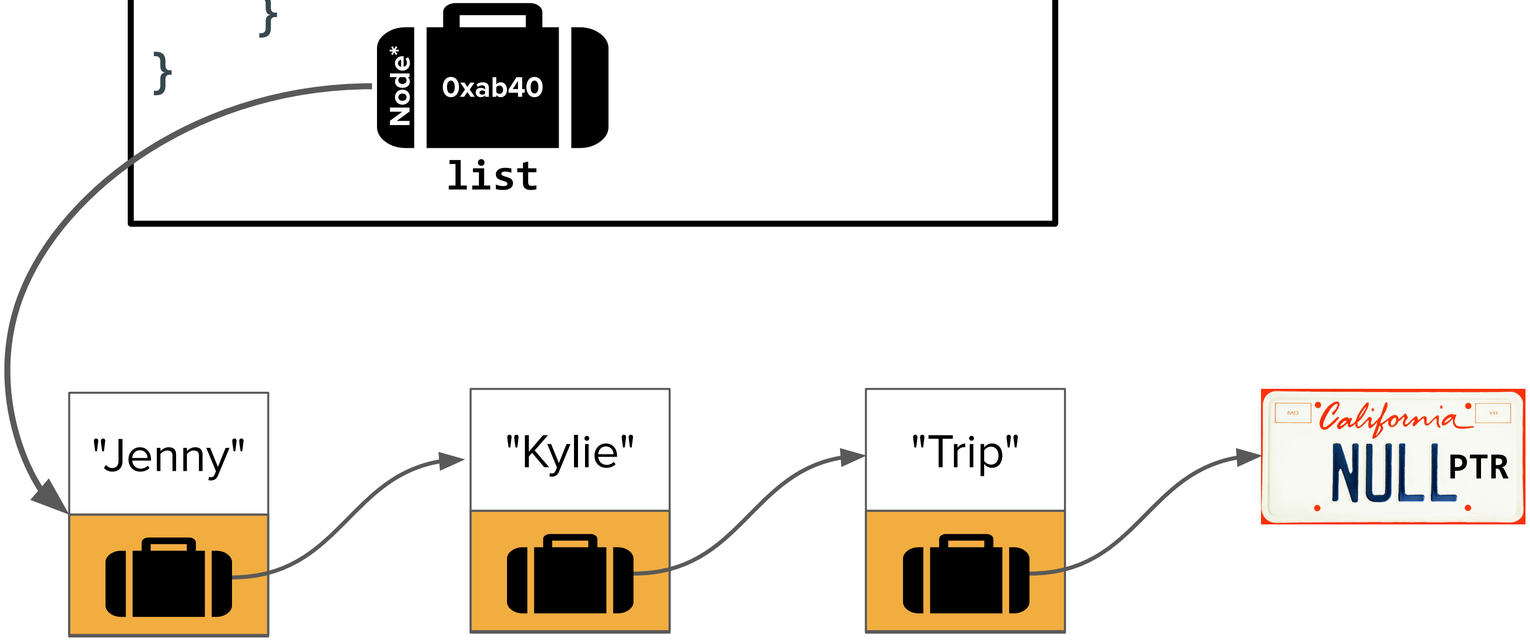


```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```

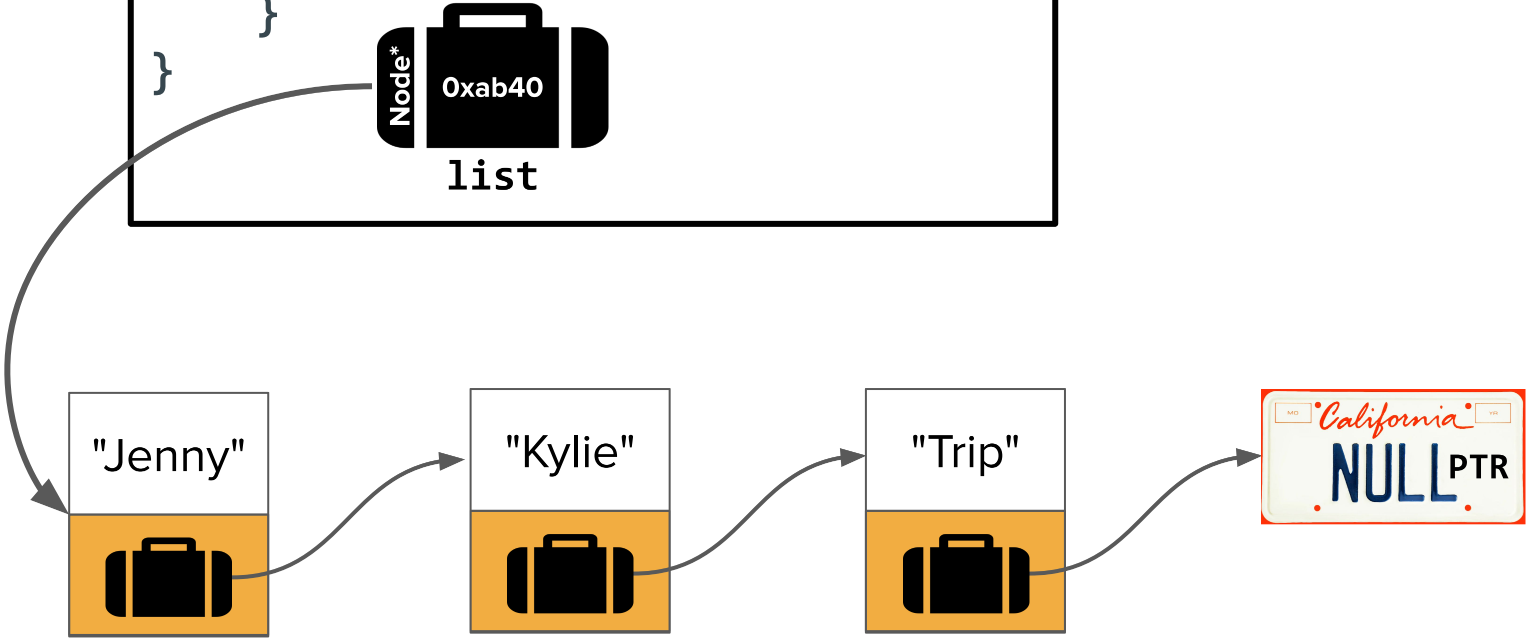


Freeing linked lists, the right way from the top

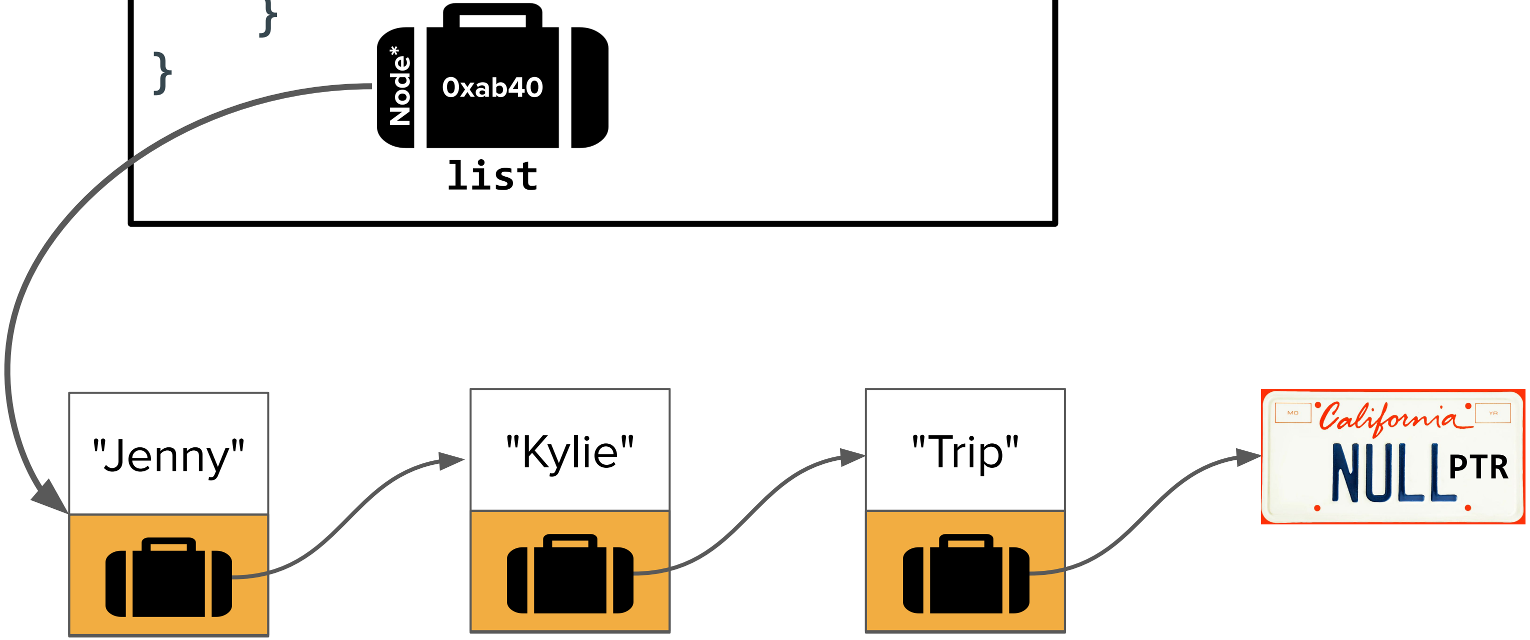
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



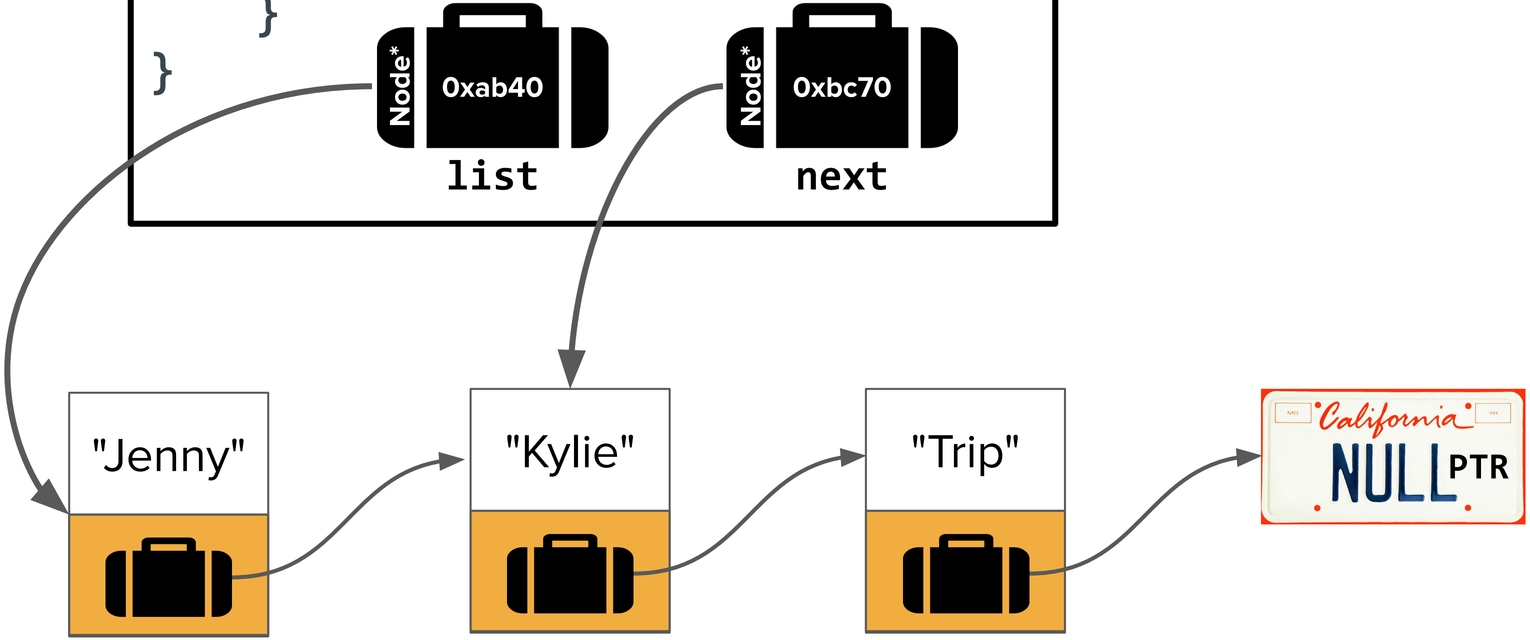
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



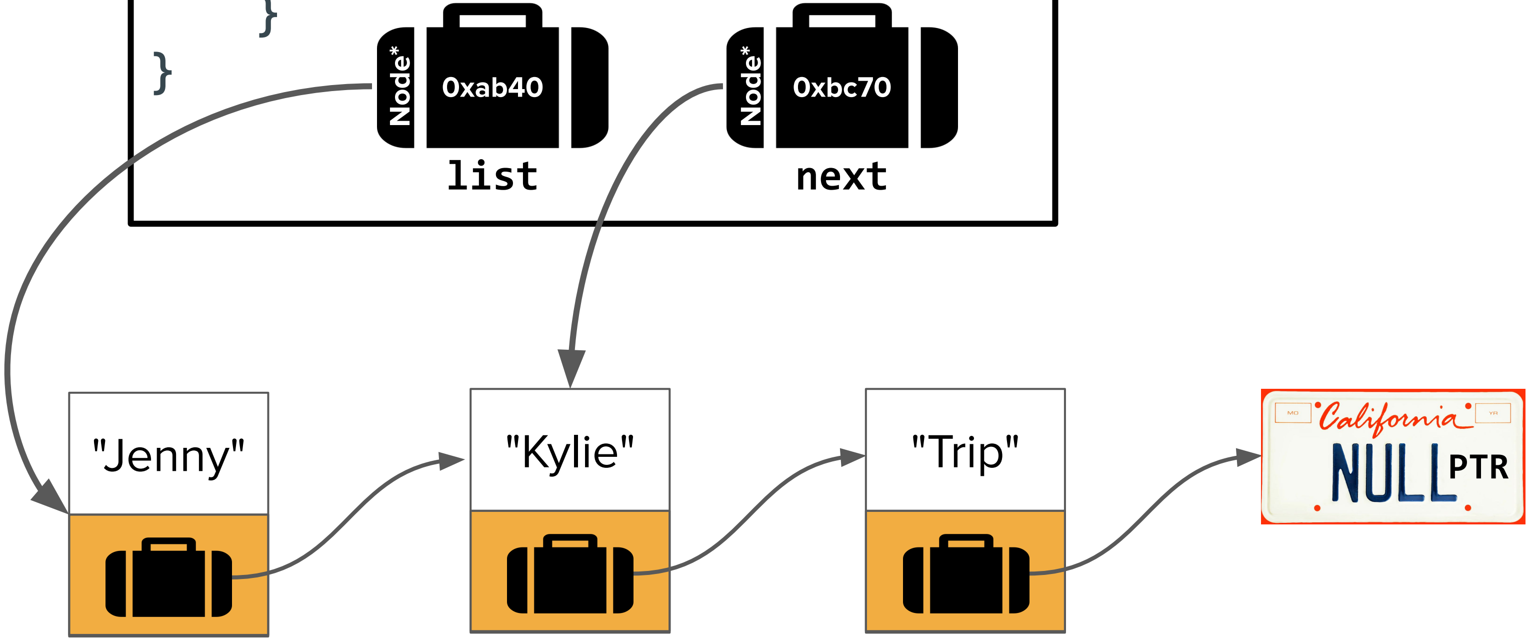
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



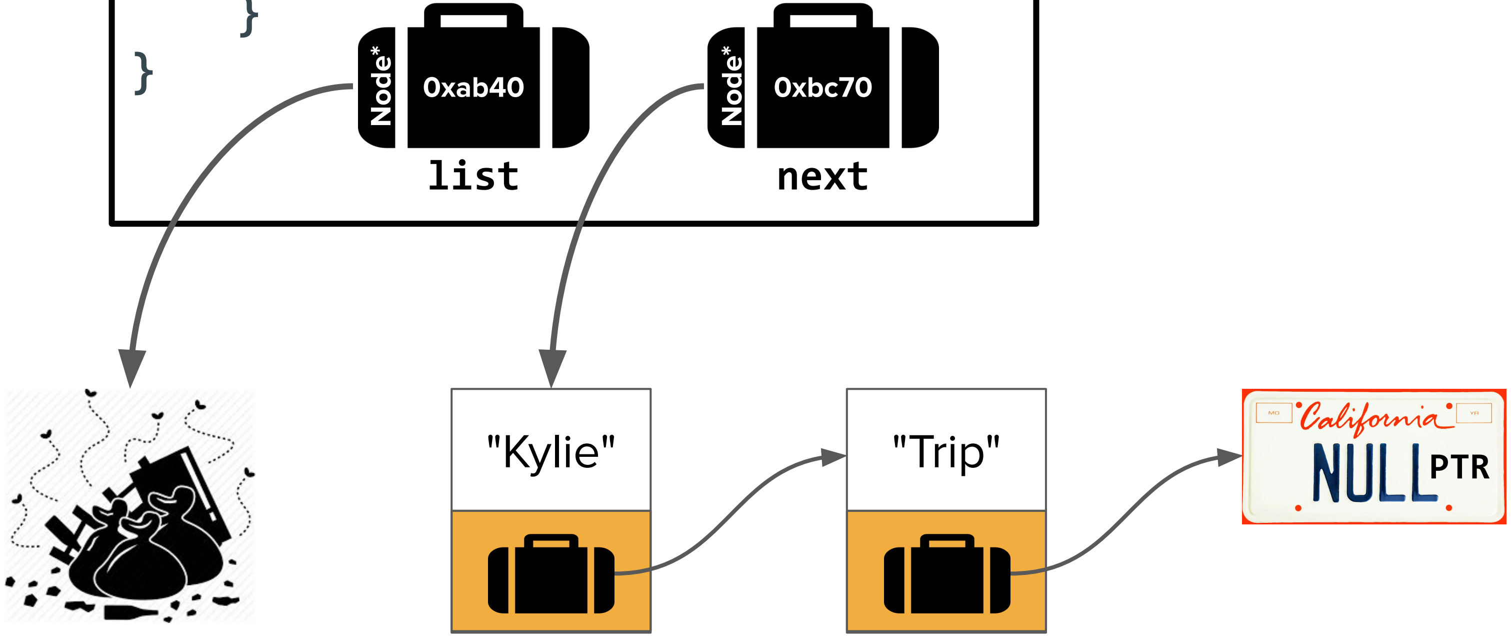
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



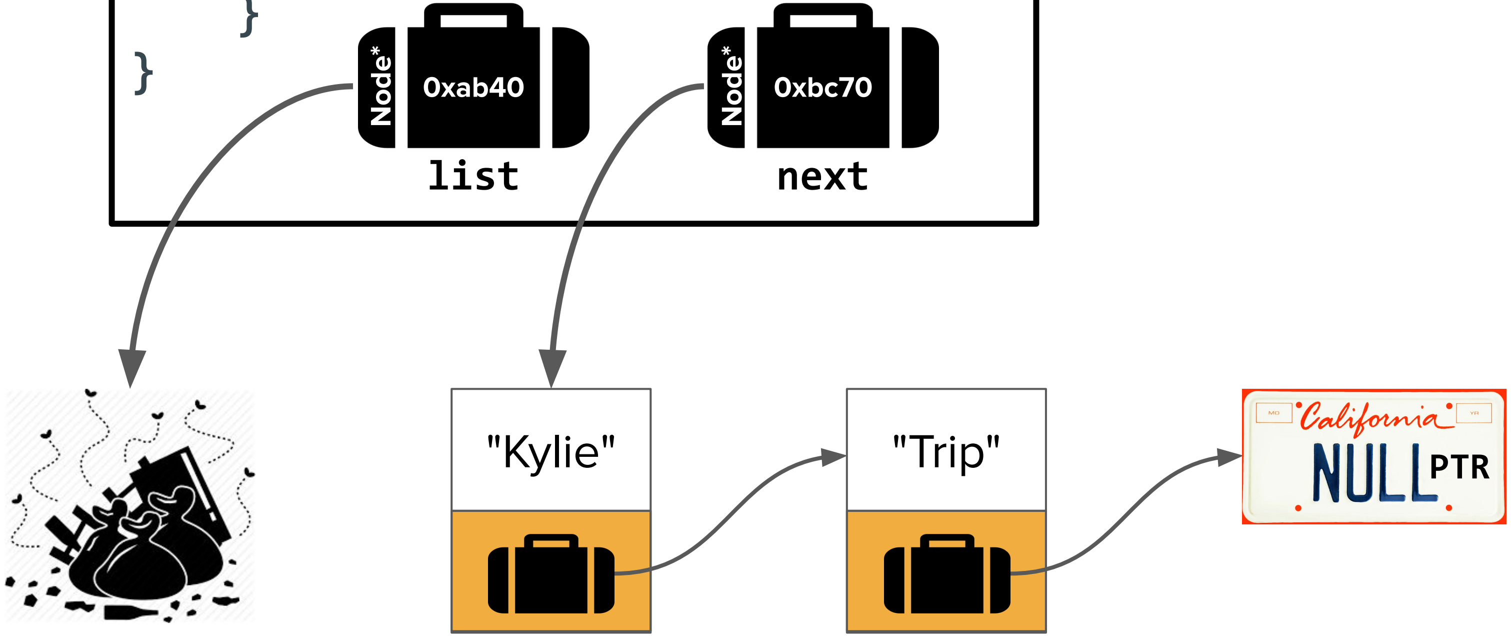
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



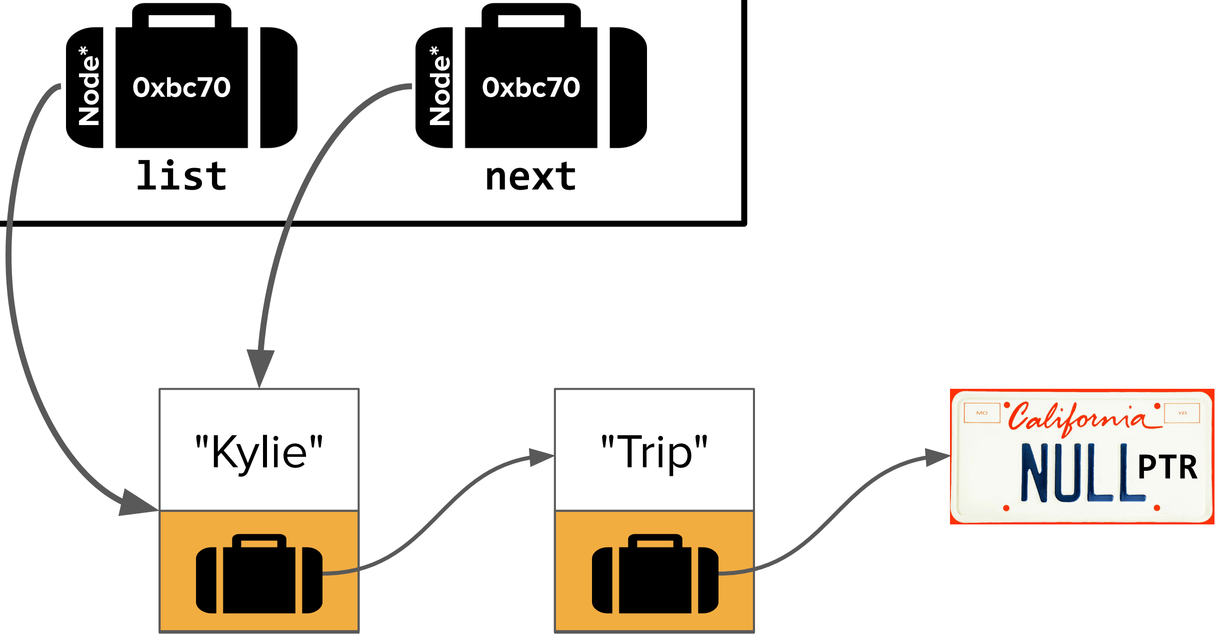
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



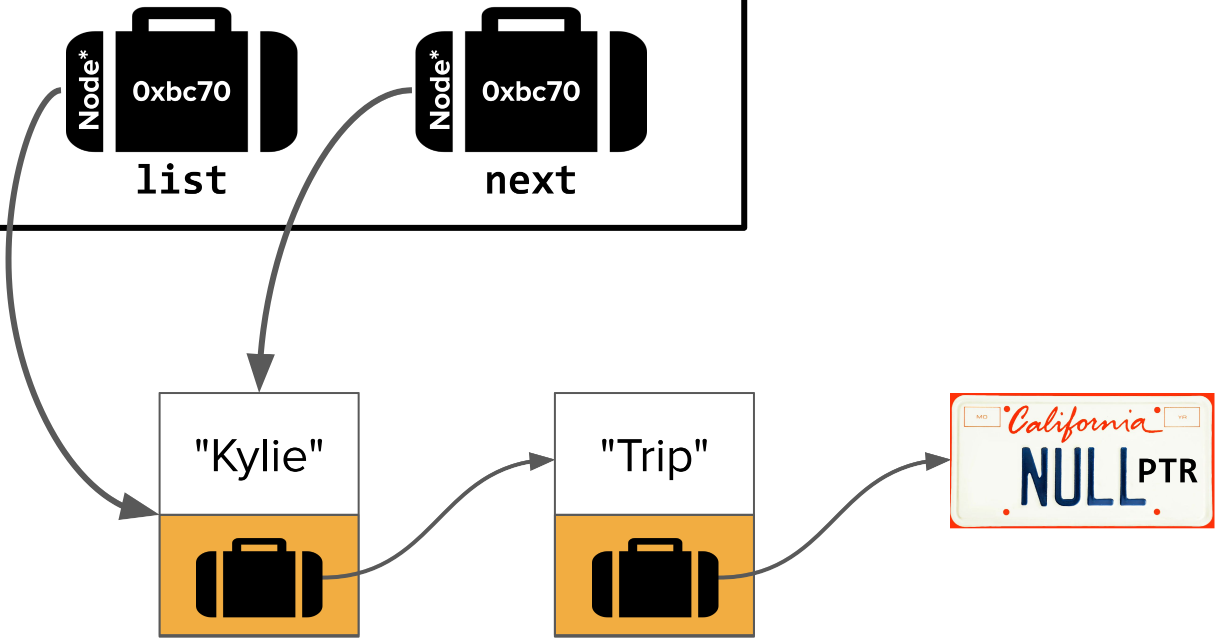
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



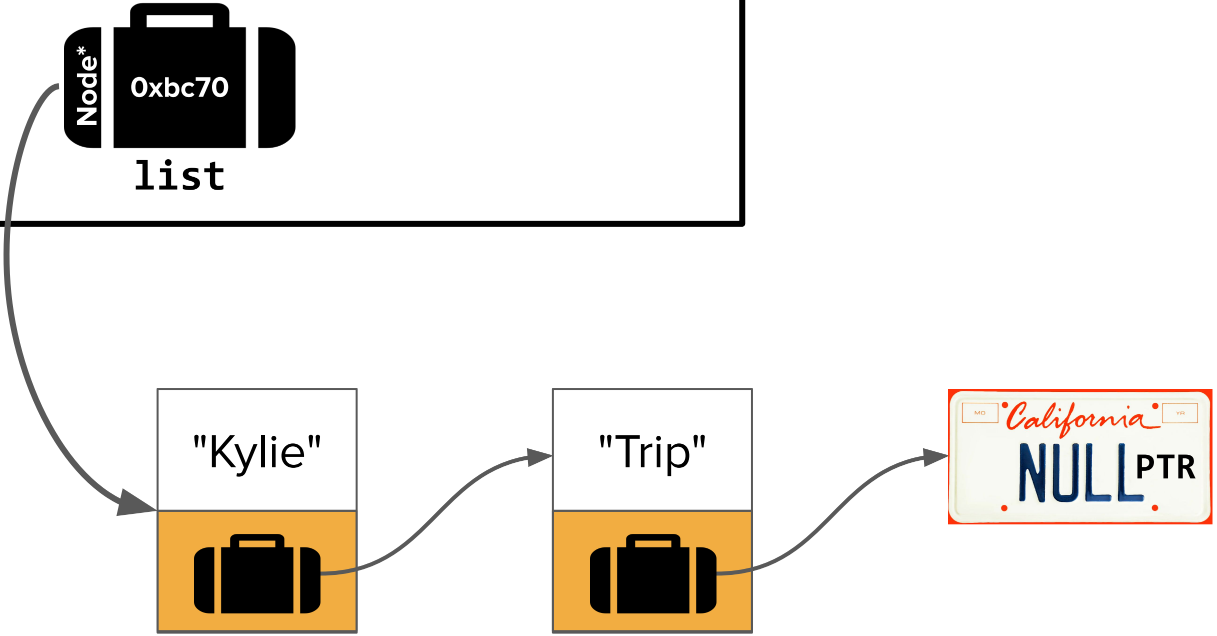
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



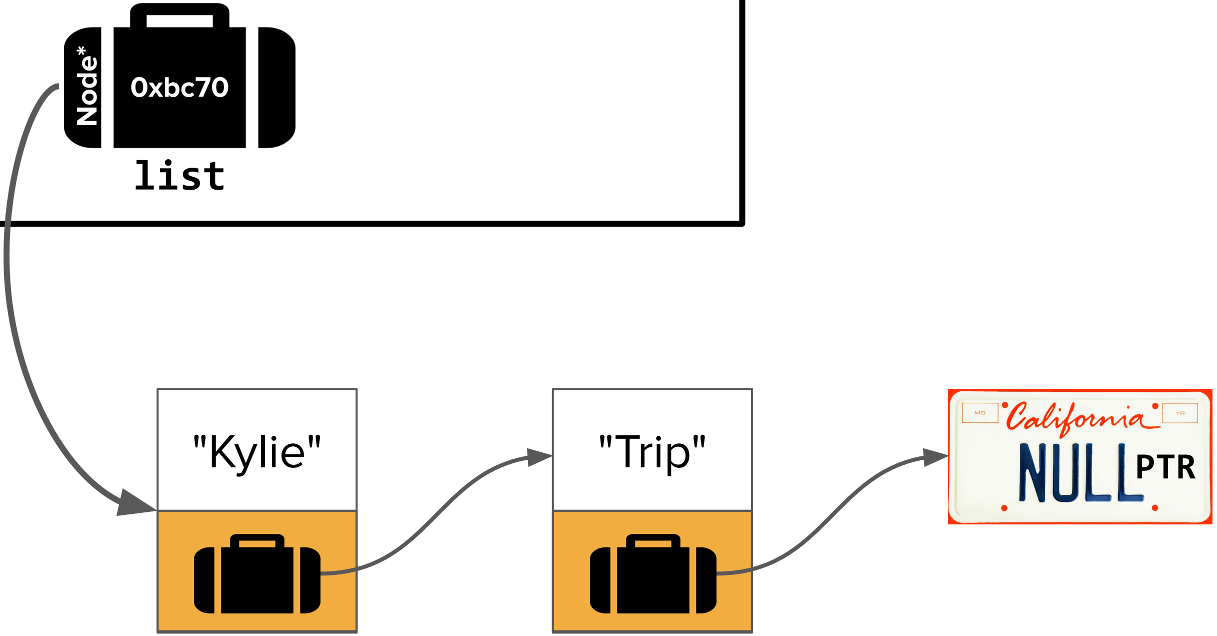
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



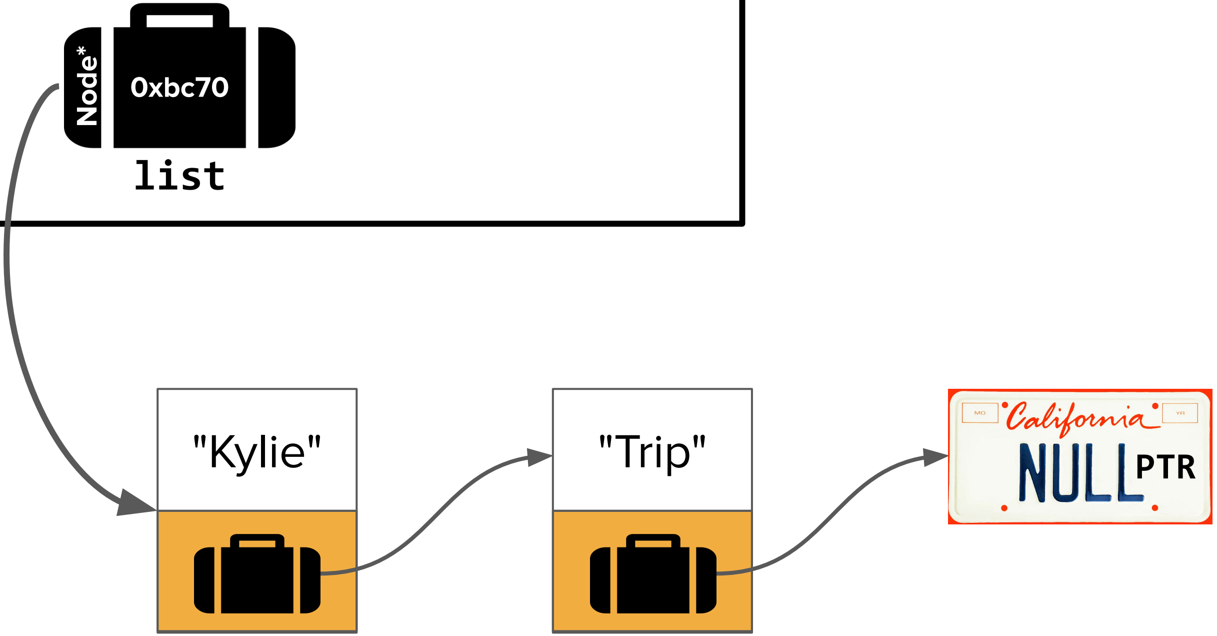
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



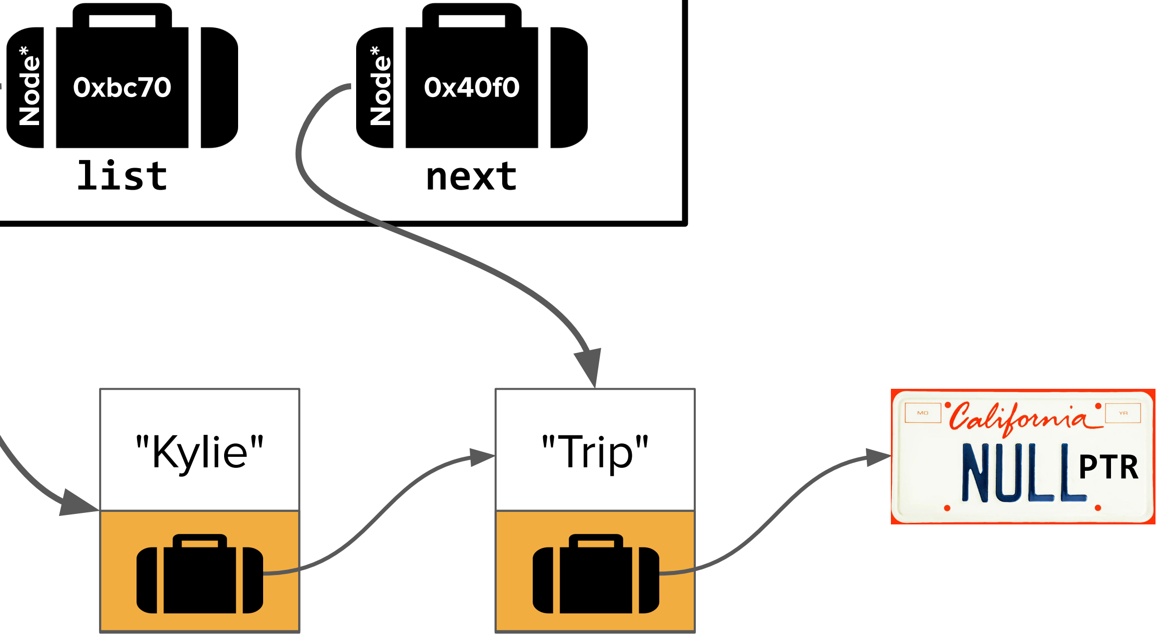
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



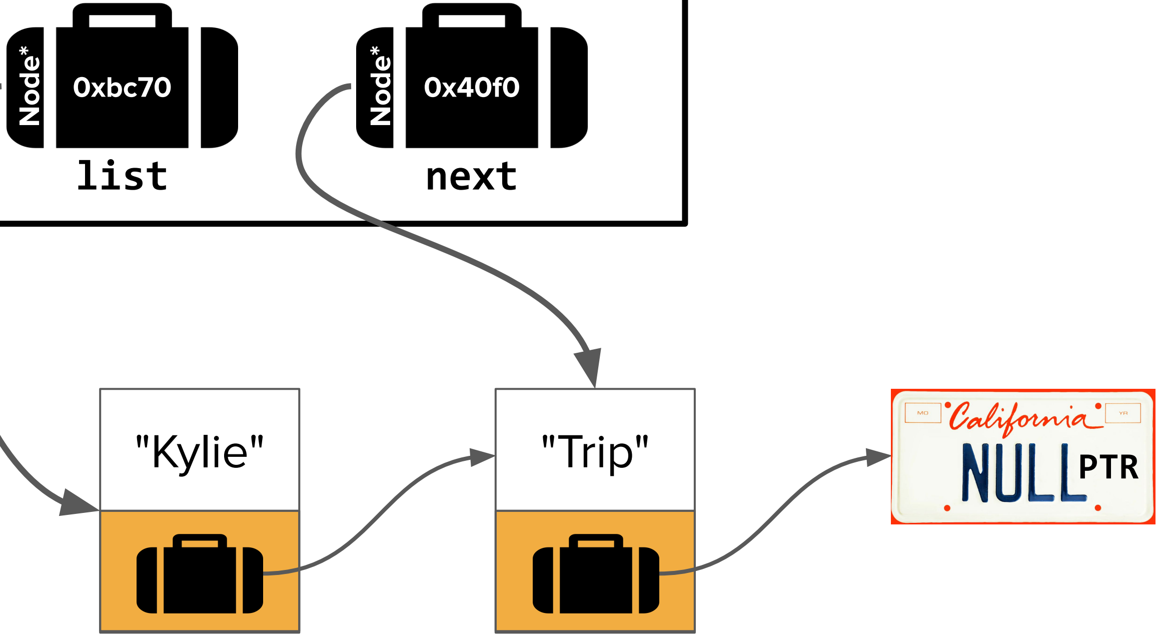
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



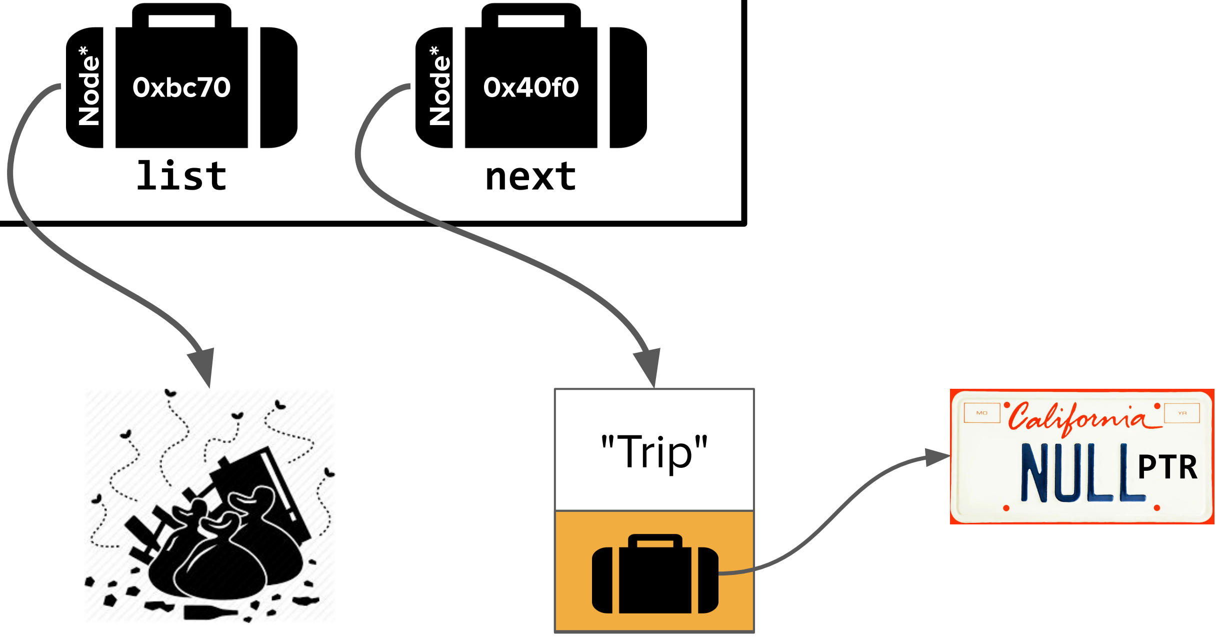
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



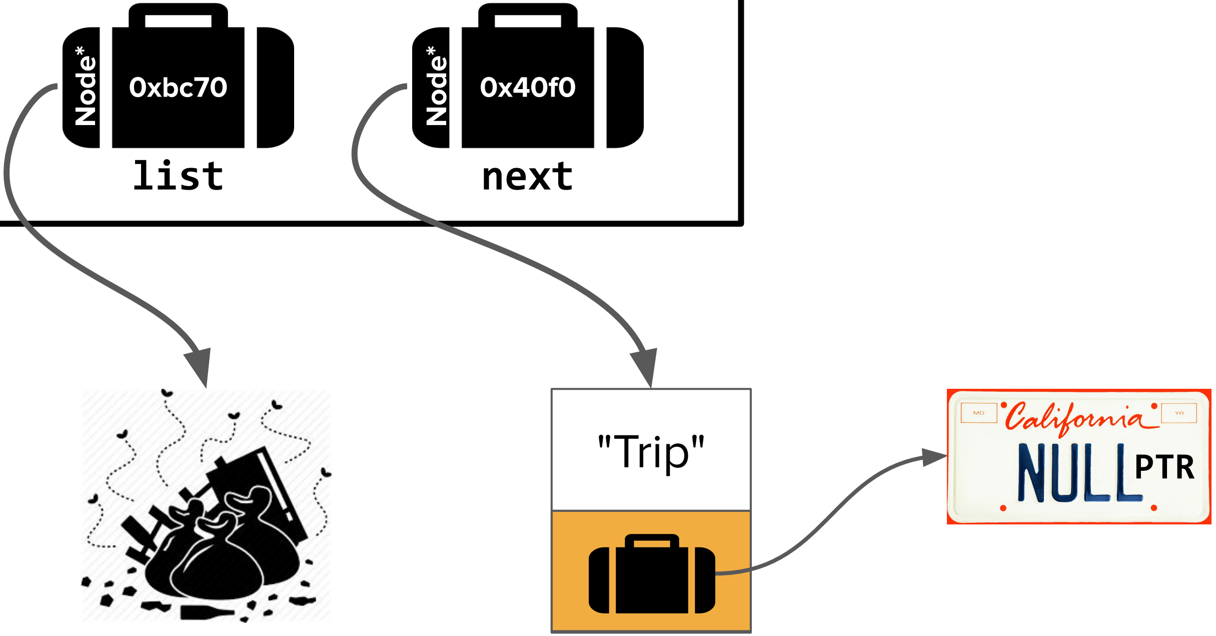
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



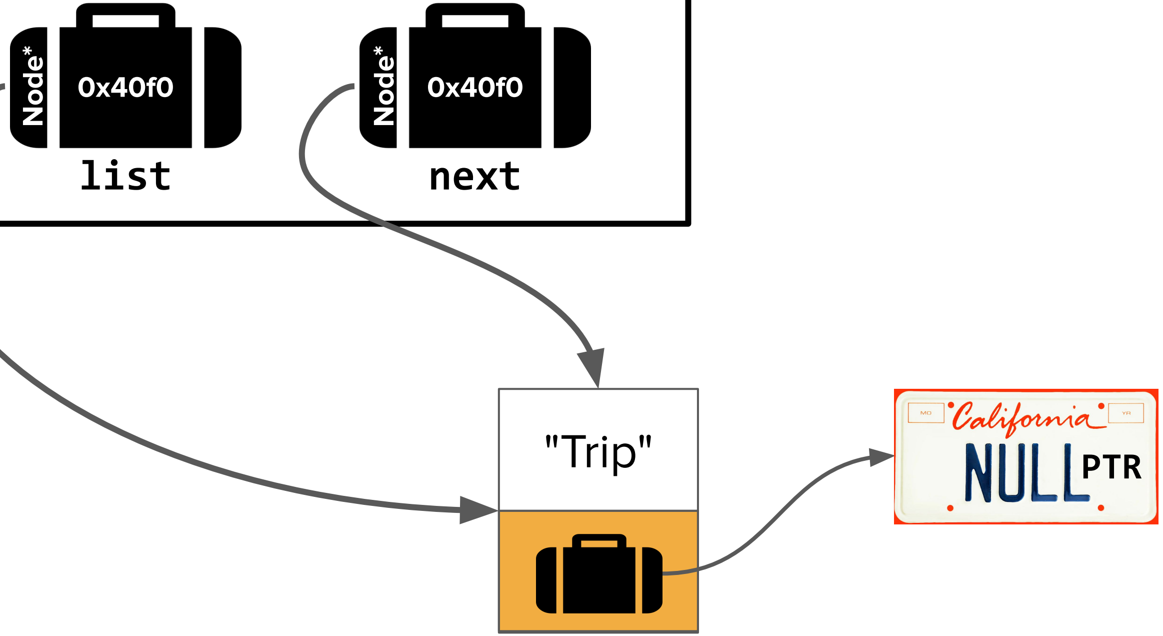
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



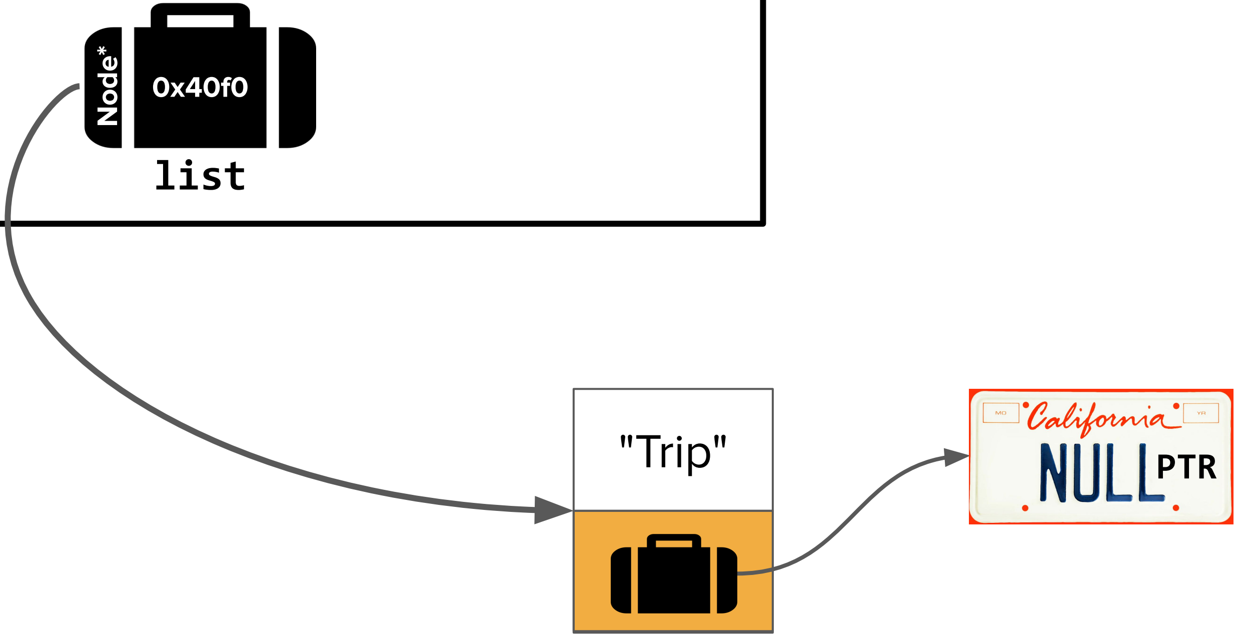
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



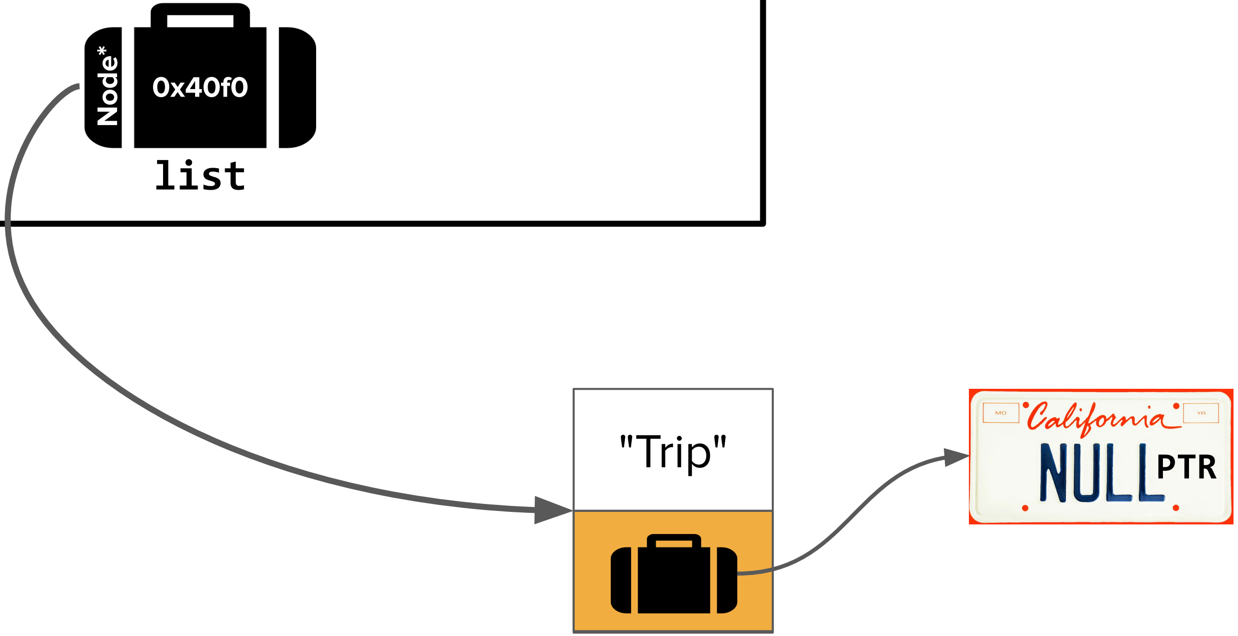
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



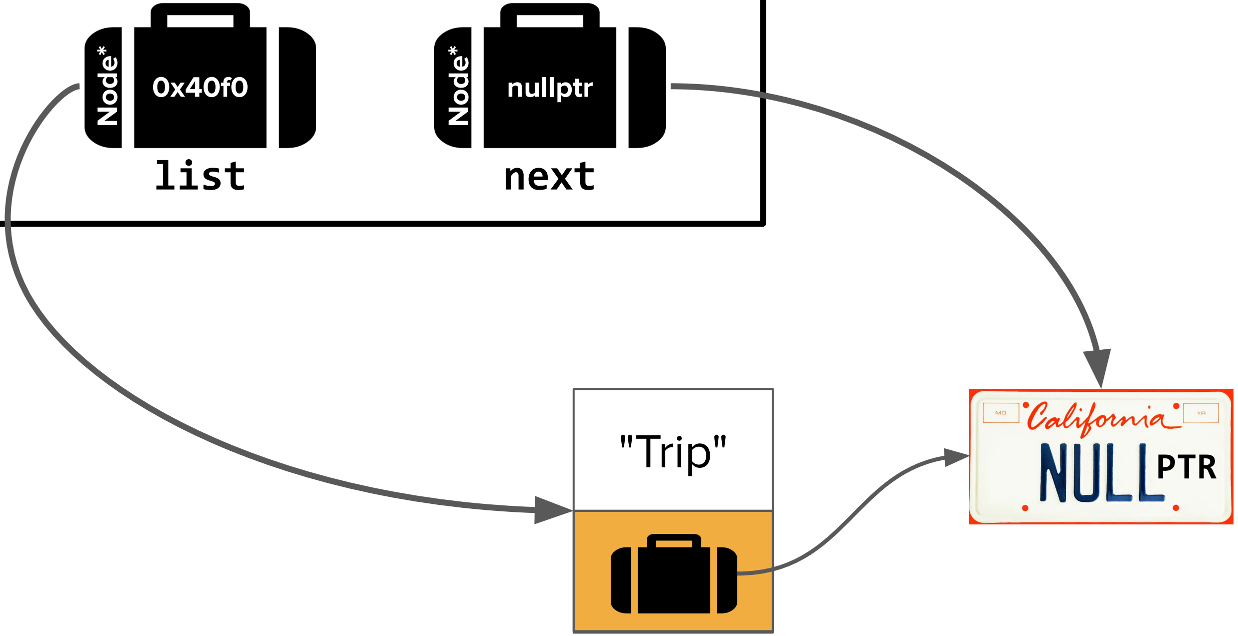
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



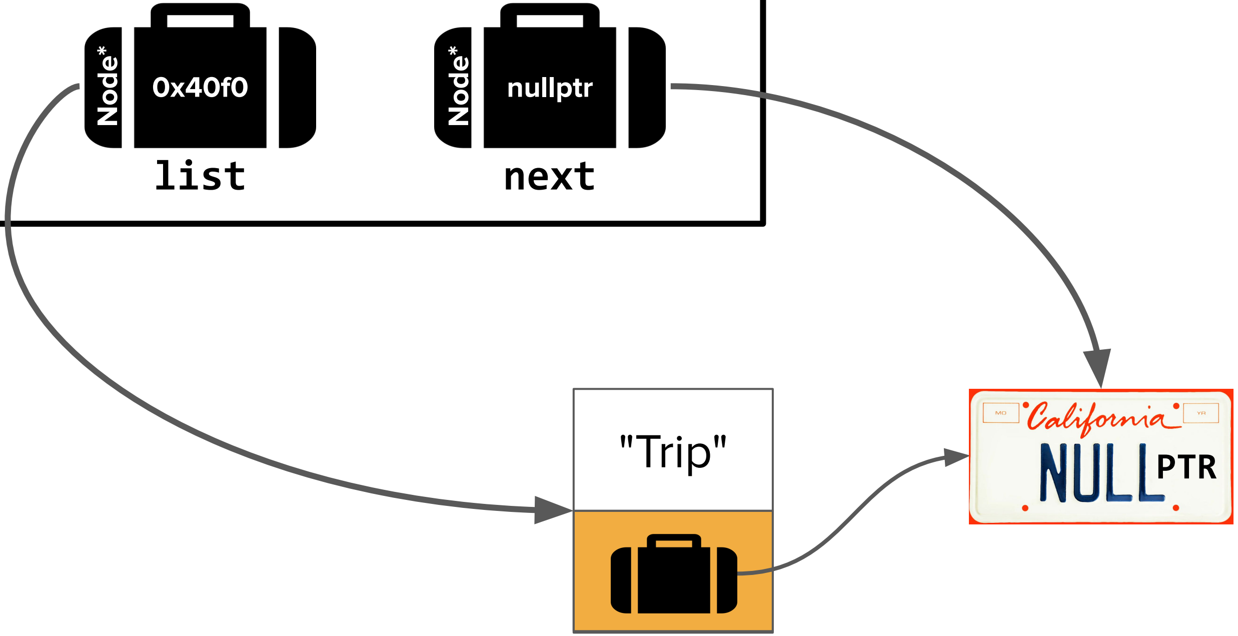
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



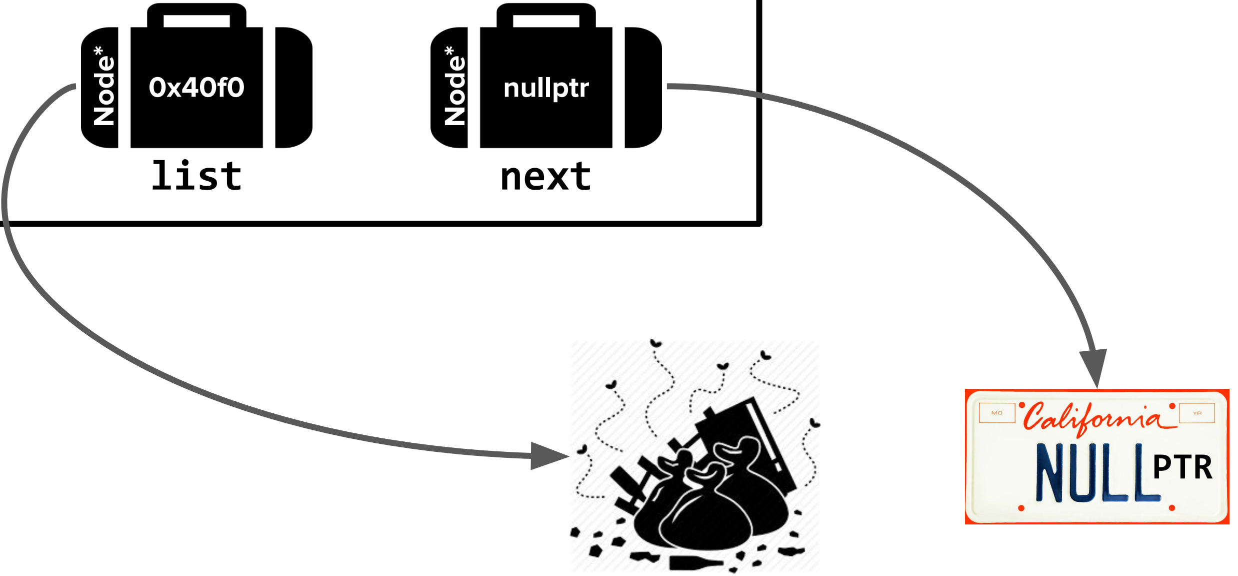
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



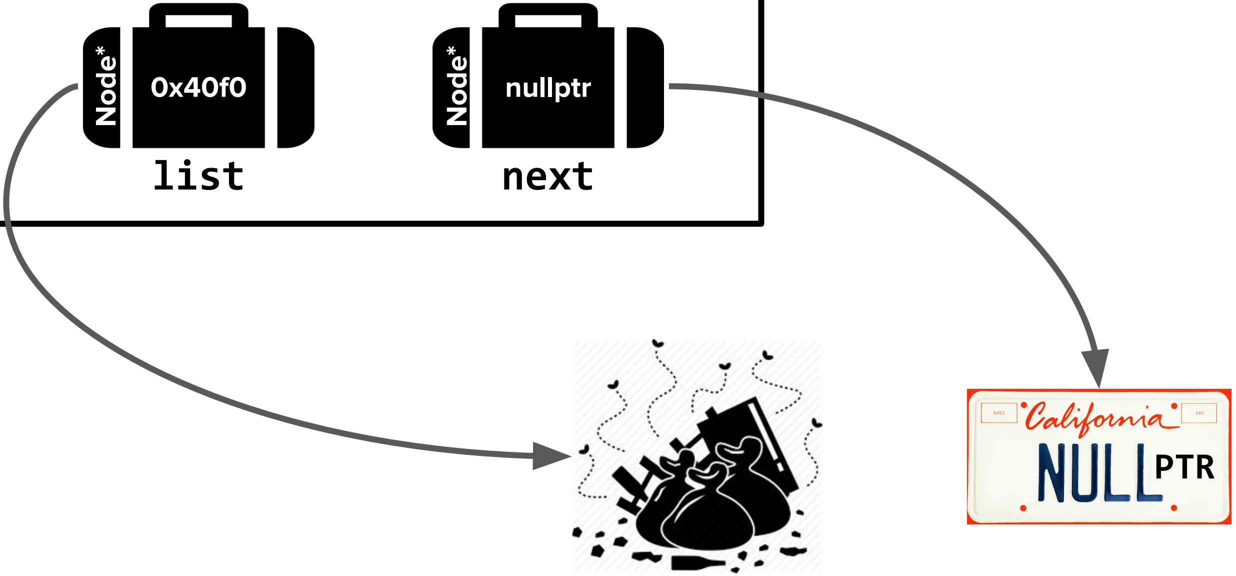
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



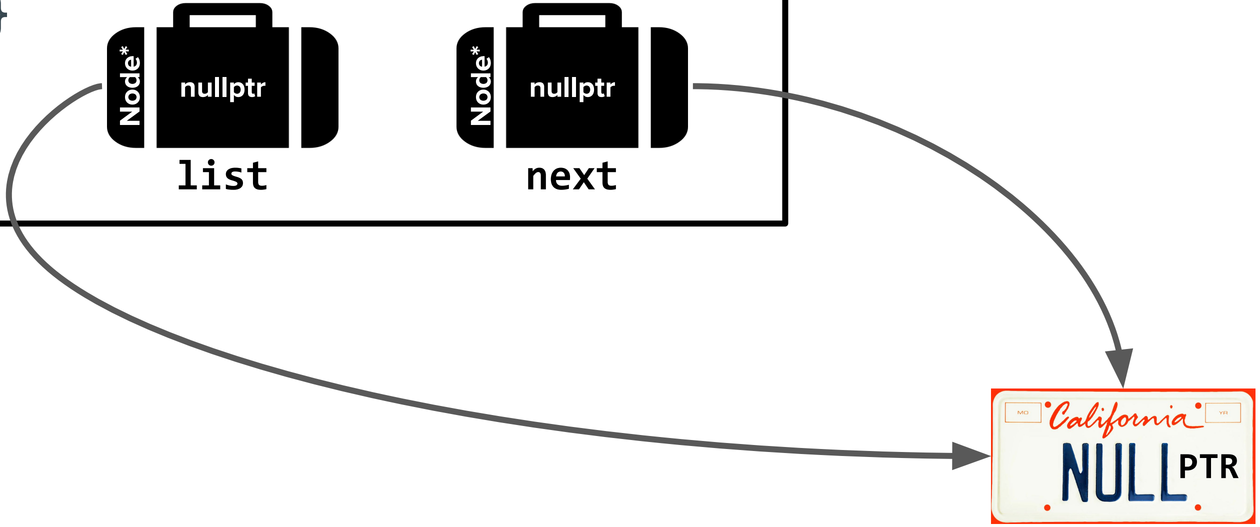
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



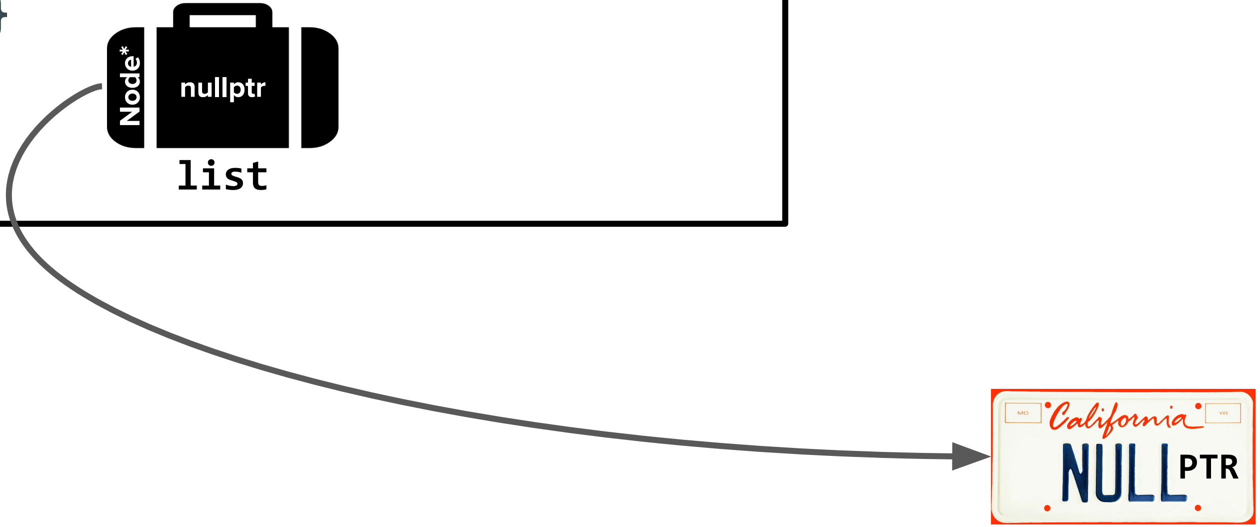
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



All memory freed! Wooo!

Traversal utility functions

- Freeing a linked list
- **Printing a linked list**
- Measuring the length of a list

Printing a linked list

Inspecting Linked List Contents

- Being able to "see" the contents of a linked list is a really helpful debugging tool!

Inspecting Linked List Contents

- Being able to "see" the contents of a linked list is a really helpful debugging tool!

- There are two main ways to do so: using the **debugger** and printing to the **console**

Inspecting Linked List Contents

- Being able to "see" the contents of a linked list is a really helpful debugging tool!
- There are two main ways to do so: using the **debugger** and printing to the **console**
- First attempt: What is the result of the following code? (Poll)

```
/* Creates a list with contents "Hello" -> "World" -> nullptr */
Node* list = createList();
cout << list << endl;
```

Inspecting Linked List Contents

- Being able to "see" the contents of a linked list is a really helpful debugging tool!

- There are two main ways to do so: using the **debugger** and printing to the **console**

- First attempt: What is the result of the following code? (Poll)

```
/* Creates a list with contents "Hello" -> "World" -> nullptr */
Node* list = createList();
cout << list << endl;
```

Answer: Some memory address is printed! We can't predict the exact value.

Inspecting Linked List Contents

- Being able to "see" the contents of a linked list is a really helpful debugging tool!
- There are two main ways to do so: using the **debugger** and printing to the **console**
- First attempt (directly printing list pointer) unsuccessful.
- Second attempt: Let's write a function to print the list!

`printList()`

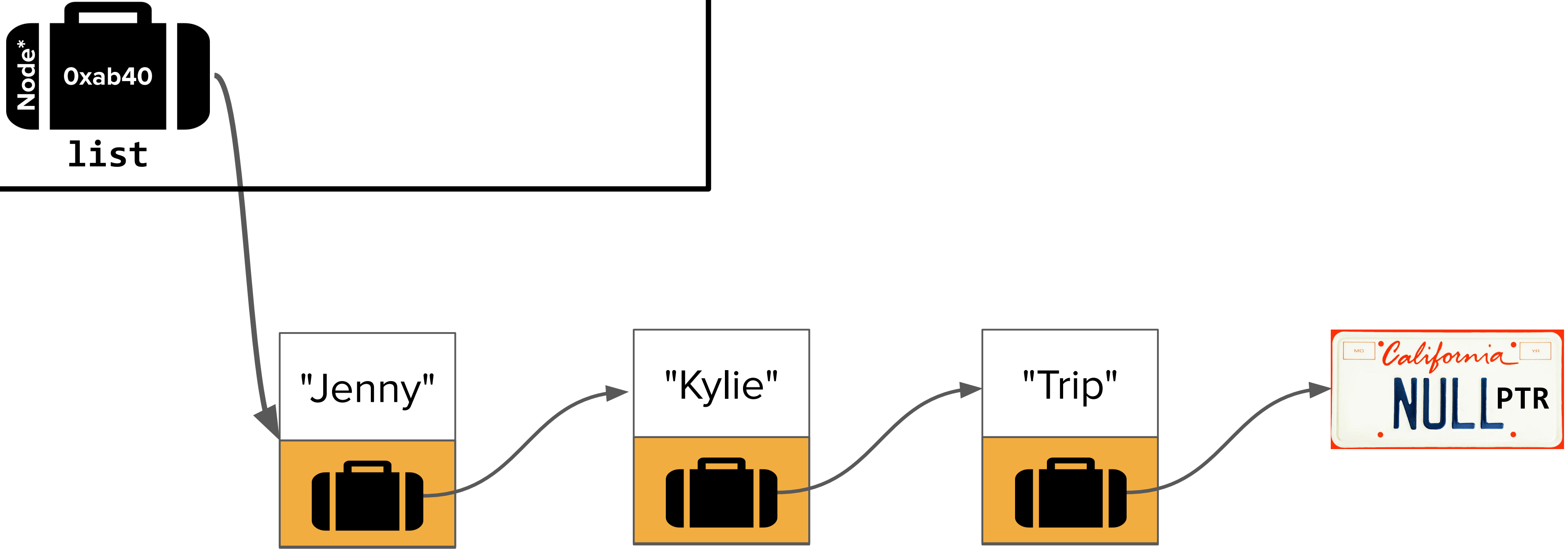
Let's code it!

How does it work?

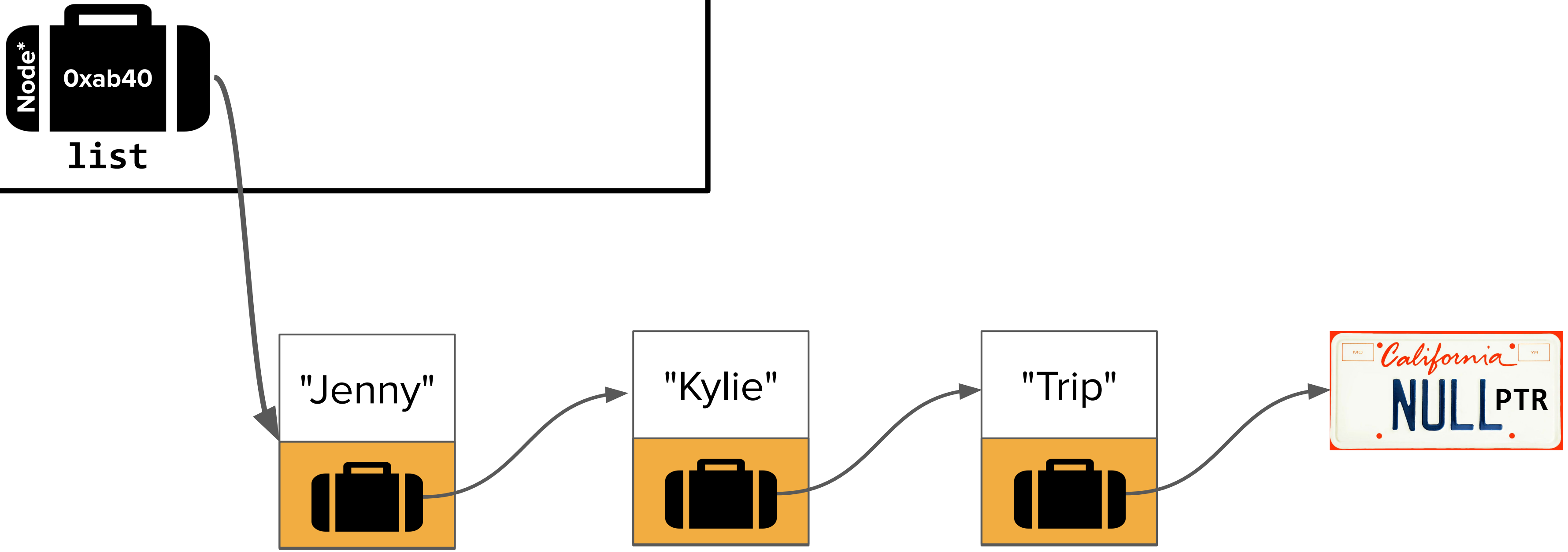
```
int main() {
    Node* list = readList();
    printList(list);

    /* other list things happen... */
}
```

```
int main() {
    Node* list = readList();
    printList(list);

    /* other list things happen... */
}
```

```
int main() {
    Node* list = readList();
    printList(list);

    /* other list things happen... */
}
```

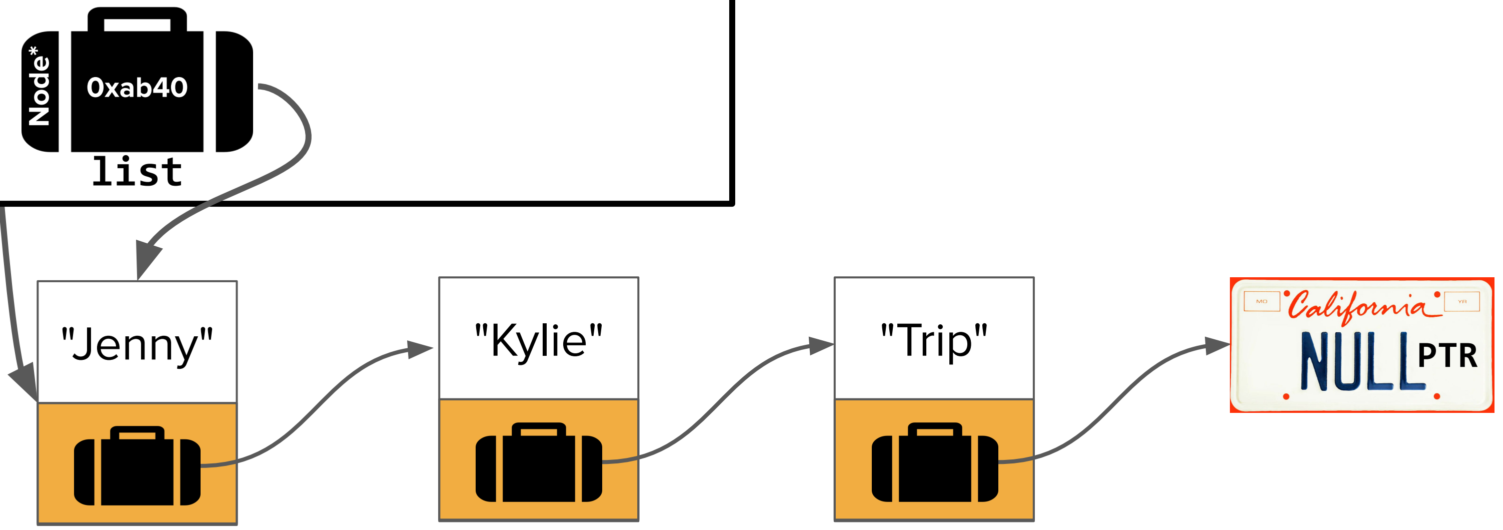


```
int main() {
    Node* list = readList();
    printList(list);

    /* other list things happen... */
}
```



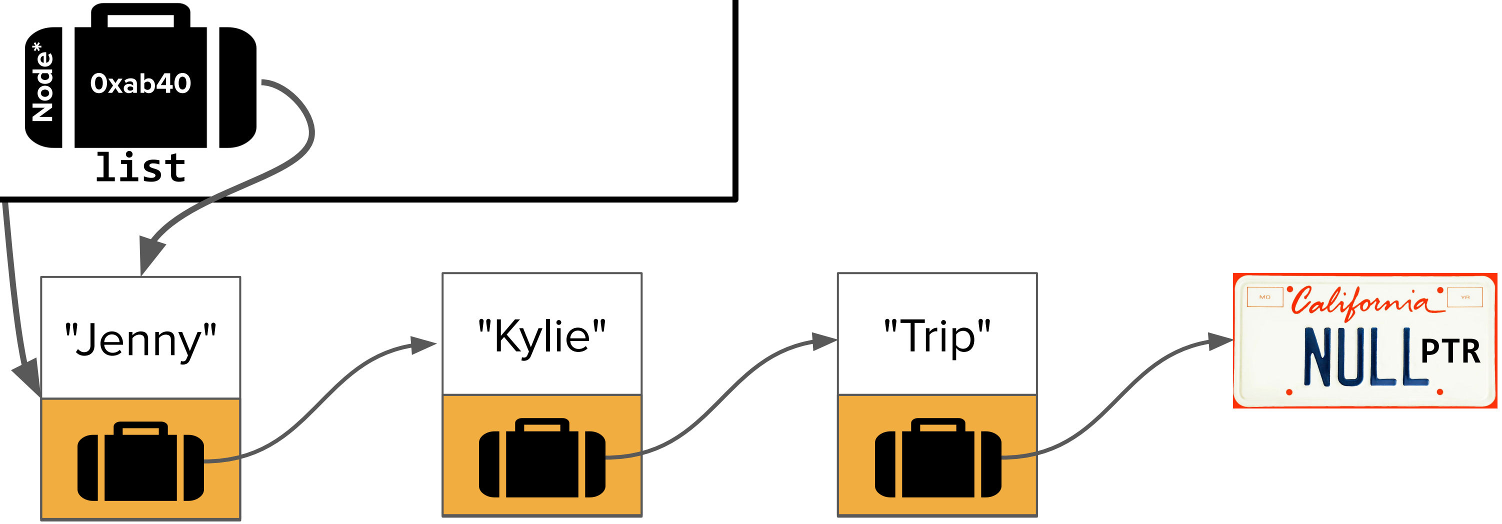
```
int main() {

}
```

```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```



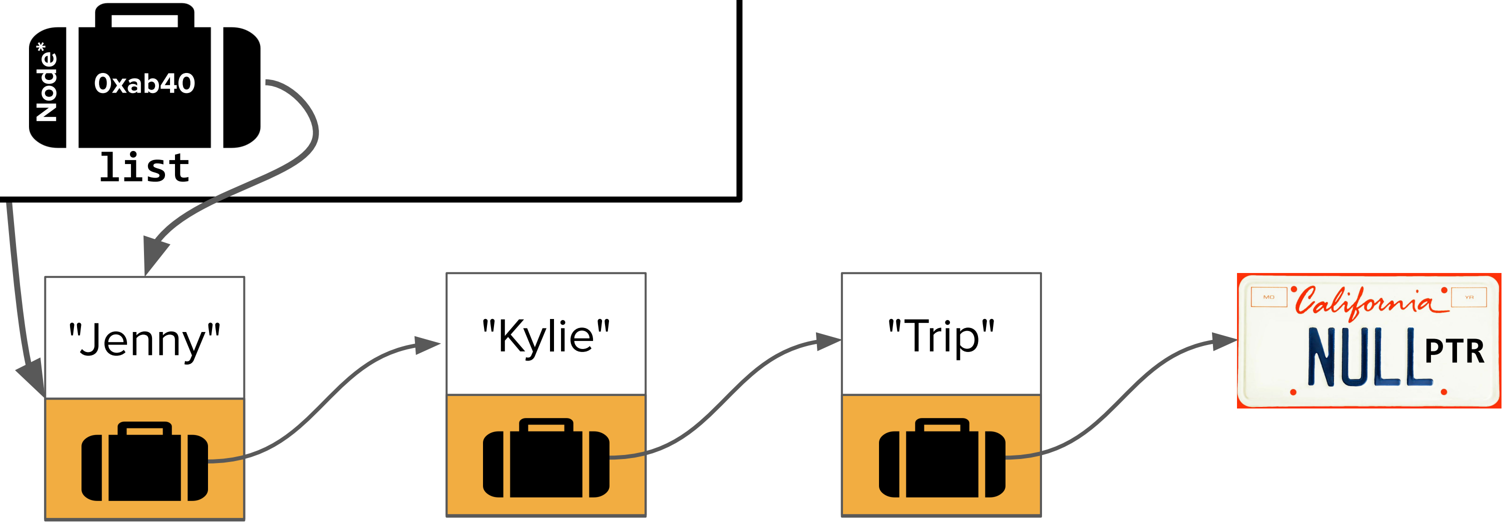
```
int main() {



}
```

```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```



```
int main() {

}
```

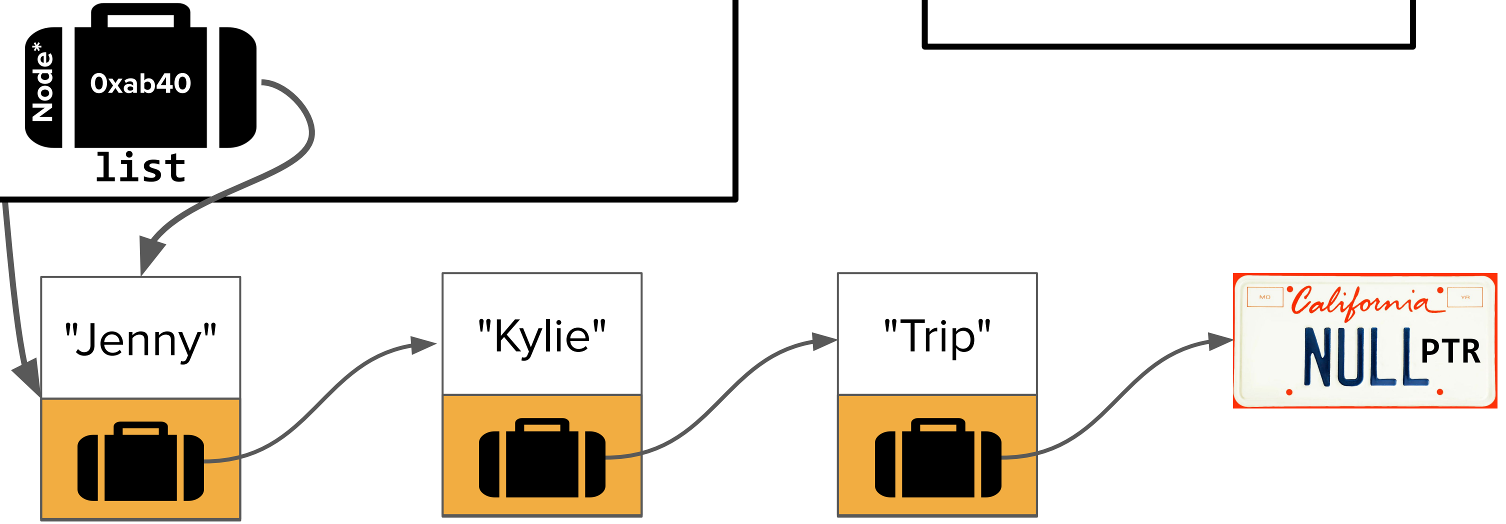
```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```



```
int main() {

}
```

```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```

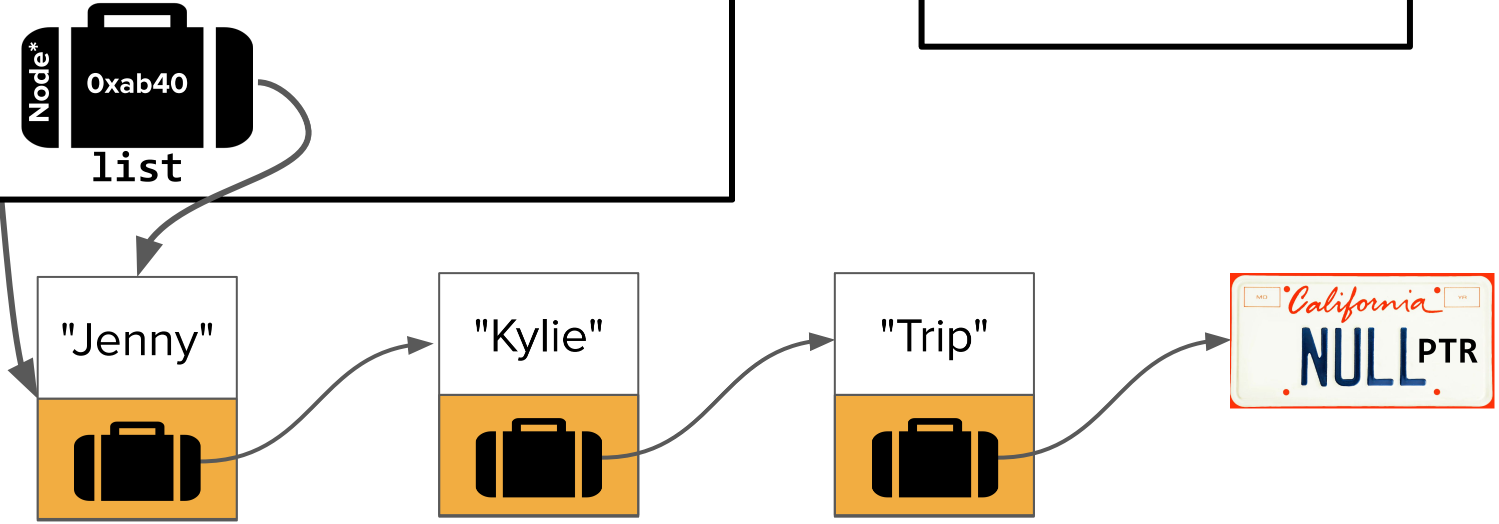
```
Jenny
```



```
int main() {

}
```

```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```

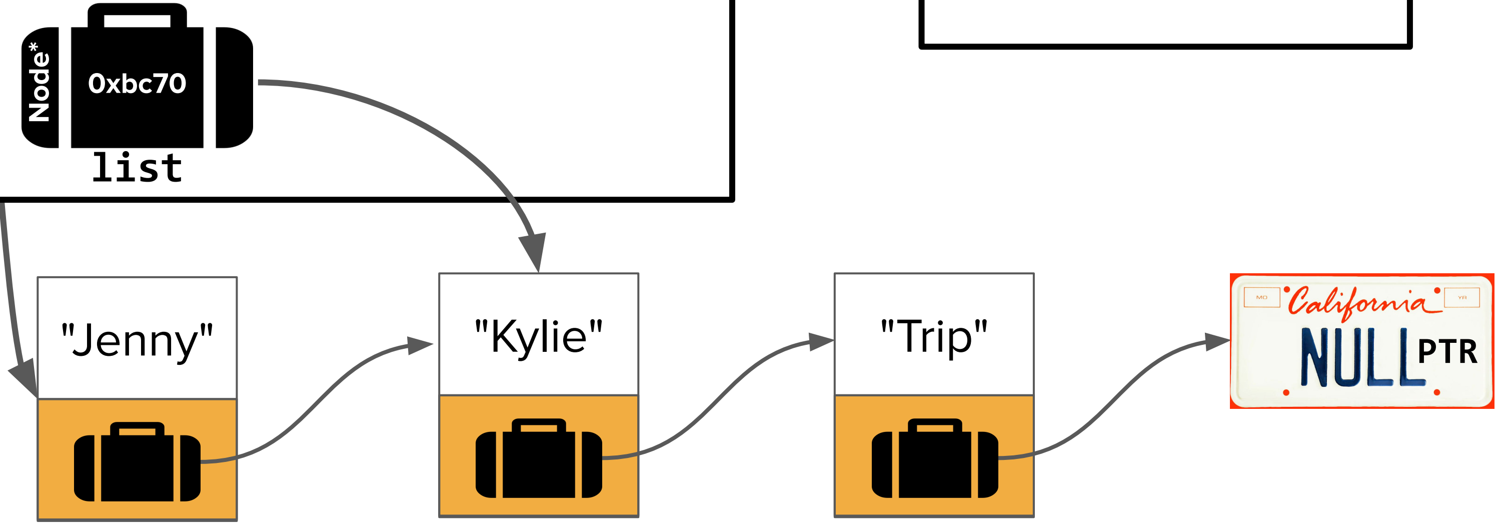
```
Jenny
```



```
int main() {

}
```

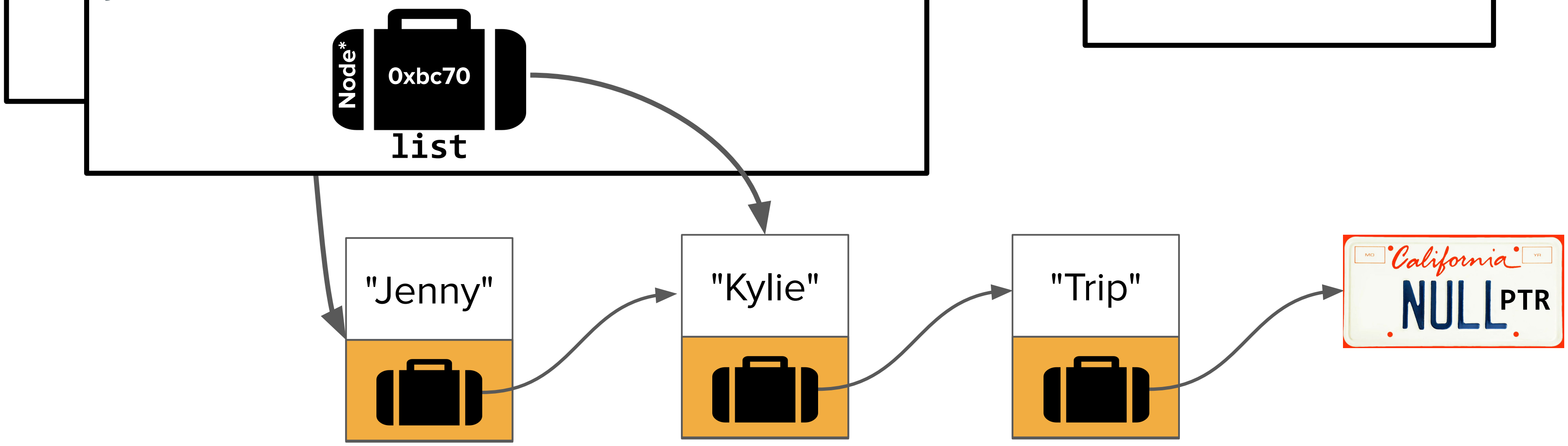
```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```

```
Jenny
```



```
int main() {

}
```

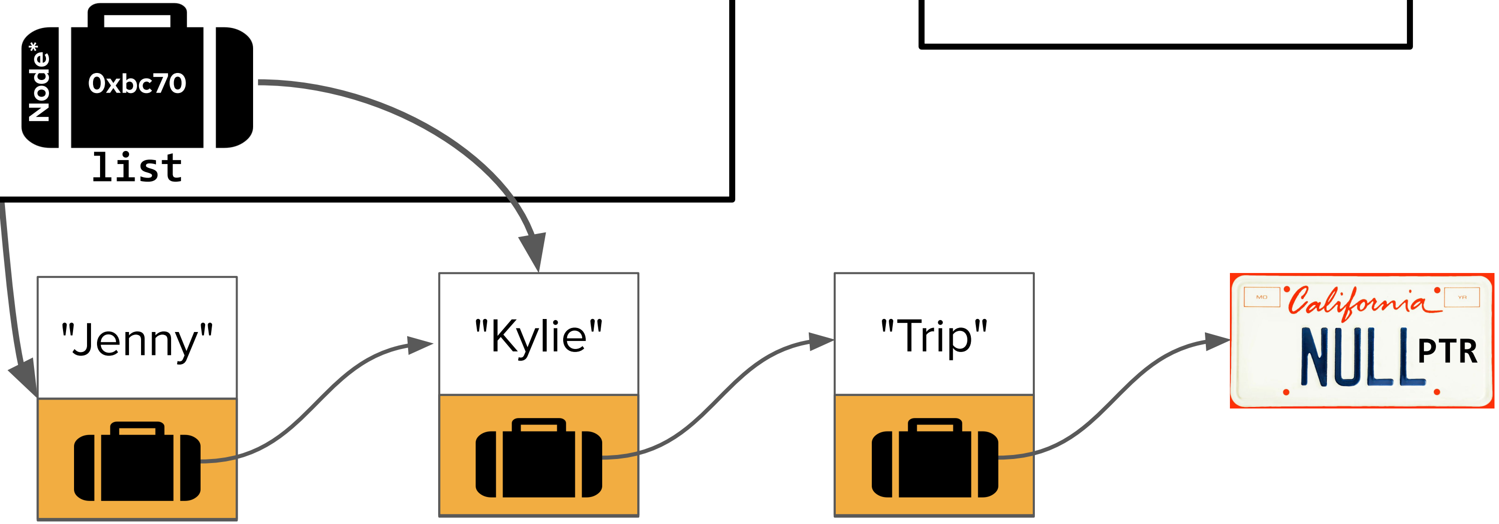
```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```



```
Jenny
```

```
int main() {

}
```

```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```

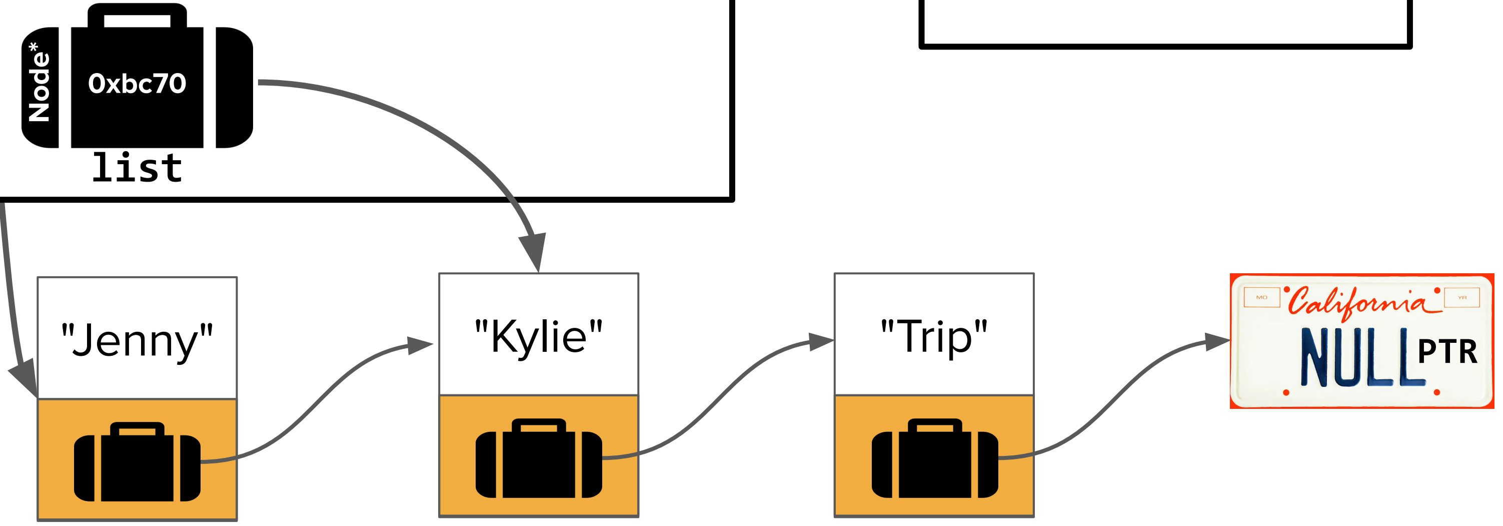


```
Jenny
```

```
int main() {

}
```

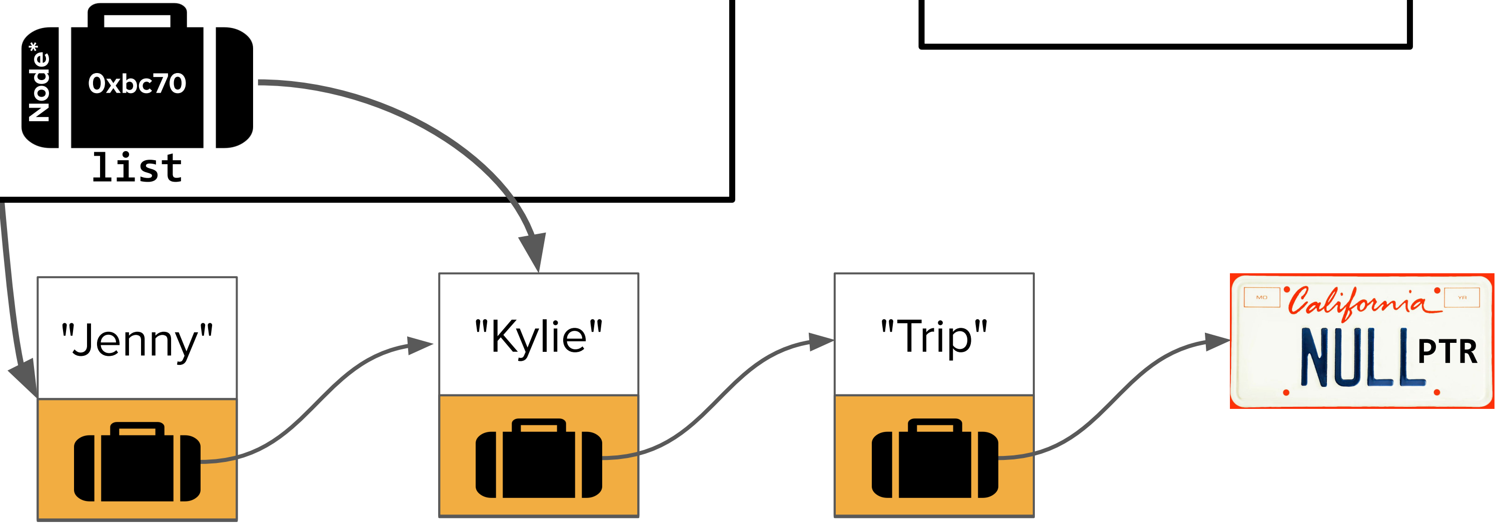
```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```

```
Jenny
Kylie
```



```
int main() {



}
```

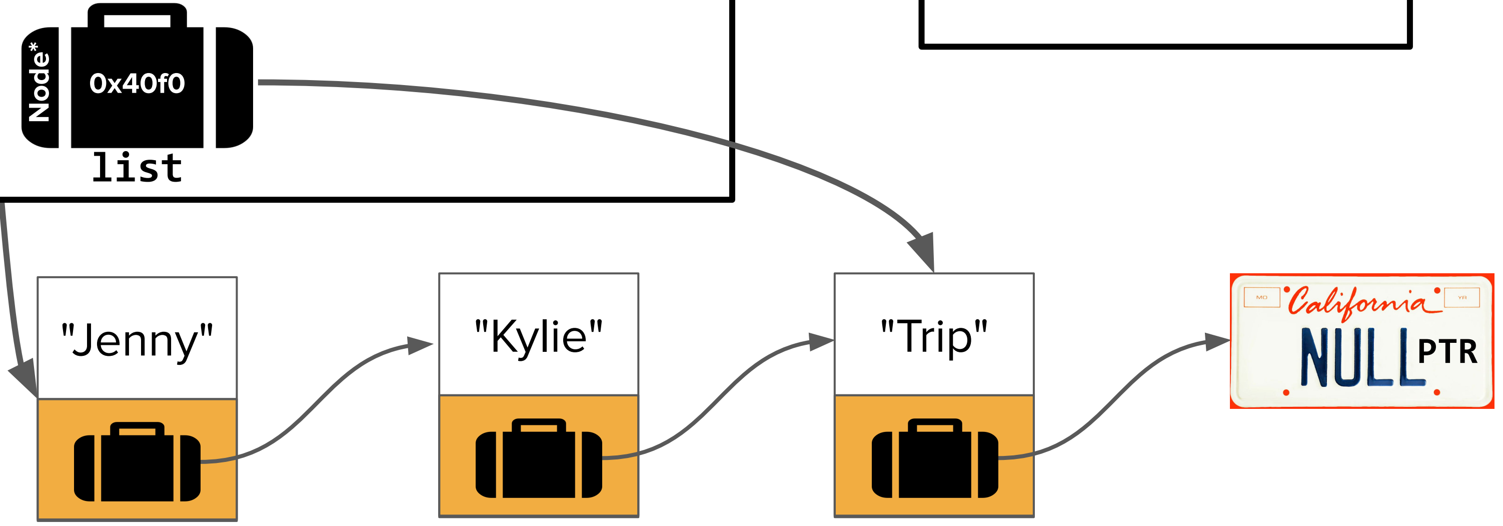
```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```



```
Jenny
Kylie
```

```
int main() {



}
```

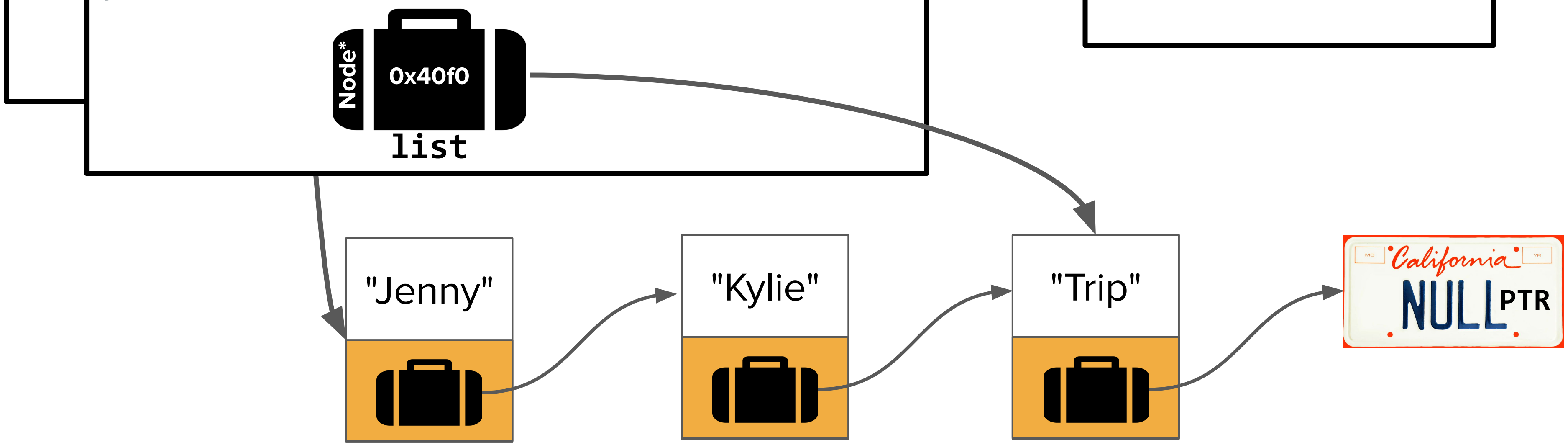
```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```



```
Jenny
Kylie
```

```
int main() {

}
```

```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```

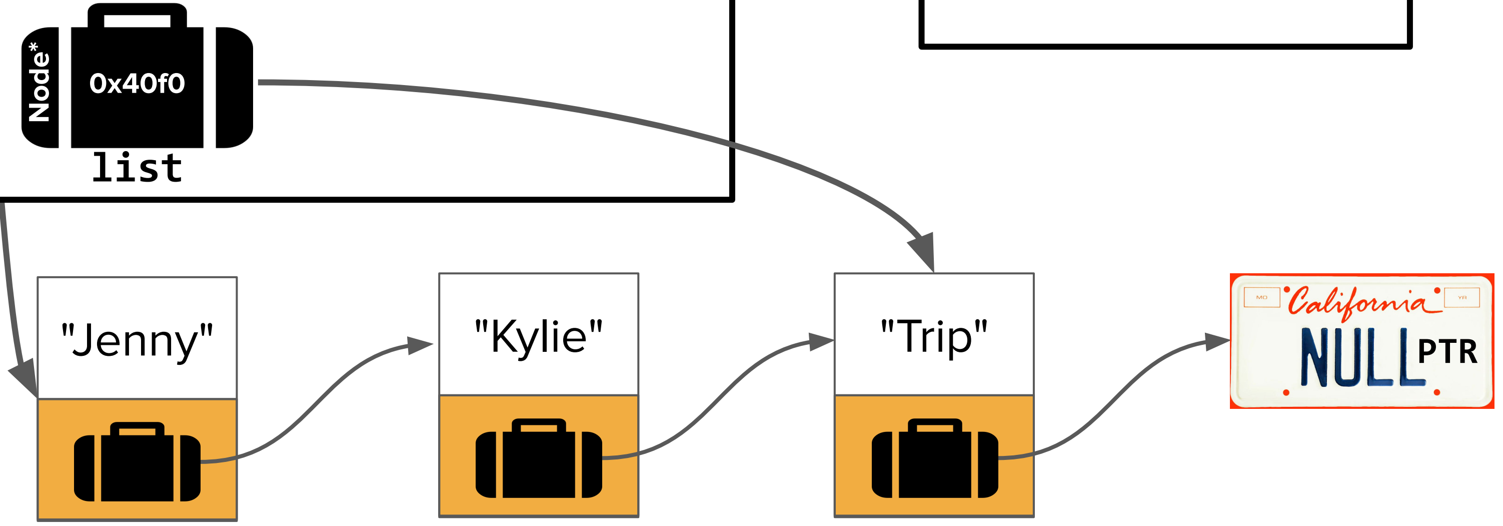


```
Jenny
Kylie
```

```
int main() {

}
```

```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```

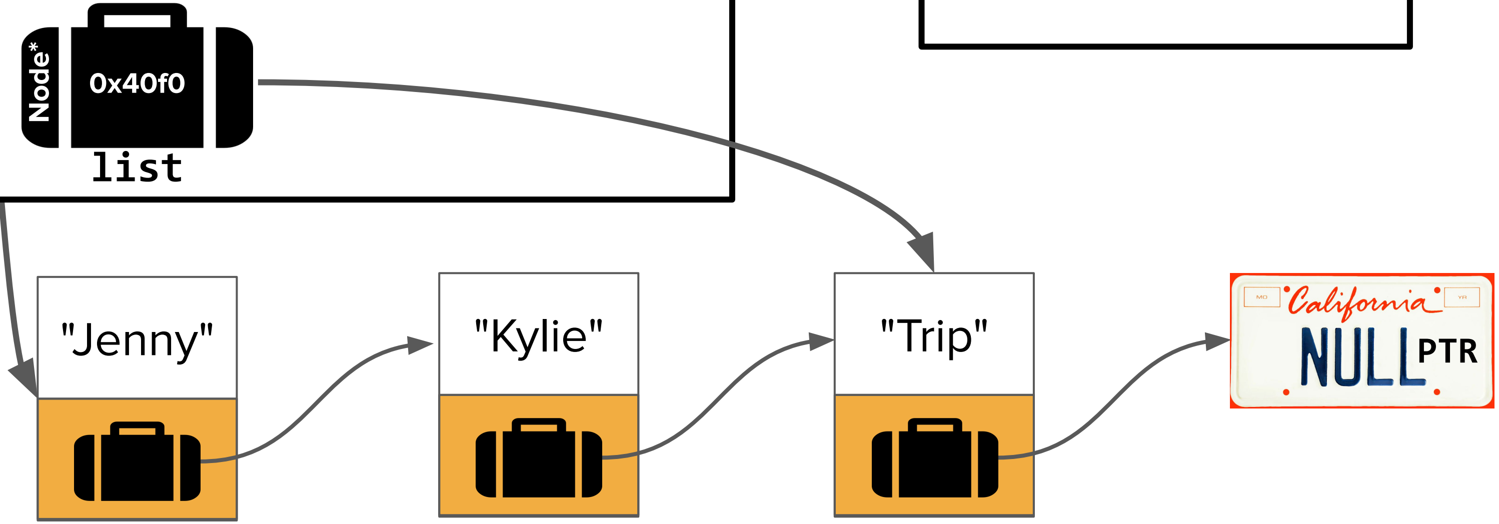
```
Jenny
Kylie
```



```
int main() {

}
```

```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```

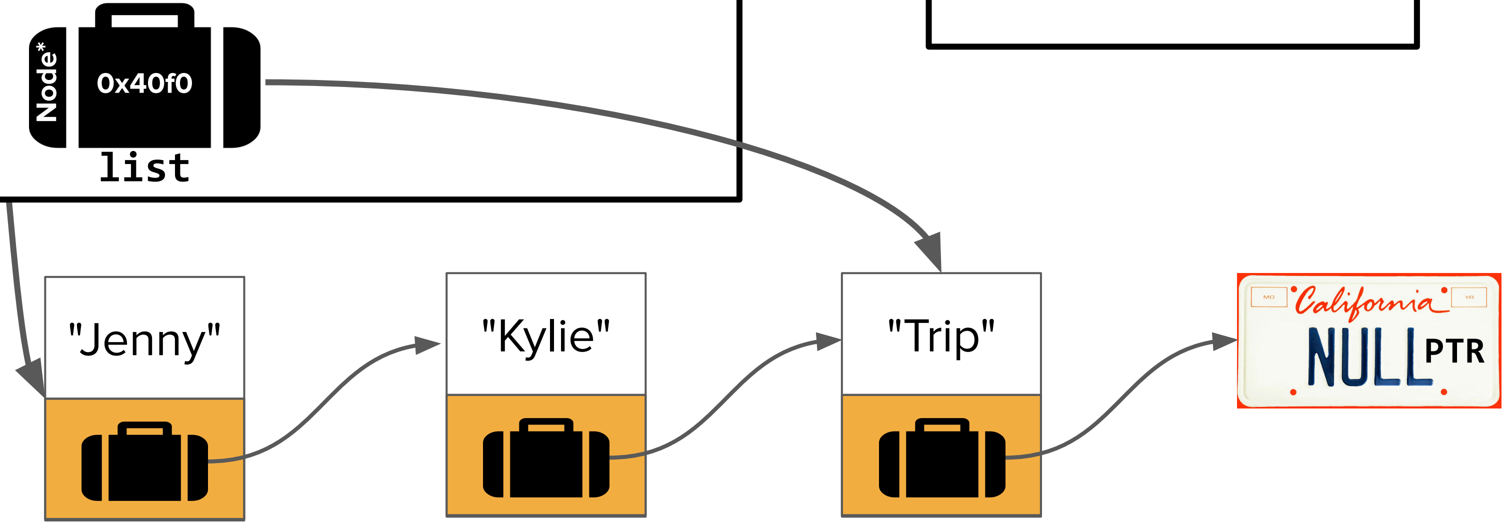
```
Jenny
Kylie
Trip
```



```
int main() {

}
```

```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```

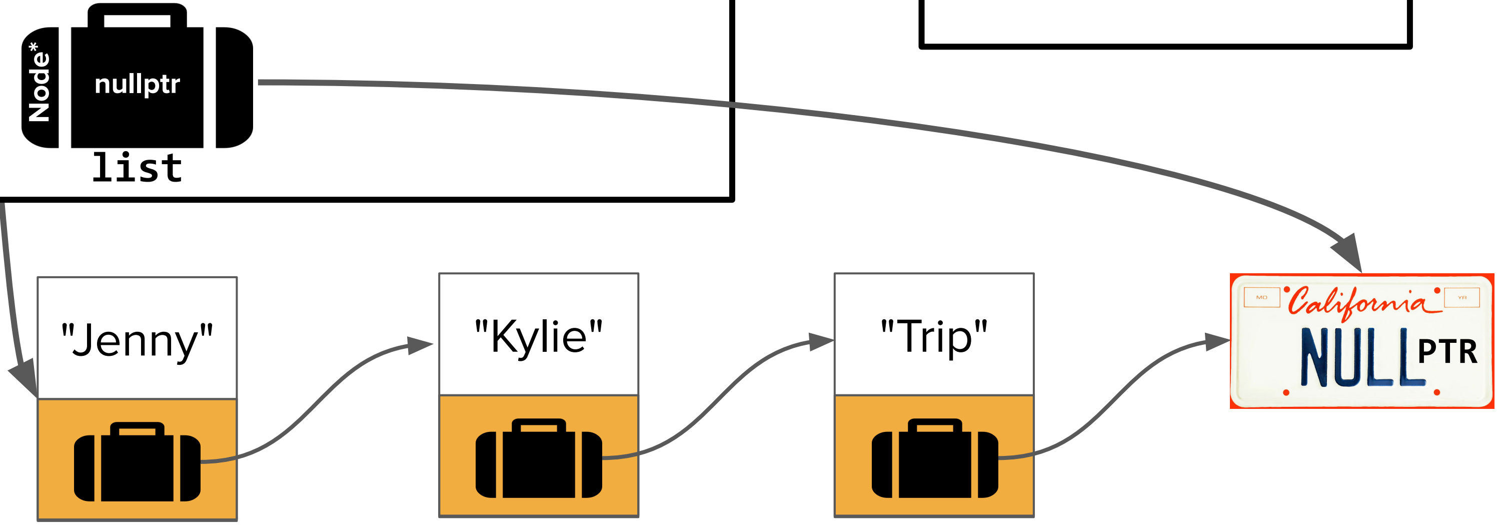
```
Jenny
Kylie
Trip
```



```
int main() {



}
```

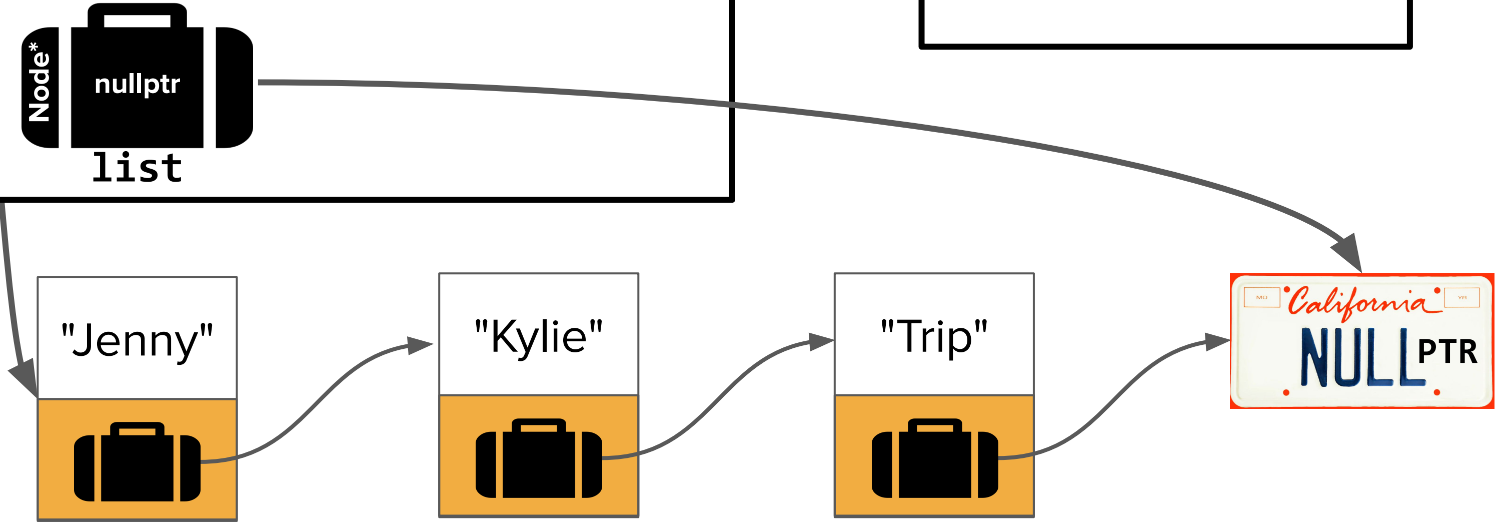
```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```

```
Jenny
Kylie
Trip
```



```
int main() {

}
```

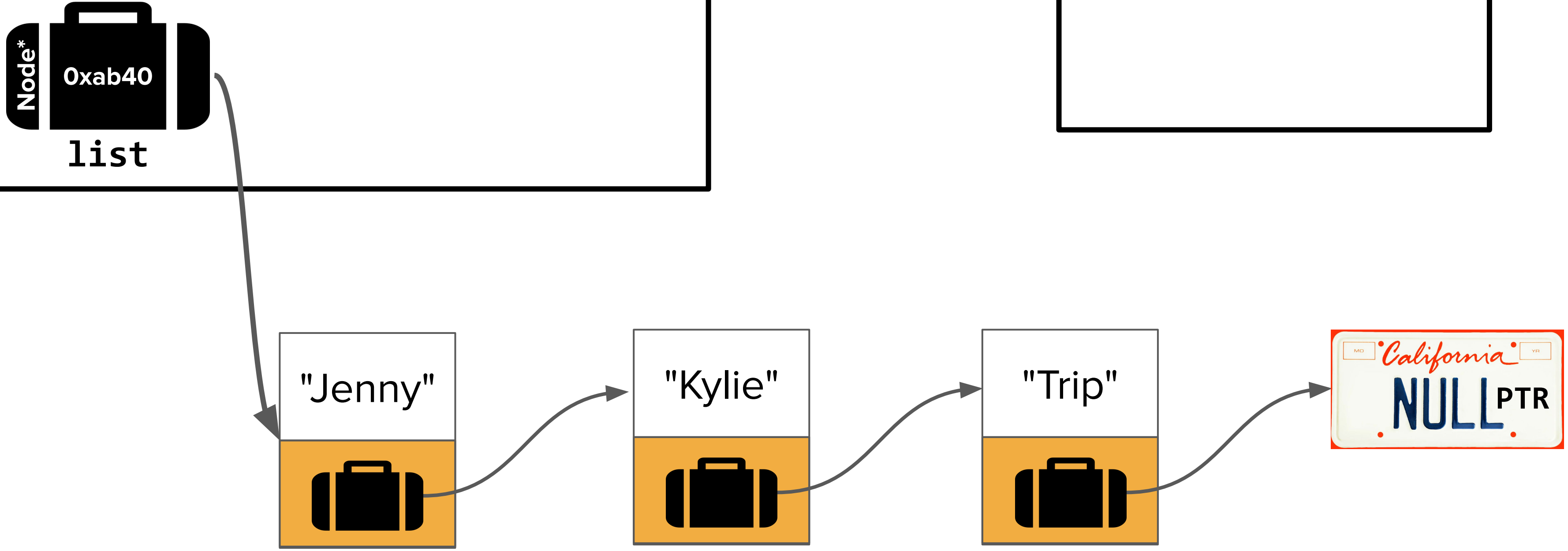
```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```



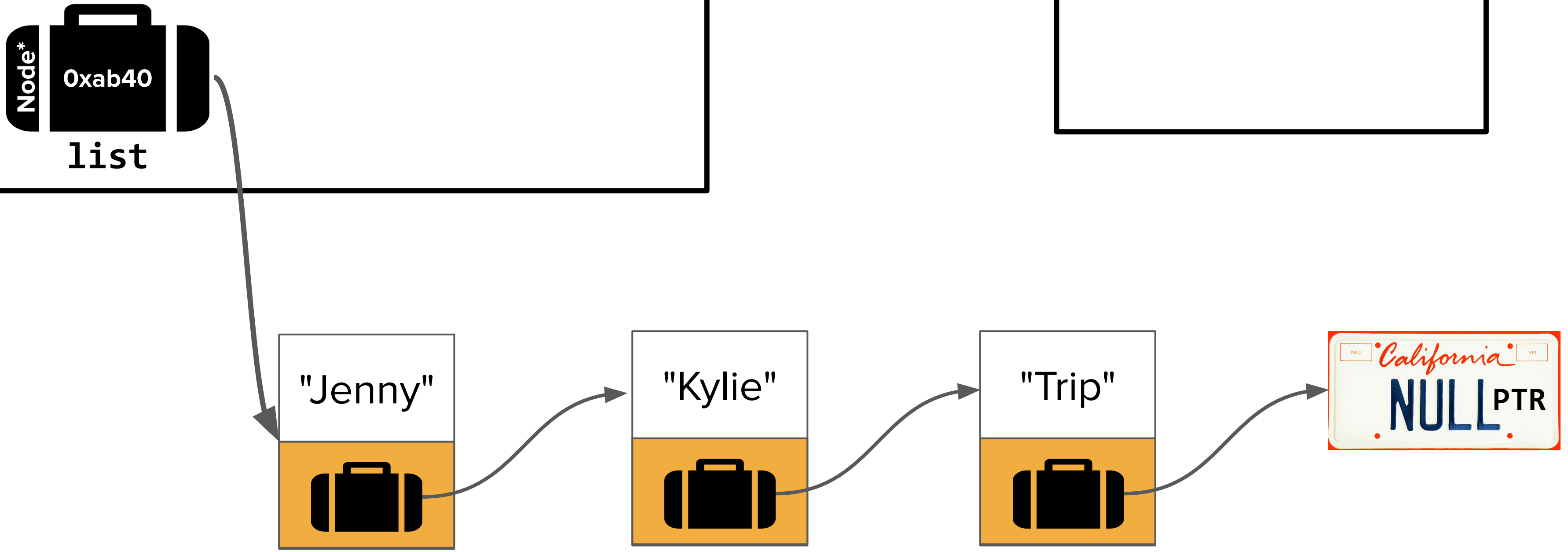
```
Jenny
Kylie
Trip
```

```
int main() {
    Node* list = readList();
    printList(list);

    /* other list things happen... */
}
```

```
Jenny
Kylie
Trip
```



```
int main() {
    Node* list = readList();
    printList(list);

    /* other list things happen... */
}
```



```
Jenny
Kylie
Trip
```

Traversal utility functions

- Freeing a linked list

- Printing a linked list

- **Measuring the length of a list**
    - We’ll go over this is as a warmup on Friday!

Summary

Linked lists can be used in standalone utility functions or in the context of classes!

Common linked lists operations

- **Traversal**
  - How do we walk through all elements in the linked list?
- **Rewiring**
  - How do we rearrange the elements in a linked list?
- **Insertion**
  - How do we add an element to a linked list?
- **Deletion**
  - How do we remove an element from a linked list?

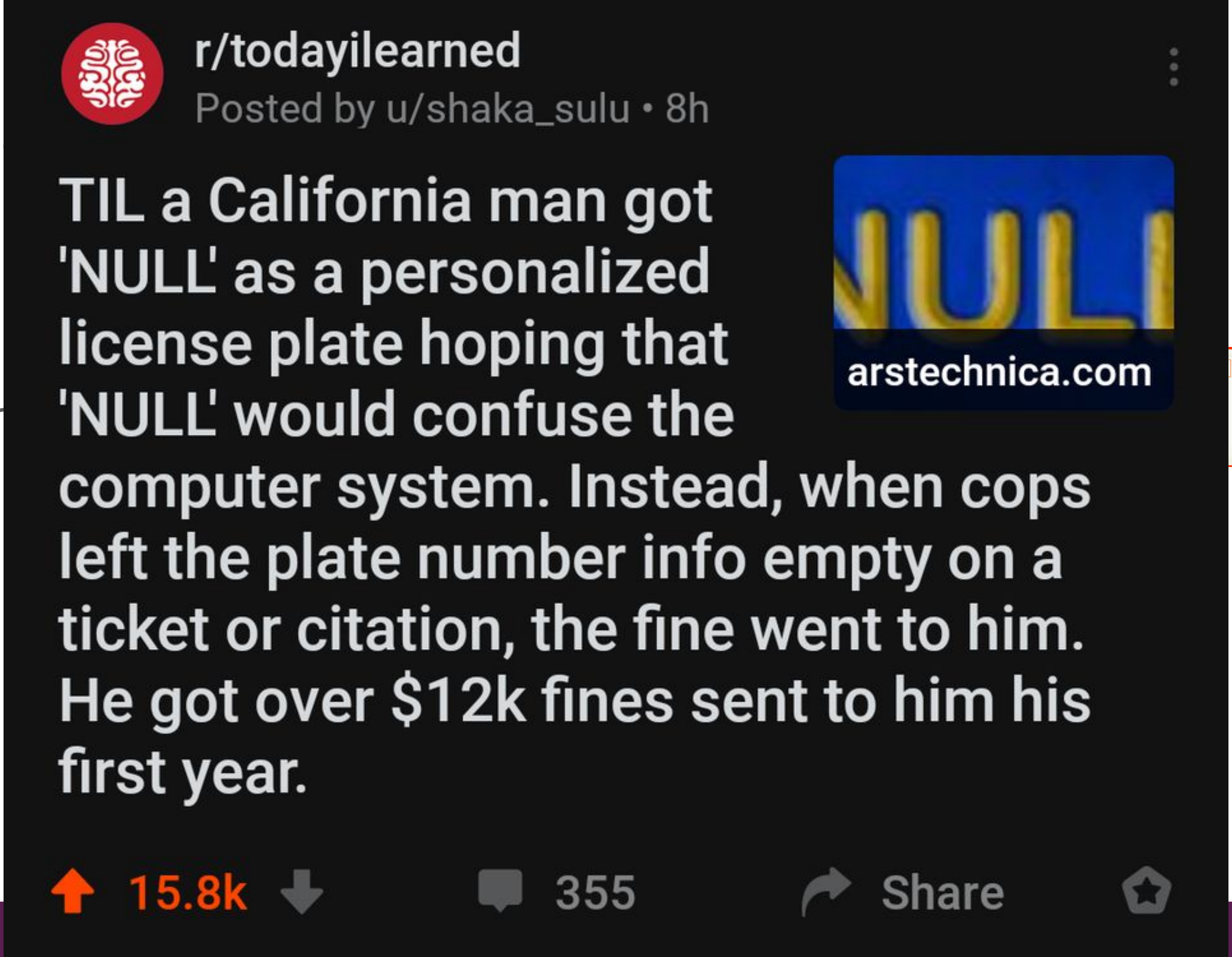
Linked list traversal takeaways

- Temporary pointers into lists are very helpful!
    - When processing linked lists iteratively, it’s common to introduce pointers that point to cells in multiple spots in the list.
    - This is particularly useful if we’re destroying or rewiring existing lists.

- Using a **`while`** loop with a condition that checks to see if the current pointer is **`nullptr`** is the prevailing way to traverse a linked list.

A link

0xfca0b000
ptr



r/todayilearned
Posted by u/shaka_sulu • 8h

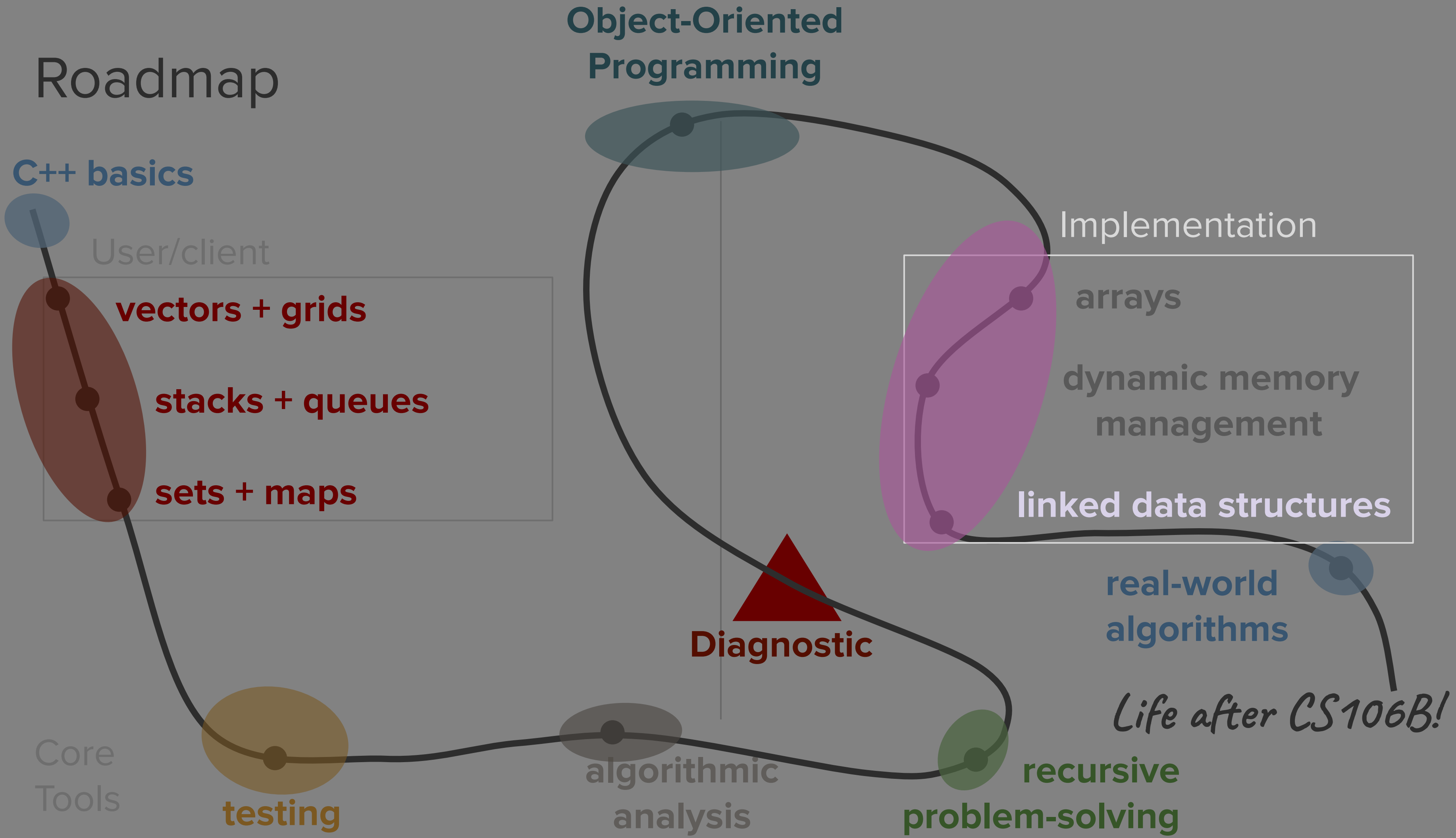
TIL a California man got 'NULL' as a personalized license plate hoping that 'NULL' would confuse the computer system. Instead, when cops left the plate number info empty on a ticket or citation, the fine went to him. He got over $12k fines sent to him his first year.

arstechnica.com

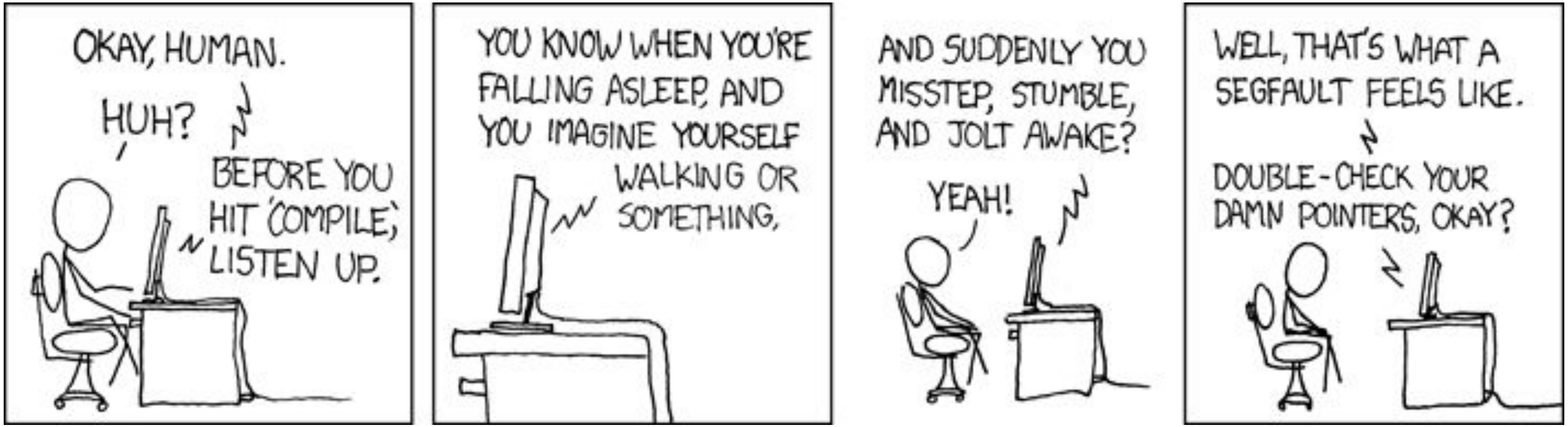
15.8k 355 Share

What’s next?

Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

More on linked lists!


OKAY, HUMAN.
HUH?
BEFORE YOU HIT 'COMPILE,' LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP, AND YOU IMAGINE YOURSELF WALKING OR SOMETHING,
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, THAT'S WHAT A SEGFAULT FEELS LIKE.
DOUBLE-CHECK YOUR DAMN POINTERS, OKAY?