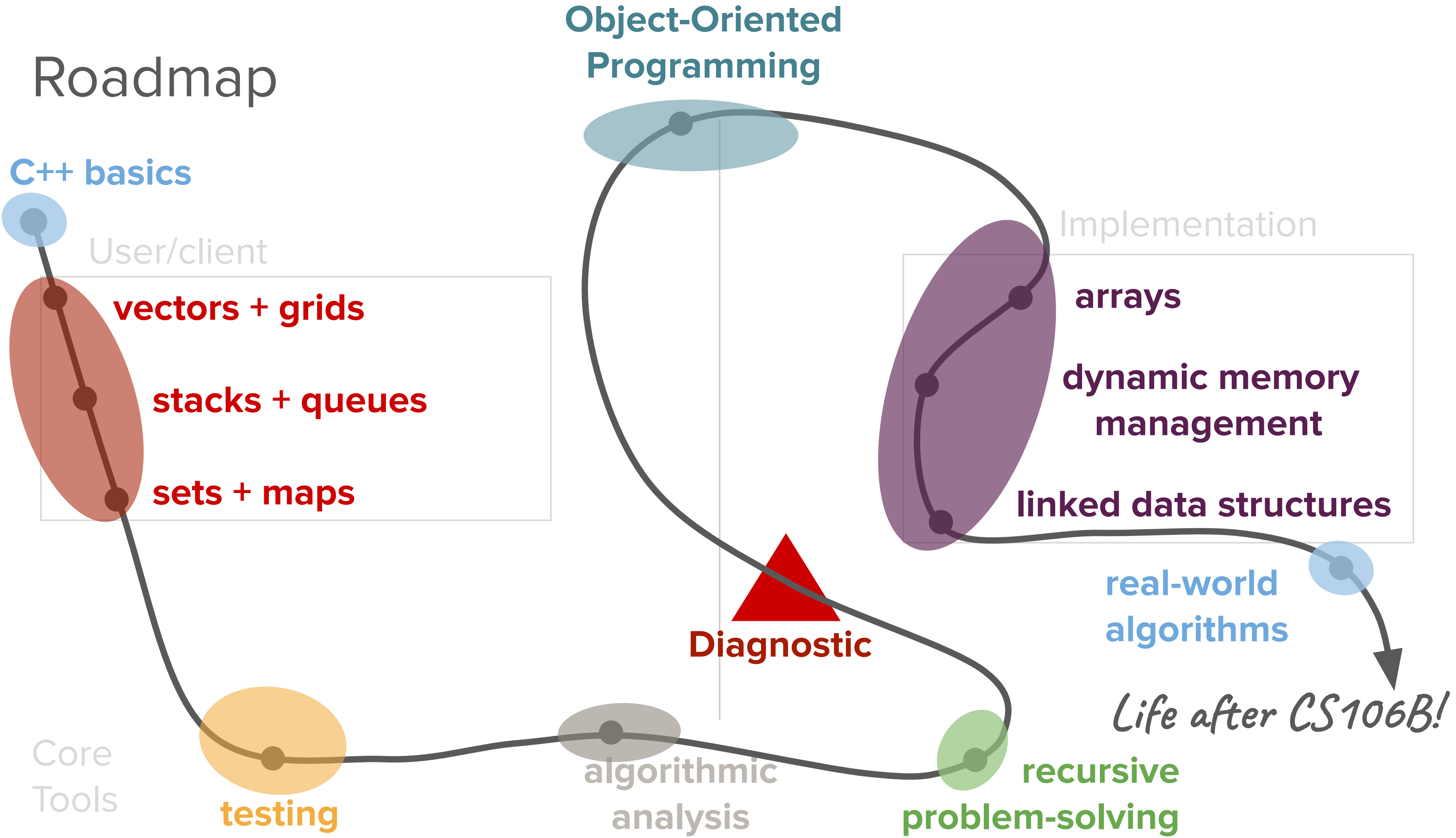
Linked List Operations

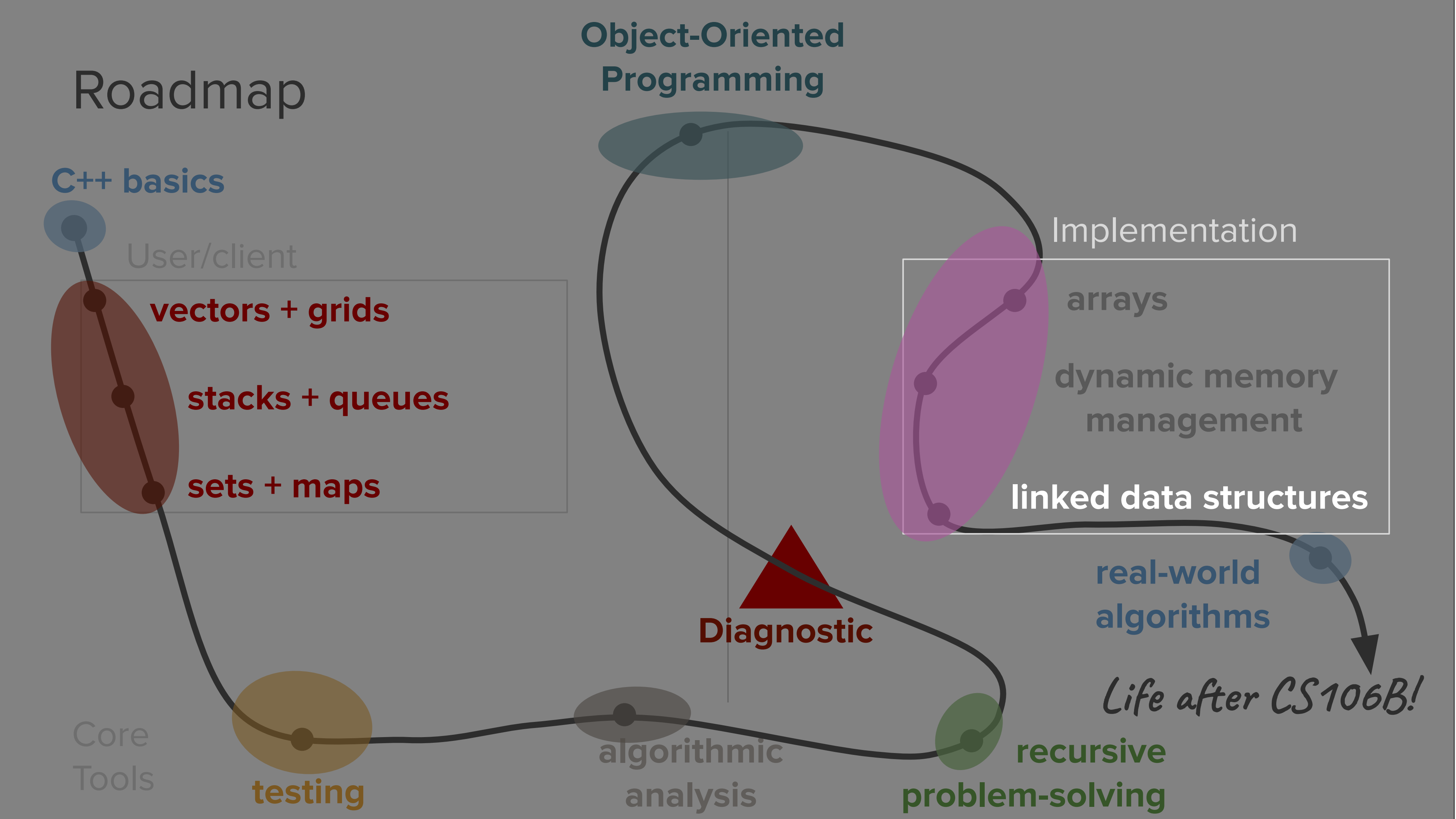
What topic are you interested in investigating for your final project?

(put your answers the chat)

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



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

Today's question

How can we write code to examine and manipulate the structure of linked lists?

Today's topics

1. Review
2. Linked List Traversal
3. Linked List Insertion

Review

[intro to linked lists]

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

Pointers move us across this boundary!

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)

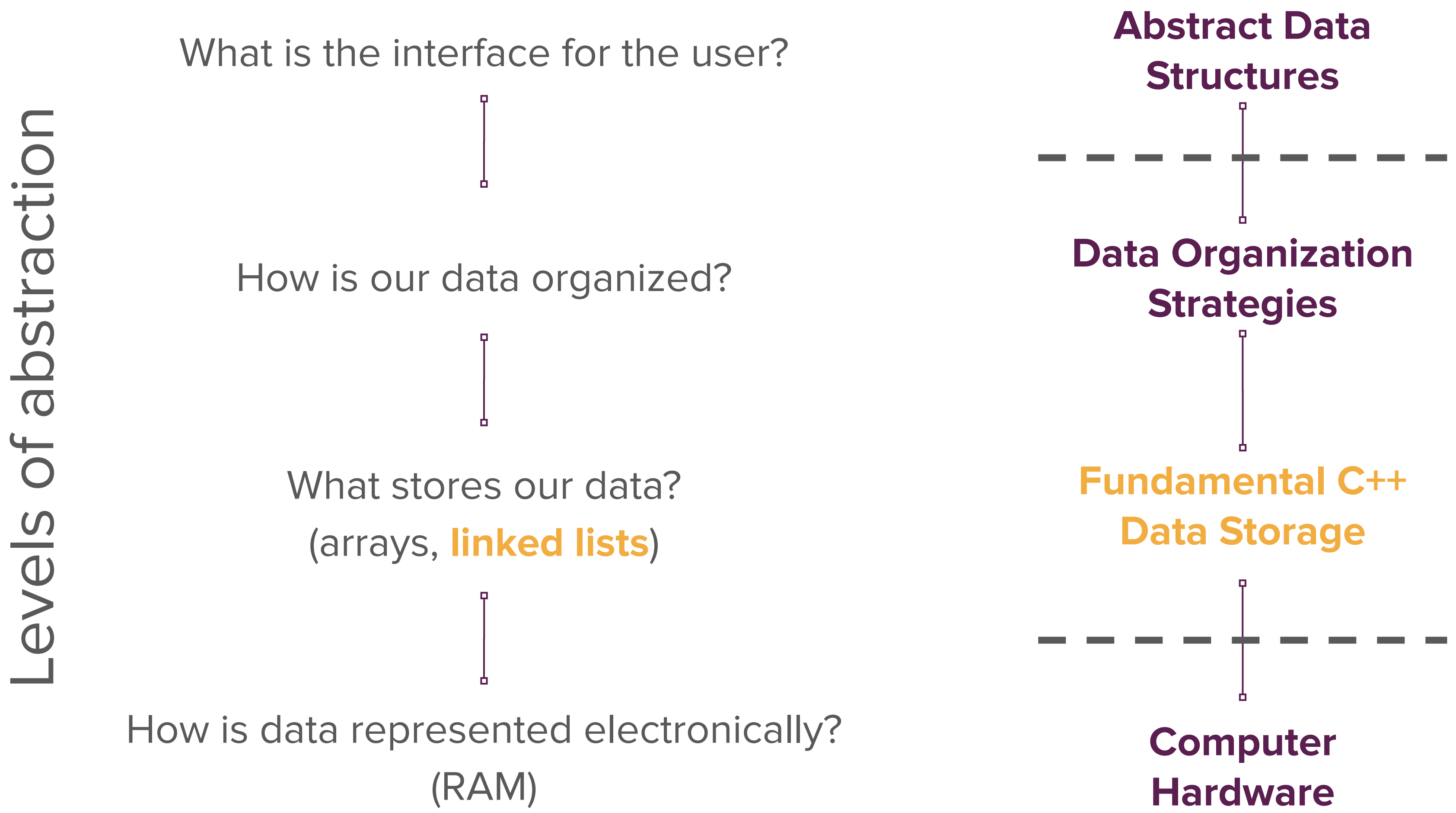
These are built on top of pointers!

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

What is a linked list?

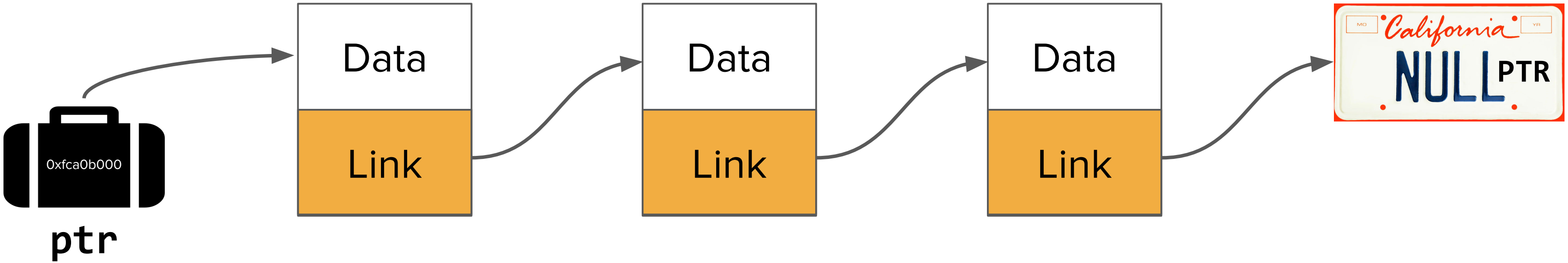
- A linked list is a **chain of nodes**, used to store a sequence of data.

- 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.

- The end of the list is marked with some special indicator.

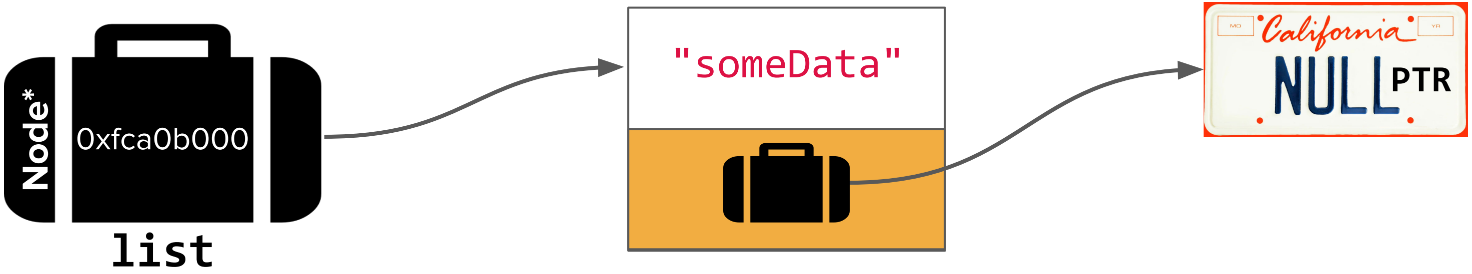
A linked list!



The **Node** struct

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

Pointer to a node

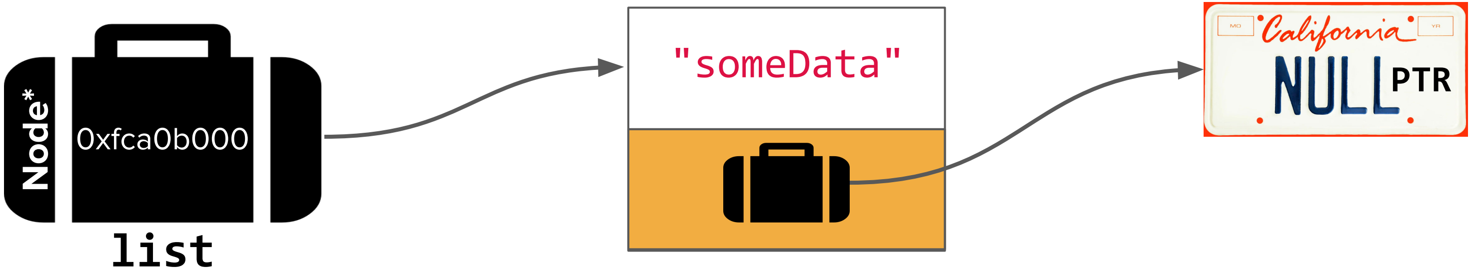



```
Node* list = new Node;
list->data = "someData";
list->next = nullptr;
```

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

New: Node struct constructor

*The Node struct also has a conveniently defined **constructor** that allows us to accomplish this in one line.*




```
Node* list = new Node("someData", nullptr);
```

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 an ADT using a Linked List

- A linked list can be the fundamental data storage backing for an ADT in much the same the same way an array can.

- We saw that linked lists function great as a way of implementing a stack!

- Three operations:
  - **`push()`** – List insertion and list rewiring
  - **`pop()`** – List deletion and list rewiring
  - **Destructor –** List traversal and list deletion

Important 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)

Linked List Operations Revisited

How can we write code to examine and manipulate the structure of linked lists?

Linked Lists Reframed

- On Thursday, we saw linked lists in the context of classes, where we used a linked list as the data storage underlying an implementation of a Stack.

Linked Lists Reframed

- On Thursday, we saw 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 Lists Reframed

- On Thursday, we saw 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 next two days. 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!

Linked List Traversal

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()`

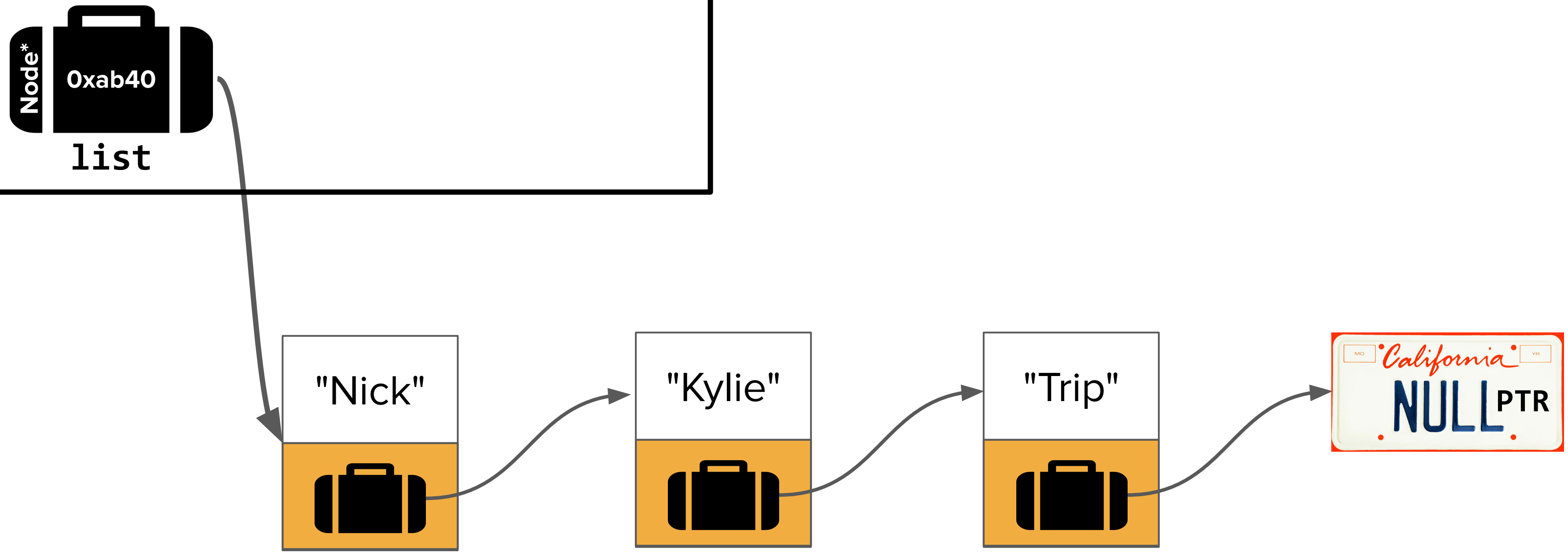
Let's code it!

How does it work?

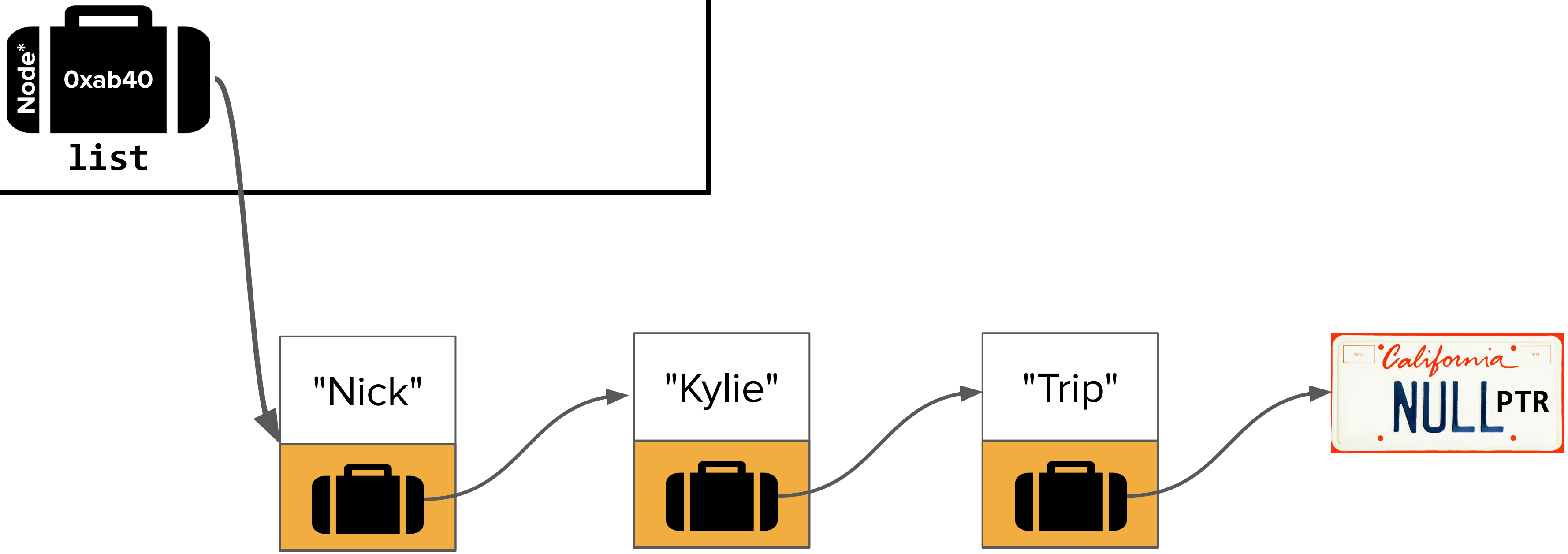
```
int main() {
    Node* list = readList();
    printList(list);

    /* 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... */
}
```

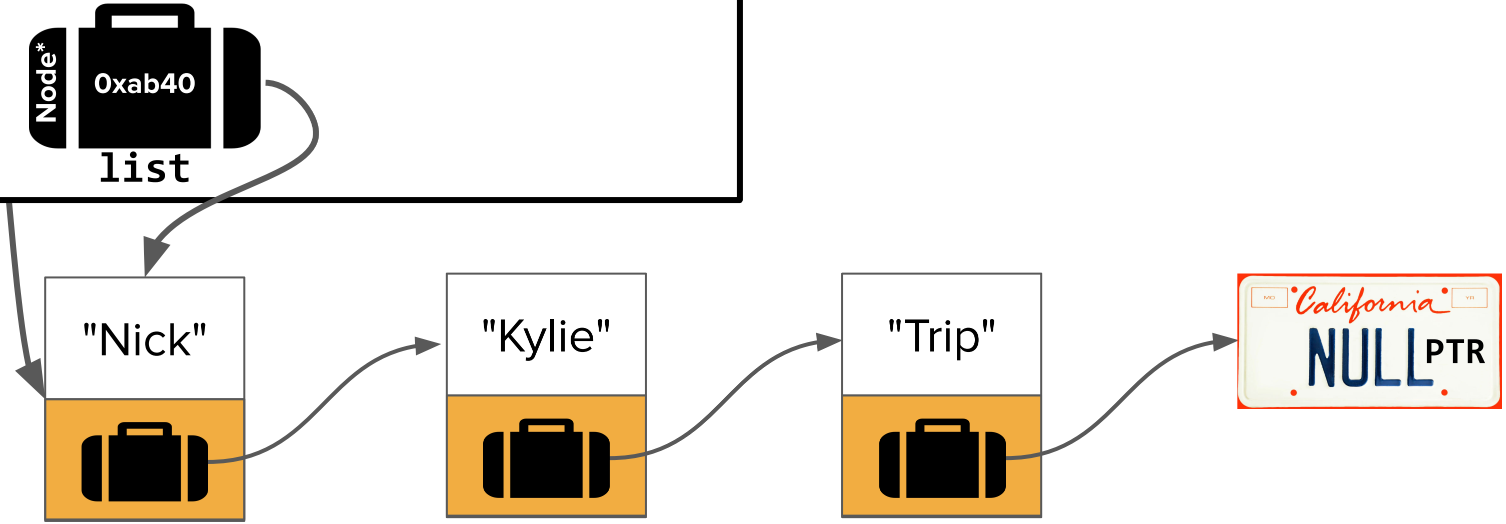


```
int main() {
    Node* list = readList();
    printList(list);

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



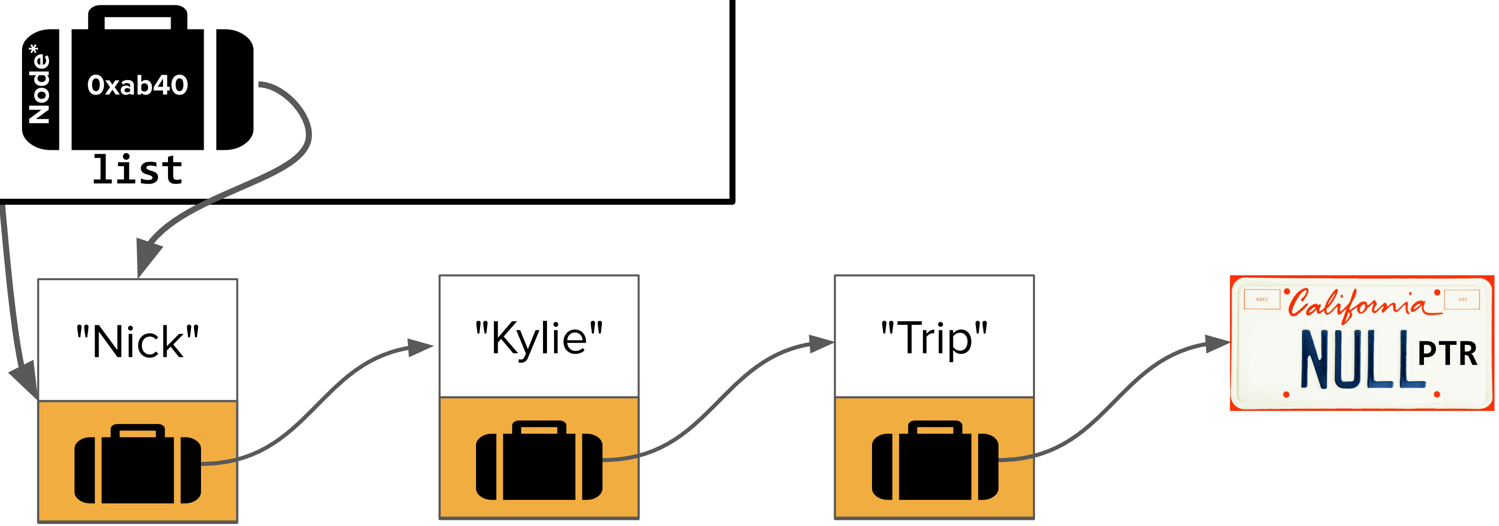
```
int main() {

}
```

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



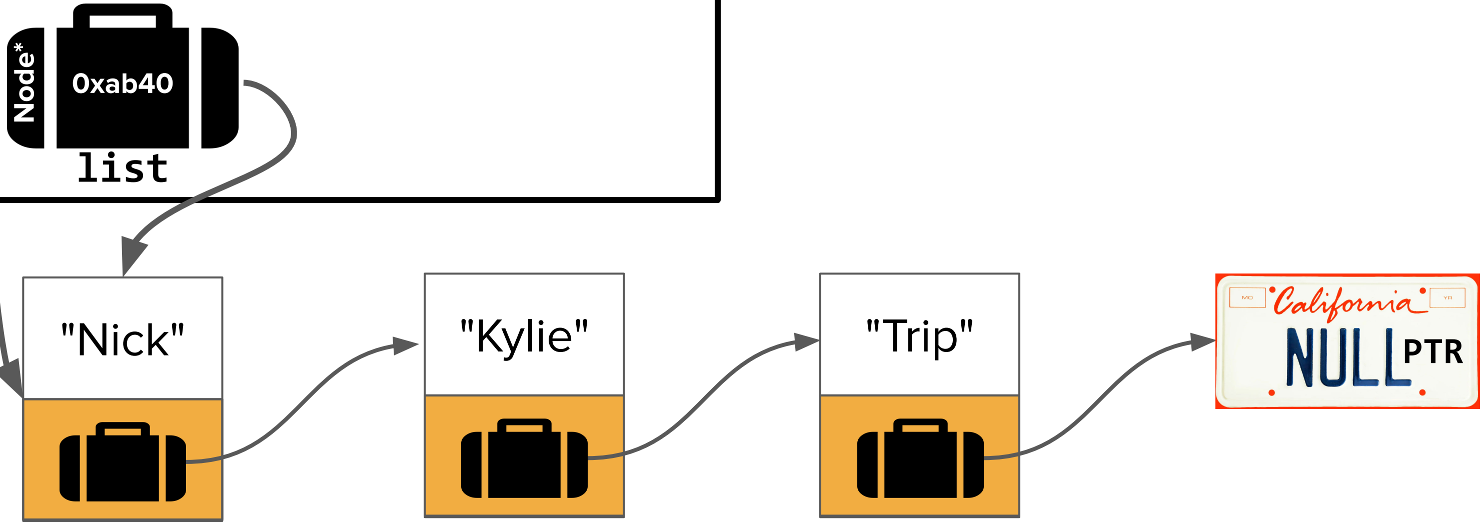
```
int main() {

}
```

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



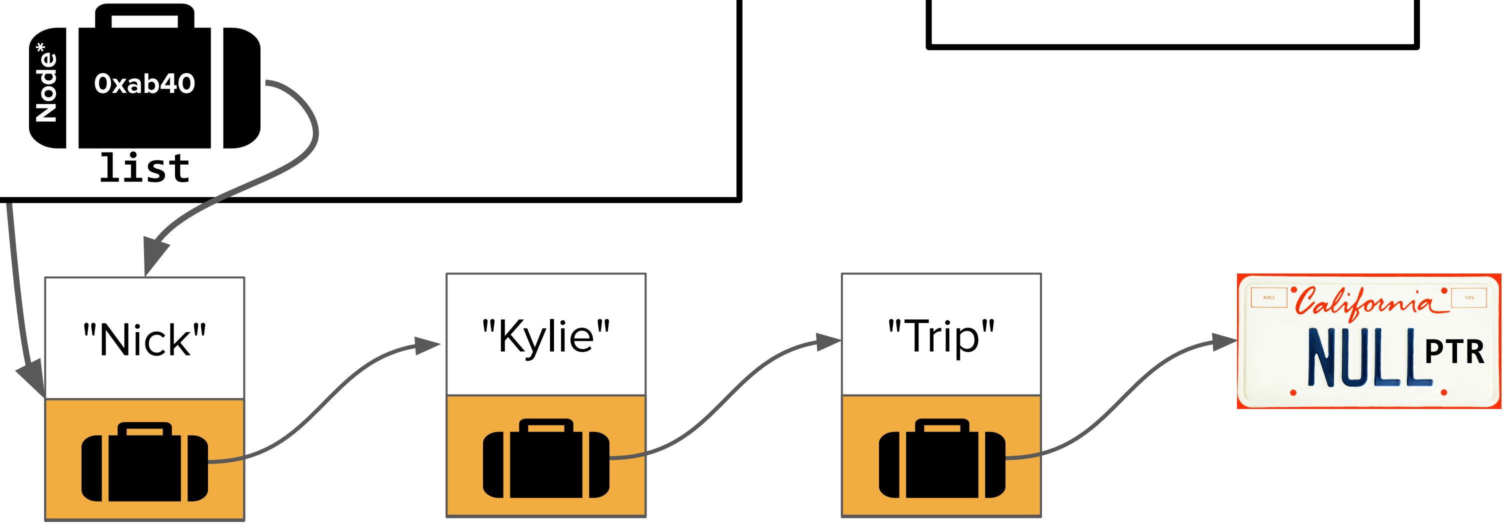
```
int main() {

}
```

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



```
int main() {

}
```

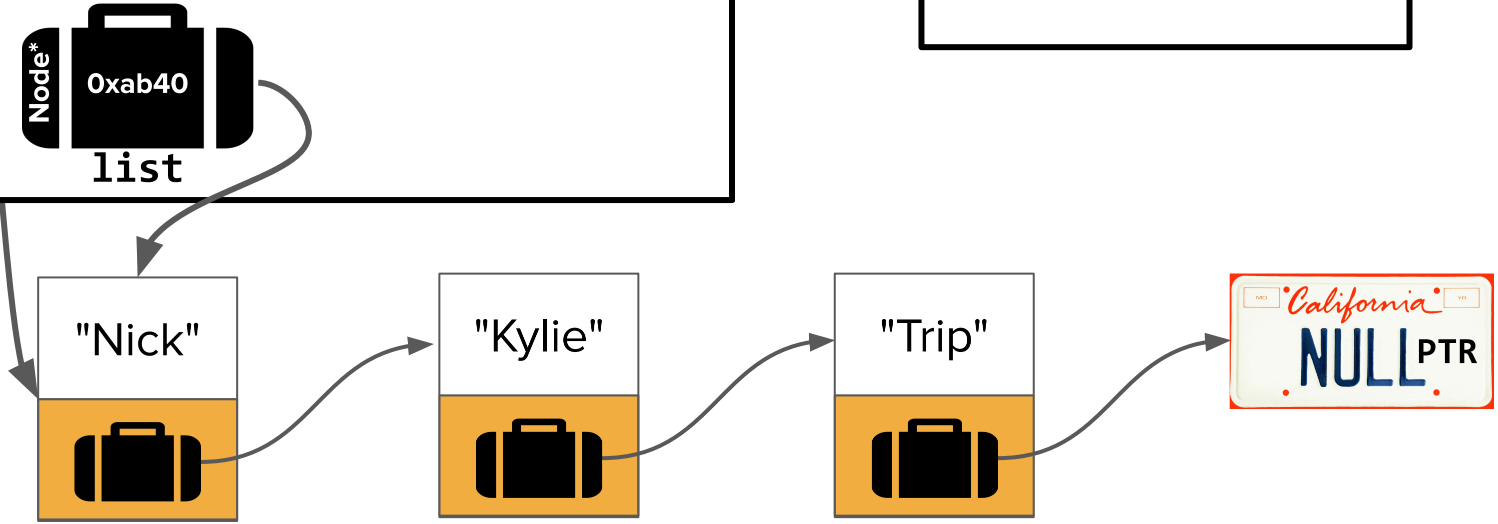
```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```



Nick

```
int main() {

}
```

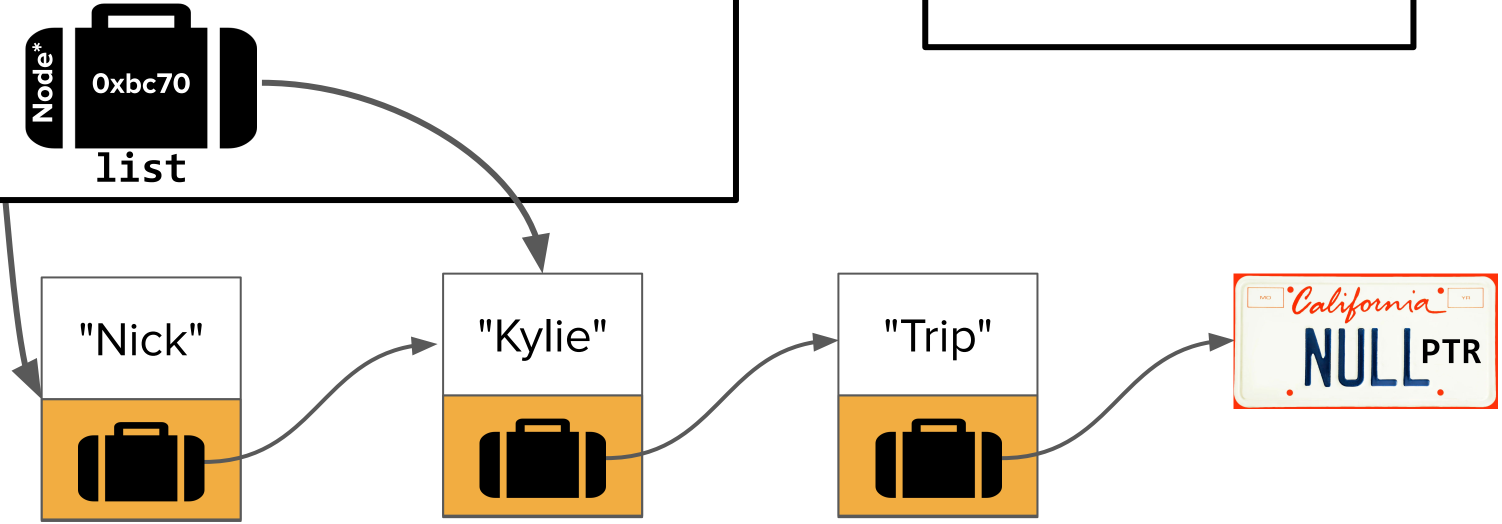
```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```




```
Nick
```

```
int main() {

}
```

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

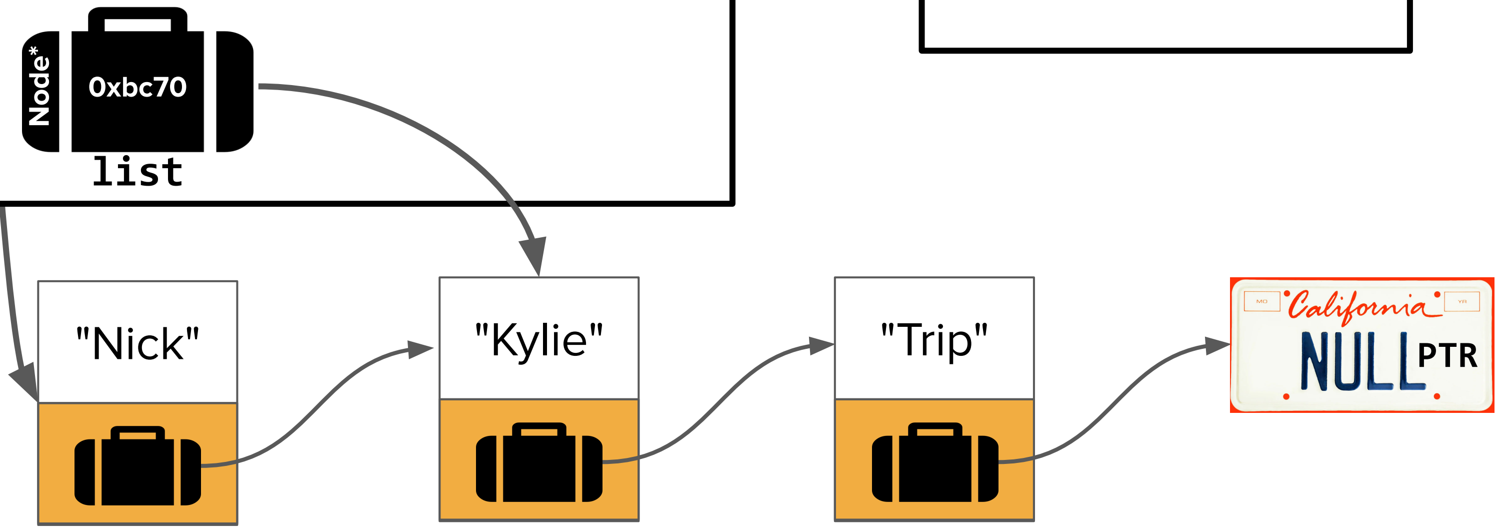



```
Nick
```

```
int main() {

}
```

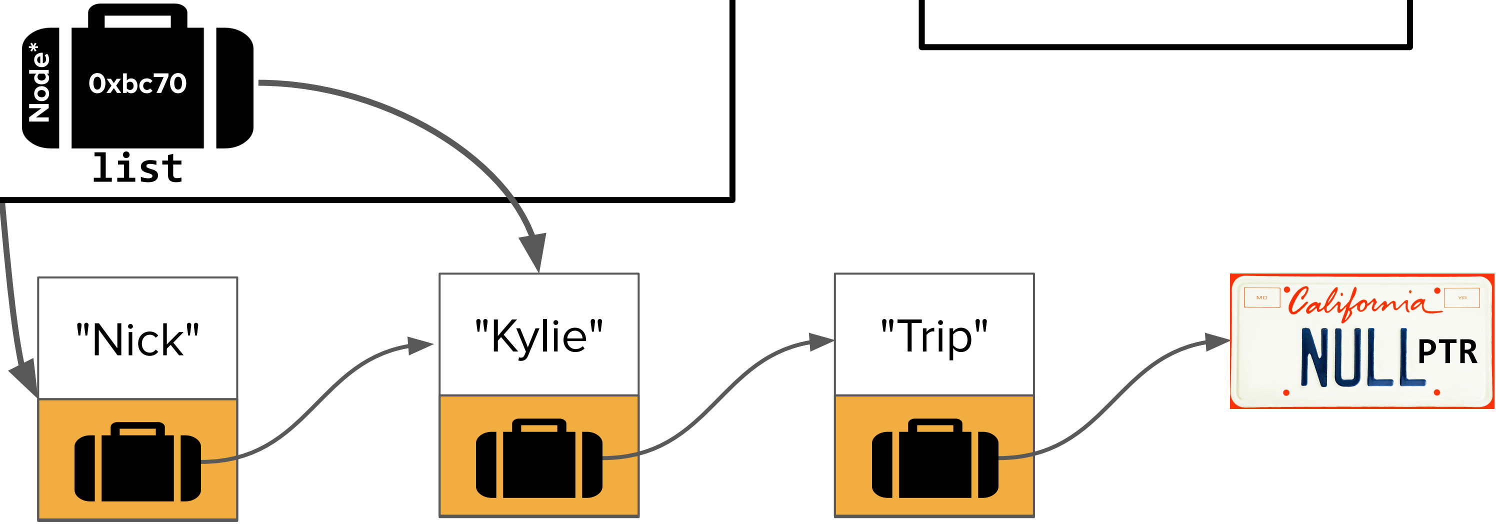
```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```

Nick



```
int main() {

}
```

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

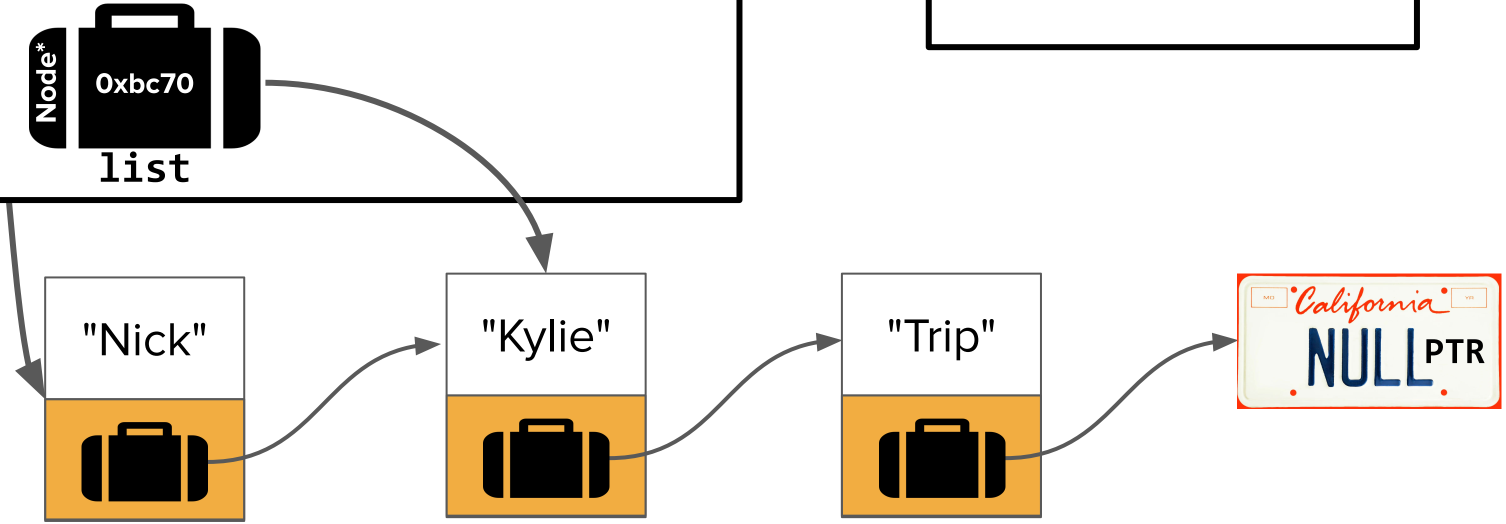


Nick

```
int main() {

}
```

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

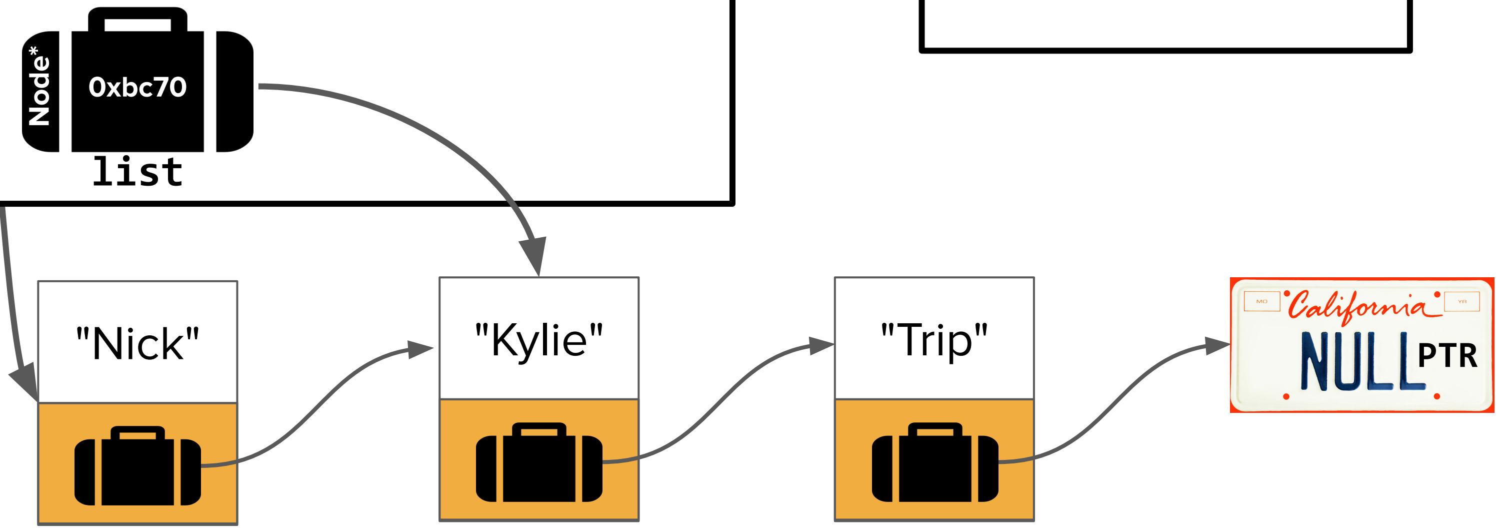
Nick
Kylie



```
int main() {

}
```

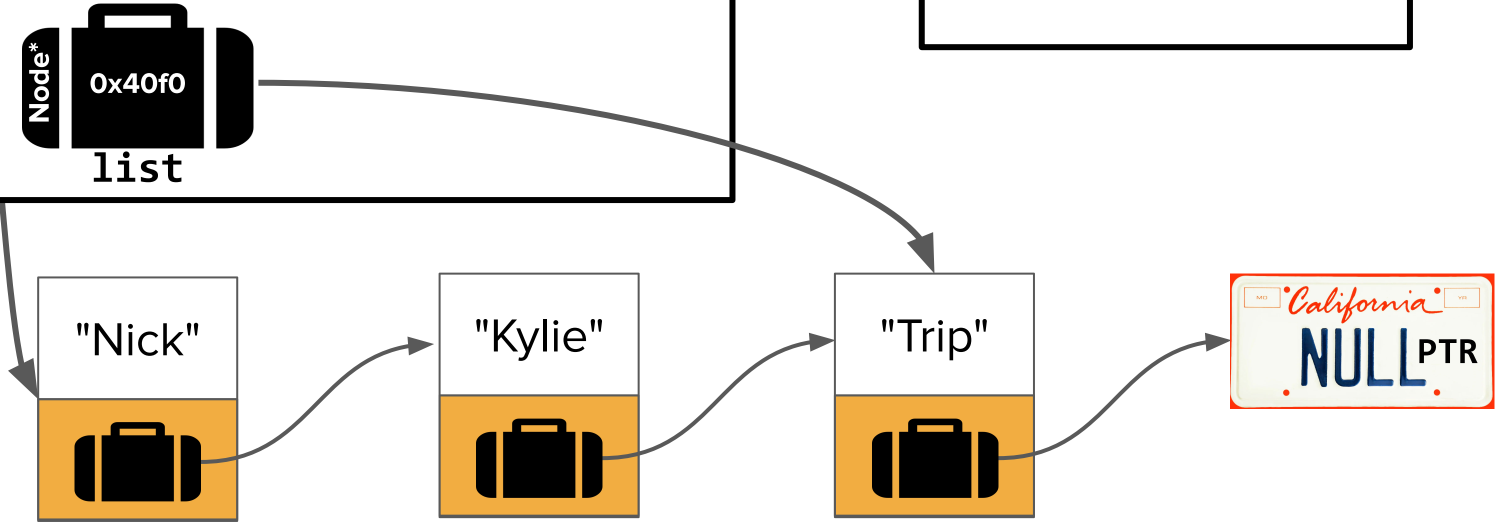
```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```

Nick
Kylie



```
int main() {

}
```

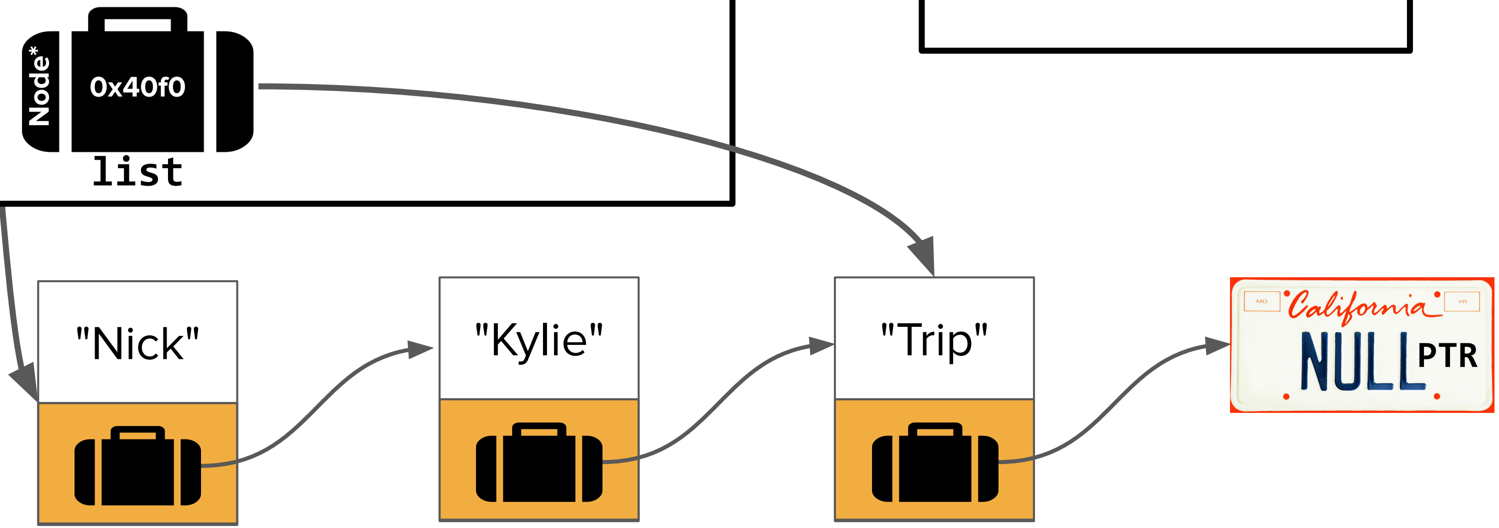
```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```




```
Nick
Kylie
```

```
int main() {

}
```

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

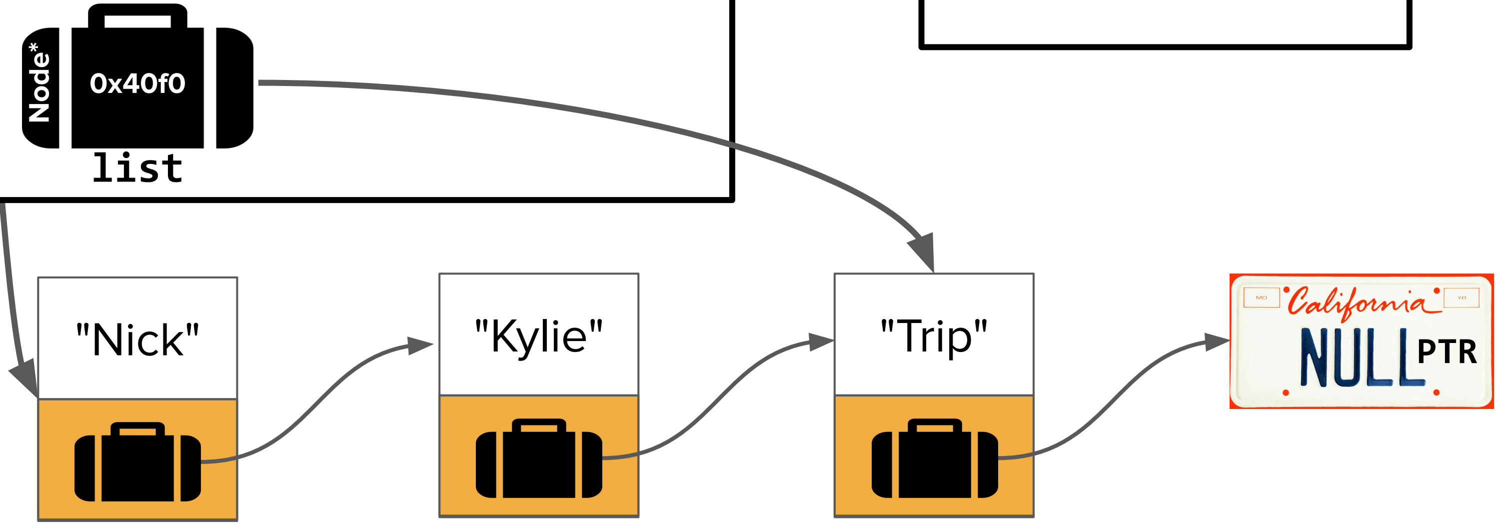



```
Nick
Kylie
```

```
int main() {

}
```

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

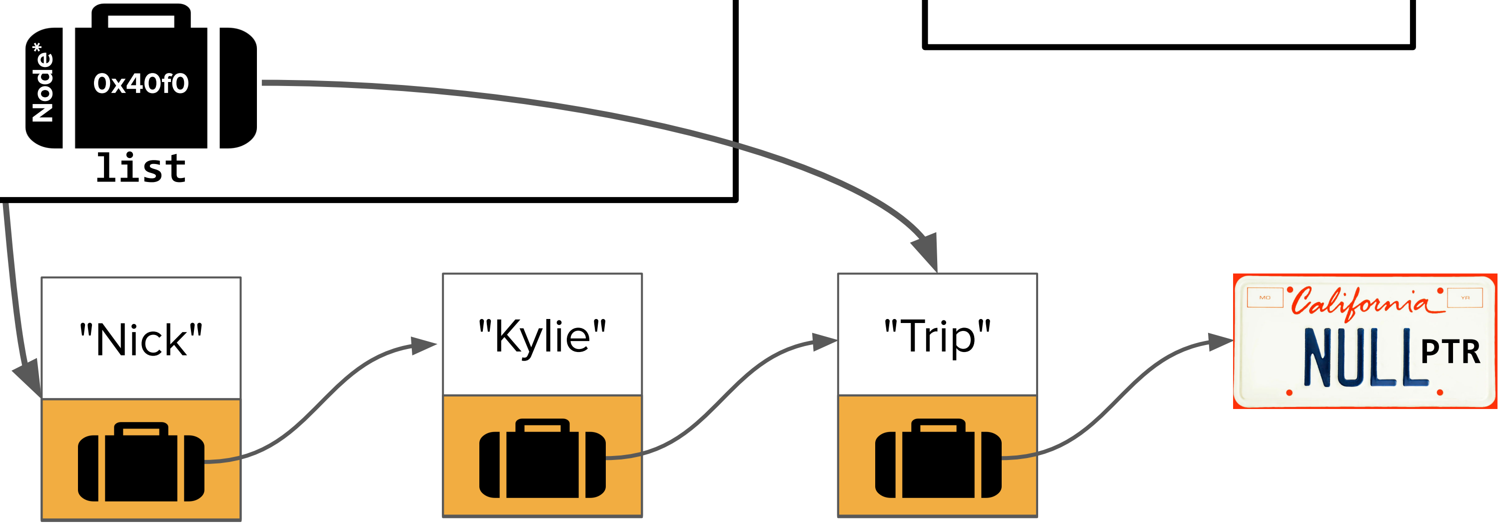
```
Nick
Kylie
```



```
int main() {

}
```

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

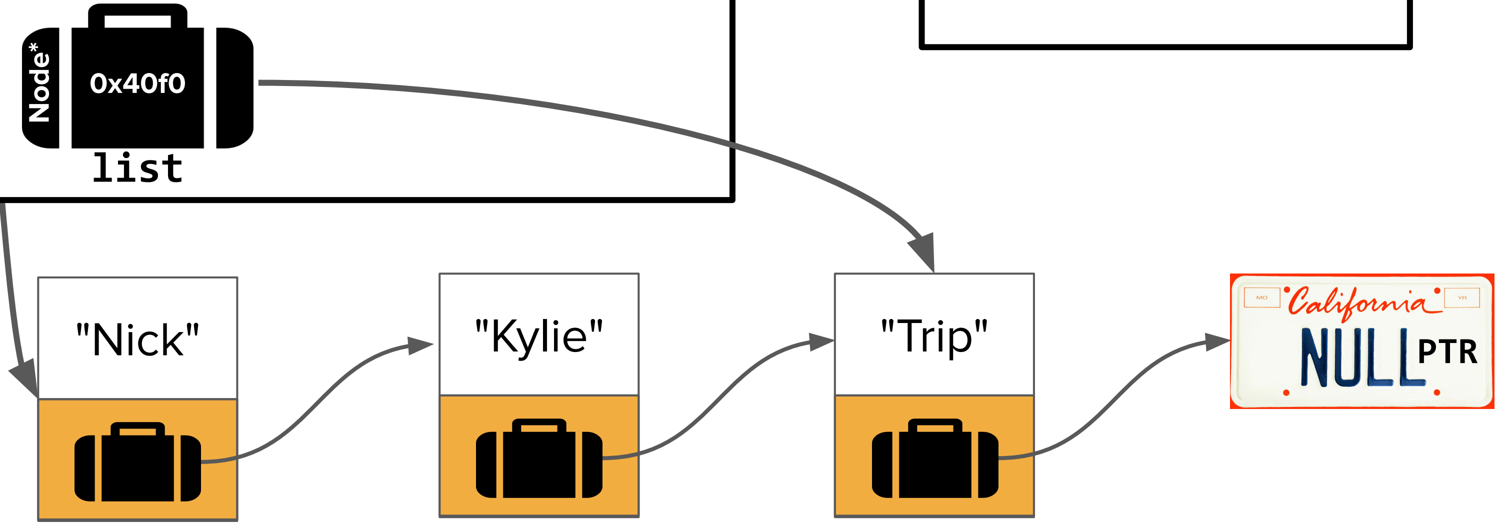
```
Nick
Kylie
Trip
```



```
int main() {

}
```

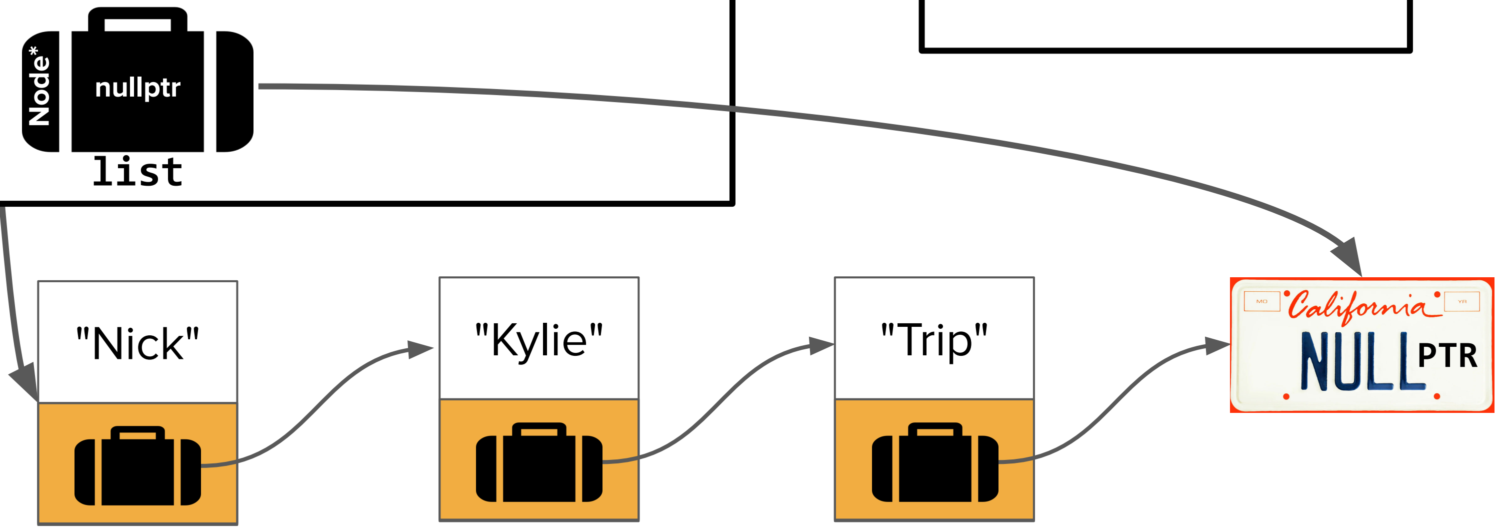
```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```

```
Nick
Kylie
Trip
```



```
int main() {

}
```

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

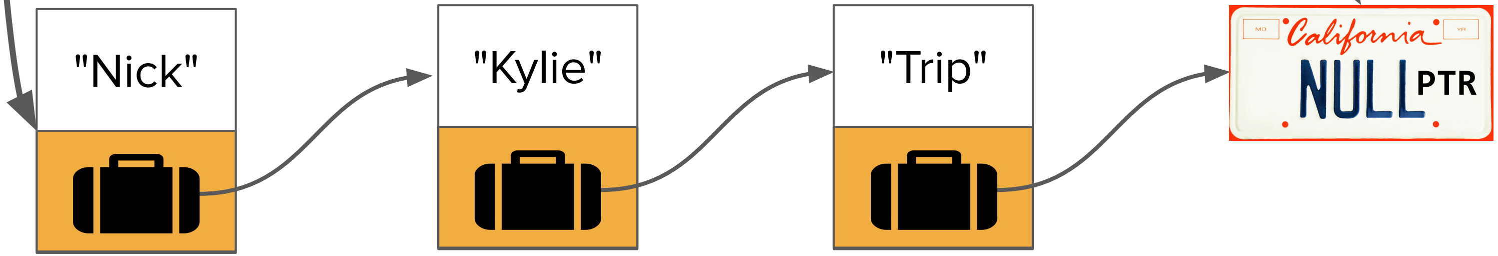



```
Nick
Kylie
Trip
```

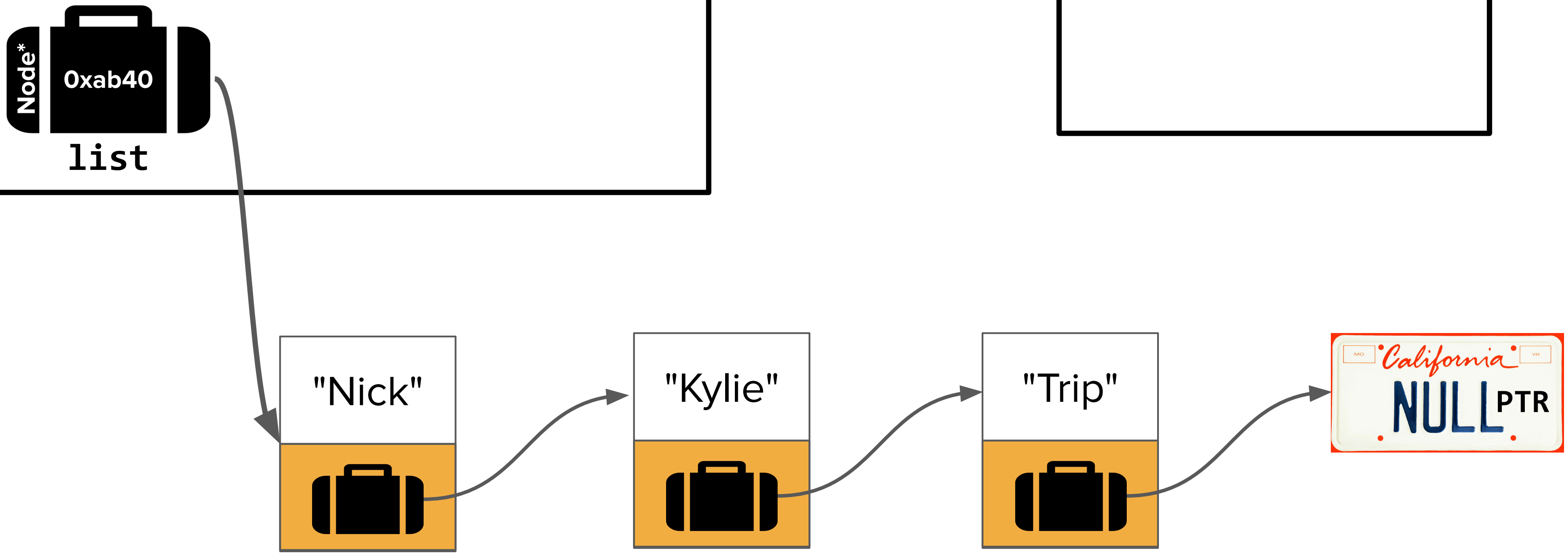
```
int main() {

}
```

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


```
Nick
Kylie
Trip
```

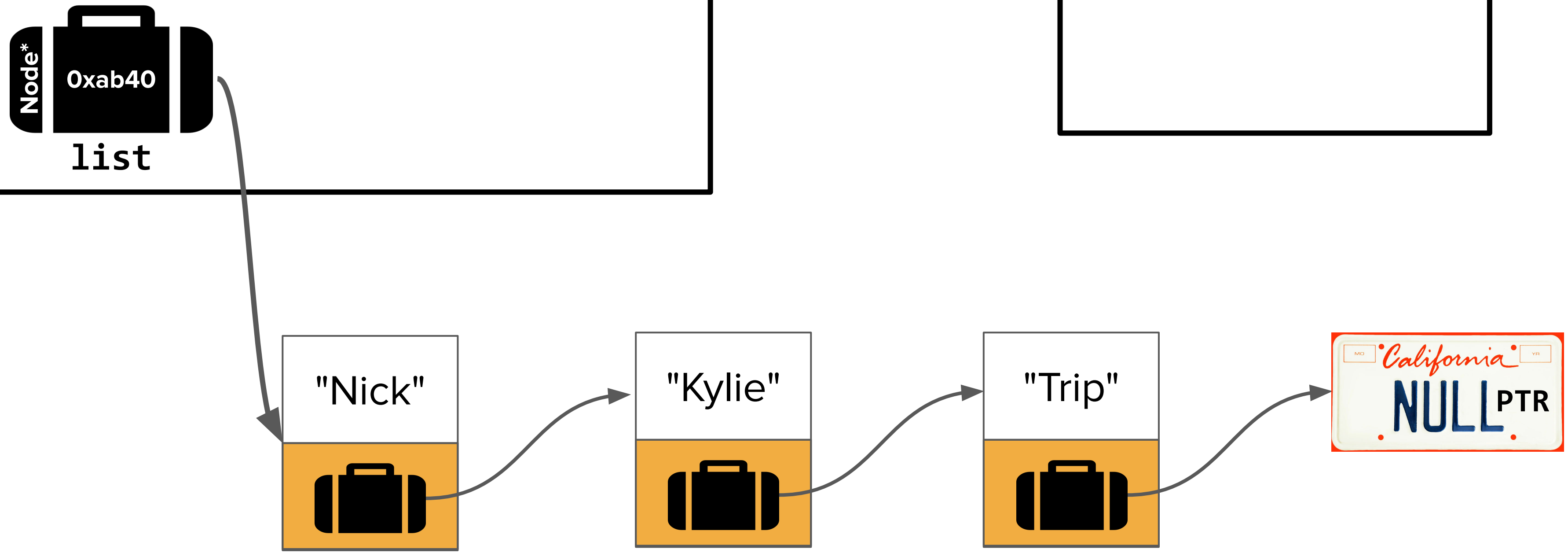


```
int main() {
    Node* list = readList();
    printList(list);

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




```
Nick
Kylie
Trip
```

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

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




```
Nick
Kylie
Trip
```

Measuring a Linked List

Measuring a Linked List

- Similar to arrays, a linked list does not have the capability to automatically report back its own "size."
- The following code is NOT valid, since list is simply a pointer

```
Node* list = readList();
cout << list.size() << endl; // WRONG! BAD!
```

- Let's write a function that allows us to calculate the number of nodes in a linked list!

`lengthOf()`

Let's code it!

Freeing a Linked List

Freeing Linked Lists

- Linked lists are built out of many different nodes, each of which have been **dynamically allocated**. This means that when we're done using a list, it is always good practice to free the memory associated with all the nodes!

Freeing Linked Lists

- Linked lists are built out of many different nodes, each of which have been dynamically allocated. This means that when we're done using a list, it is always good practice to free the memory associated with all the nodes!

- Freeing all the nodes requires **traversing the list** while safely freeing everything along the way.

Freeing Linked Lists

- Linked lists are built out of many different nodes, each of which have been dynamically allocated. This means that when we're done using a list, it is always good practice to free the memory associated with all the nodes!

- Freeing all the nodes requires traversing the list while safely freeing everything along the way.

- We've actually seen how to do this already! The **`IntStack`** destructor that we coded up together was responsible for cleaning up all the list memory.

Freeing Linked Lists

- Linked lists are built out of many different nodes, each of which have been dynamically allocated. This means that when we're done using a list, it is always good practice to free the memory associated with all the nodes!

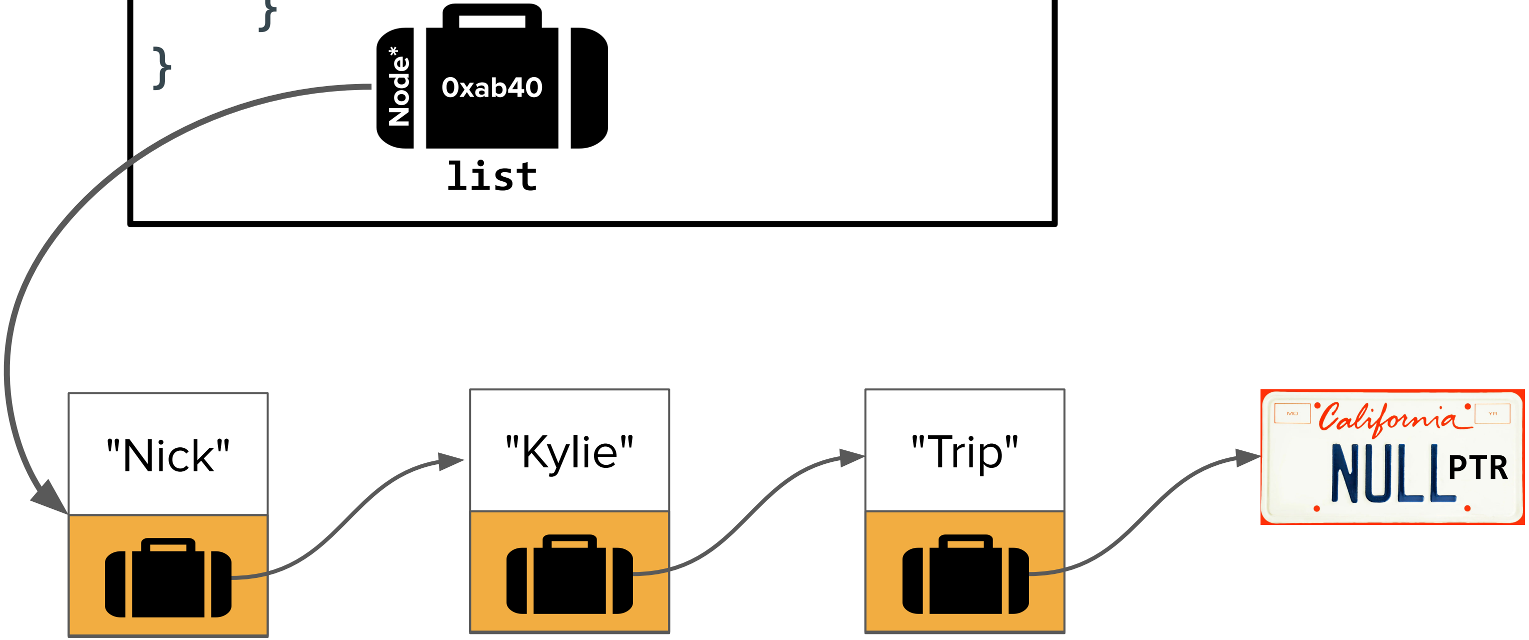
- Freeing all the nodes requires traversing the list while safely freeing everything along the way.

- We've actually seen how to do this already! The `IntStack` destructor that we coded up together was responsible for cleaning up all the list memory.

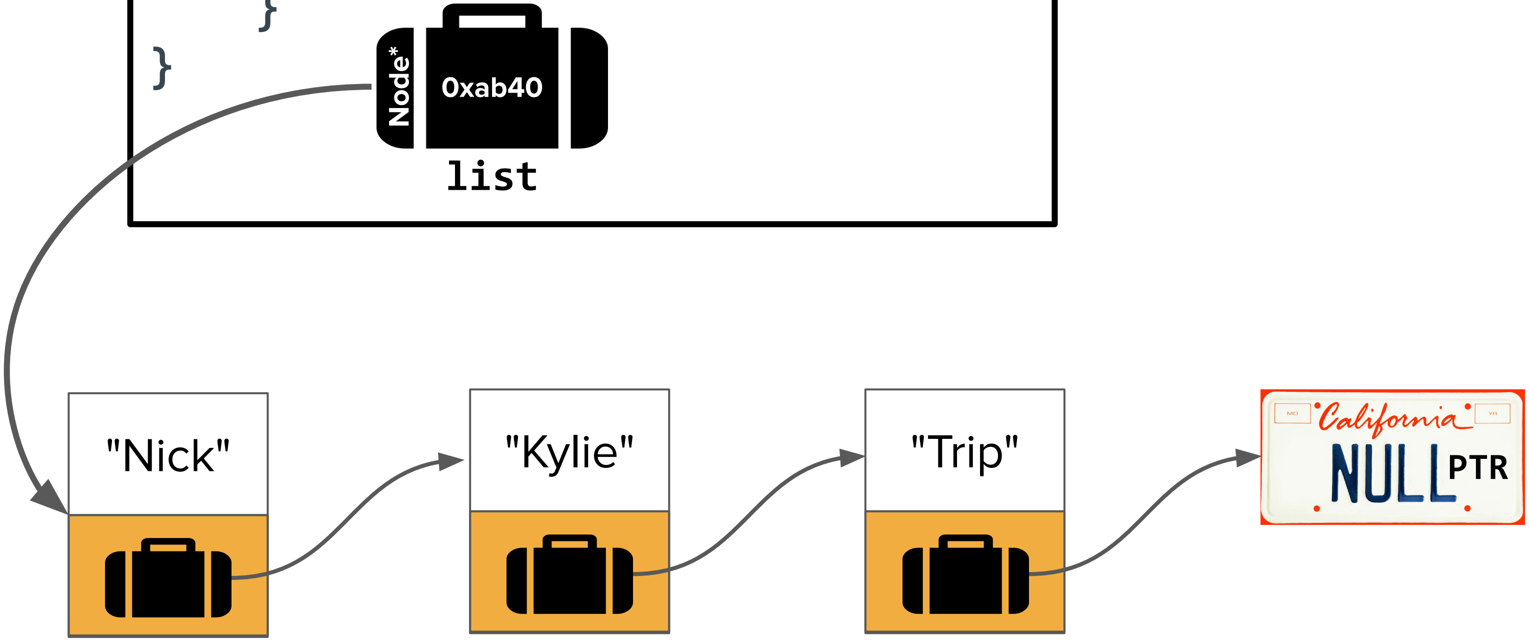
- Let's revisit how to (and how not to) accomplish this task!

Freeing Linked Lists, the Wrong Way

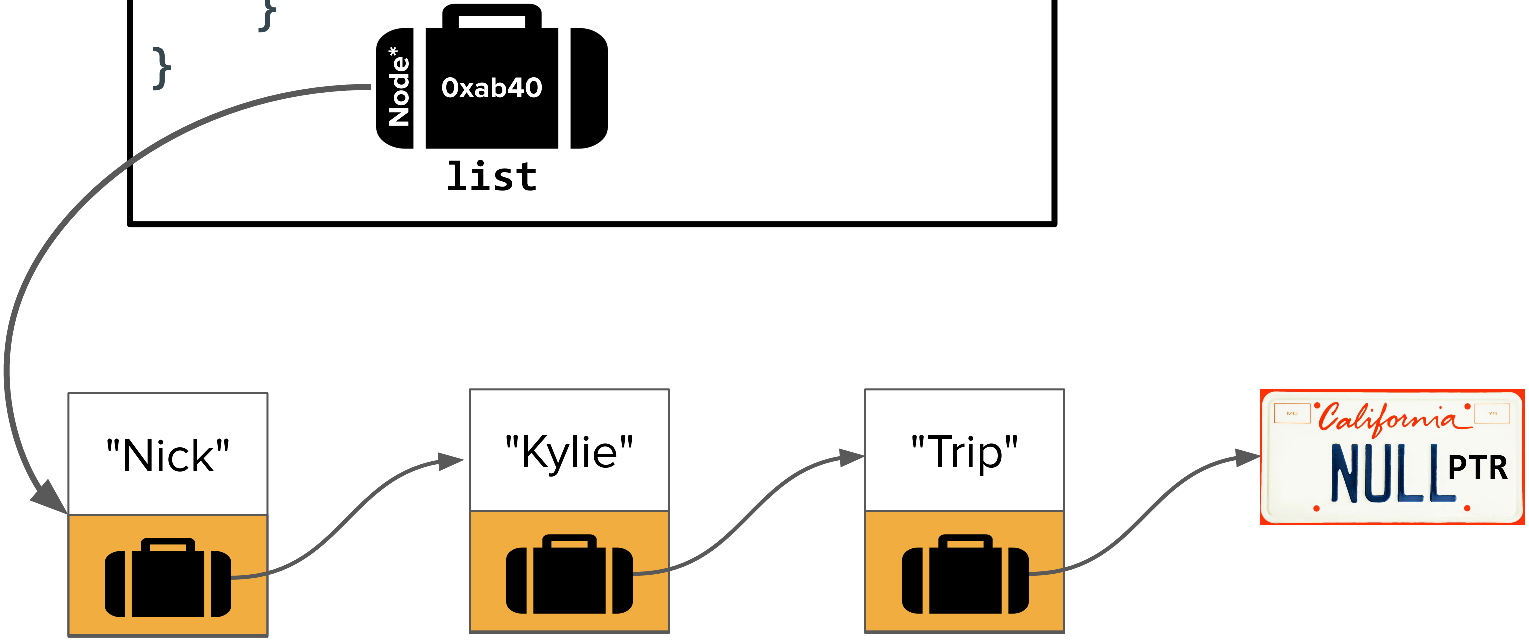
```
void freeList(Node* list) {
    /* WRONG WRONG WRONG WRONG WRONG */
    while (list != nullptr) {
        delete list;
        list = list->next;
    }
}
```



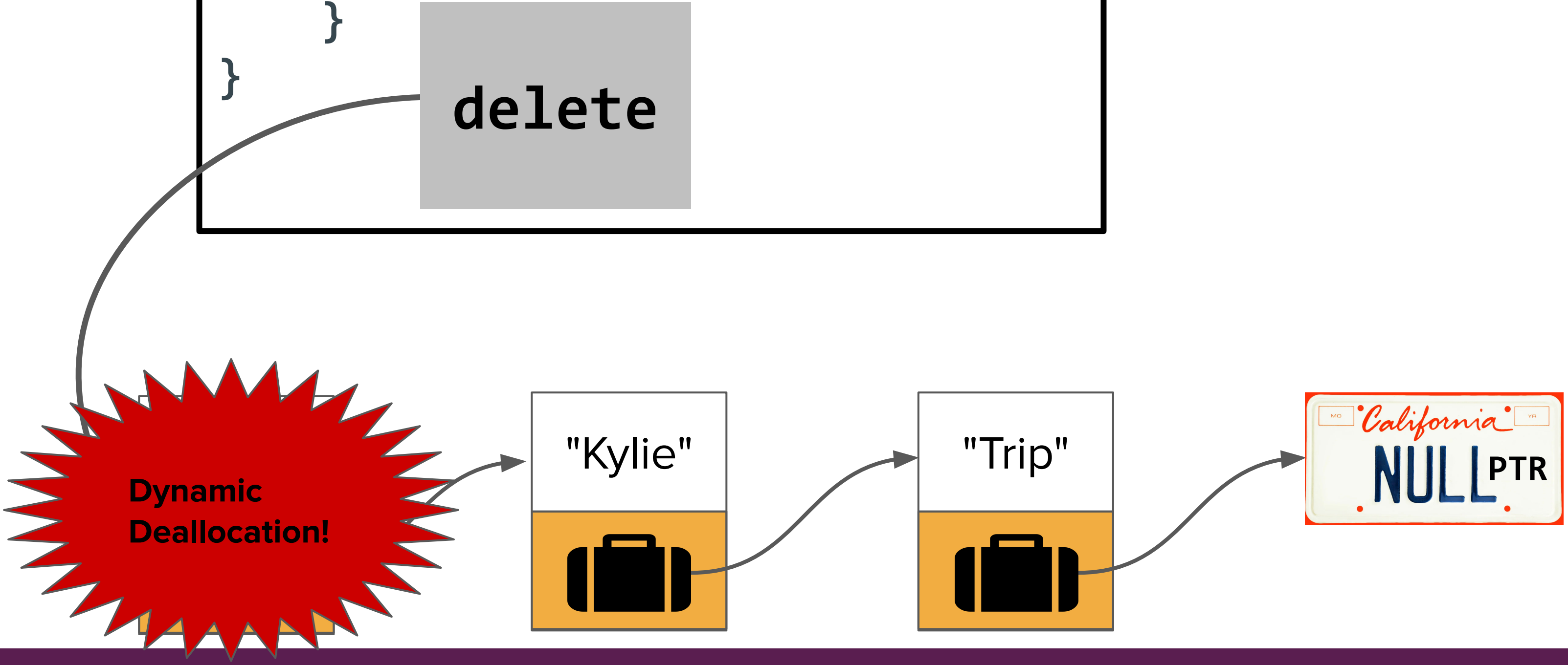
```
void freeList(Node* list) {
    /* WRONG WRONG WRONG WRONG WRONG */
    while (list != nullptr) {
        delete list;
        list = list->next;
    }
}
```



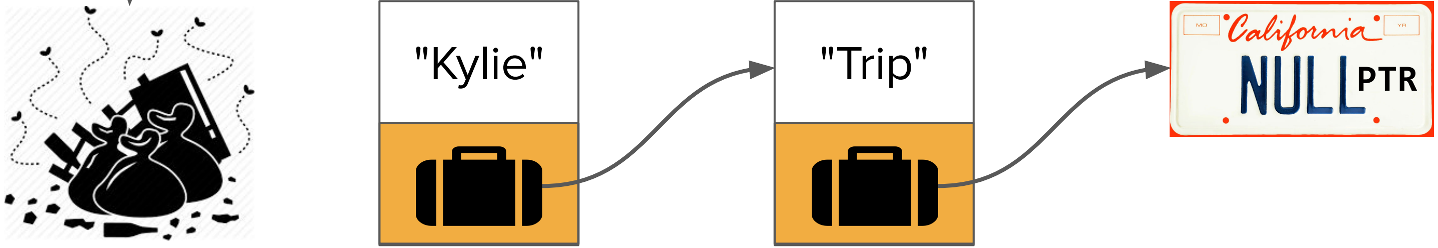
```
void freeList(Node* list) {
    /* WRONG WRONG WRONG WRONG WRONG */
    while (list != nullptr) {
        delete list;
        list = list->next;
    }
}
```



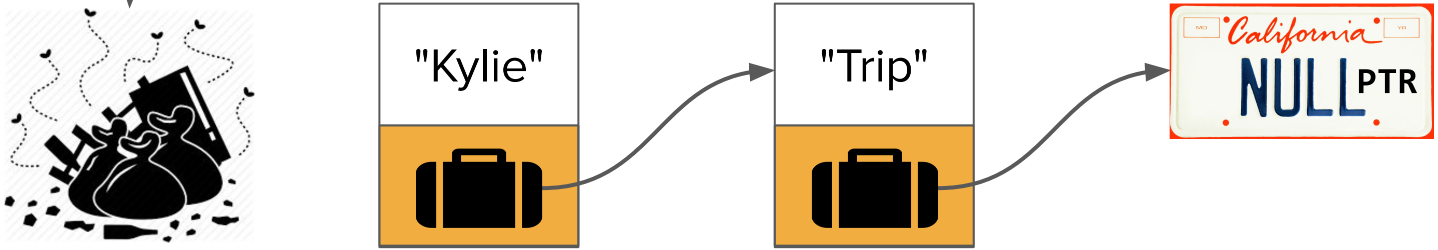
```
void freeList(Node* list) {
    /* WRONG WRONG WRONG WRONG WRONG */
    while (list != nullptr) {
        delete list;
        list = list->next;
    }
}
```



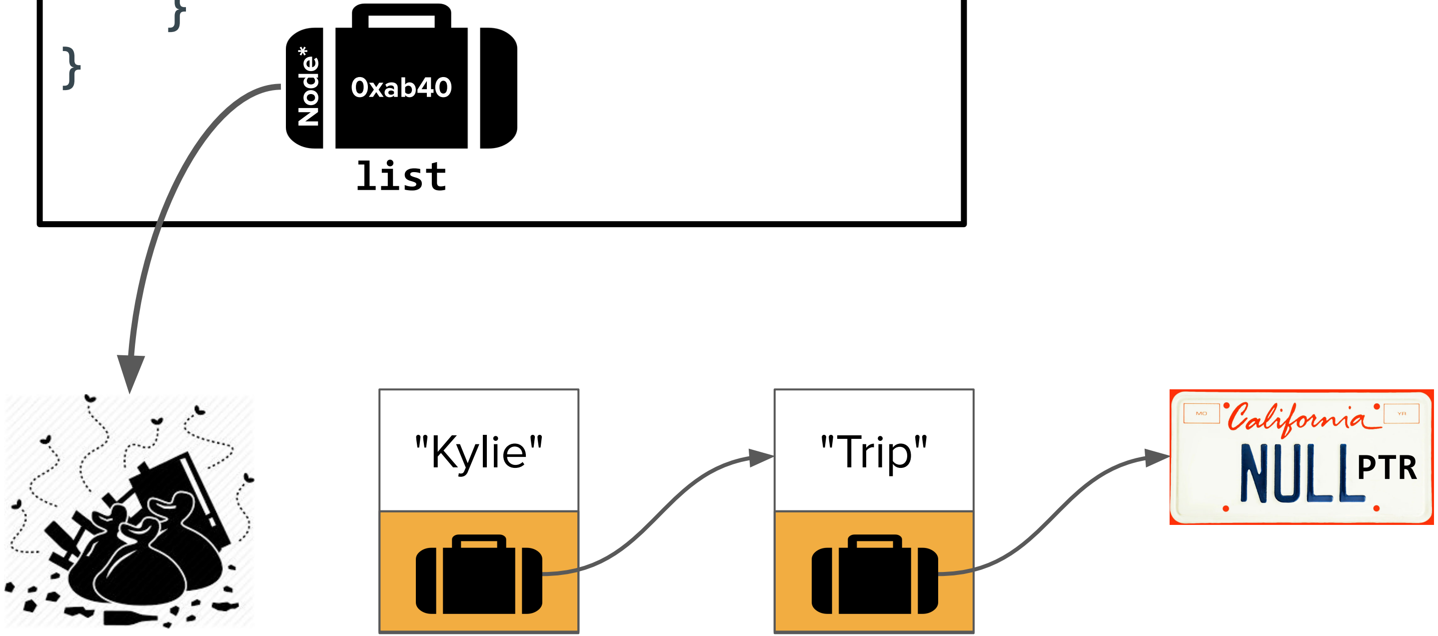
```
void freeList(Node* list) {
    /* WRONG WRONG WRONG WRONG WRONG */
    while (list != nullptr) {
        delete list;
        list = list->next;
    }
}
```



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

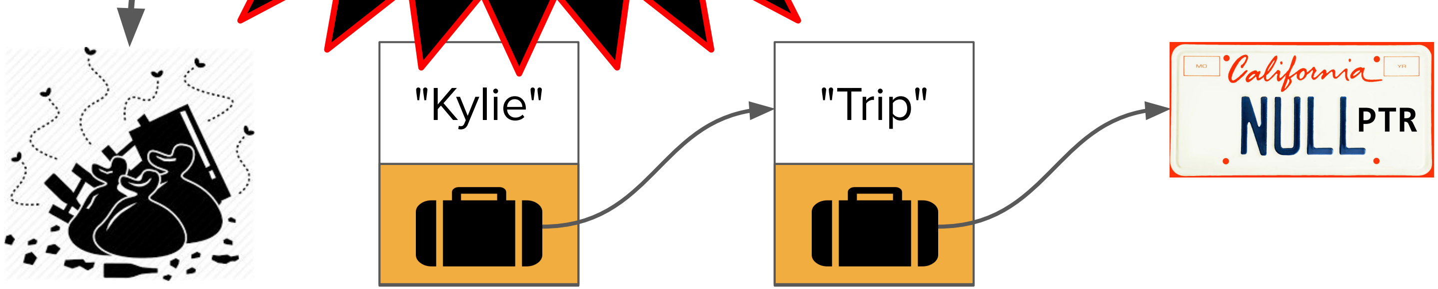


```
void freeList(Node* list) {
    /* WRONG WRONG WRONG WRONG WRONG */
    while (list != nullptr) {
        delete list;
        list = list->next;
    }
}
```



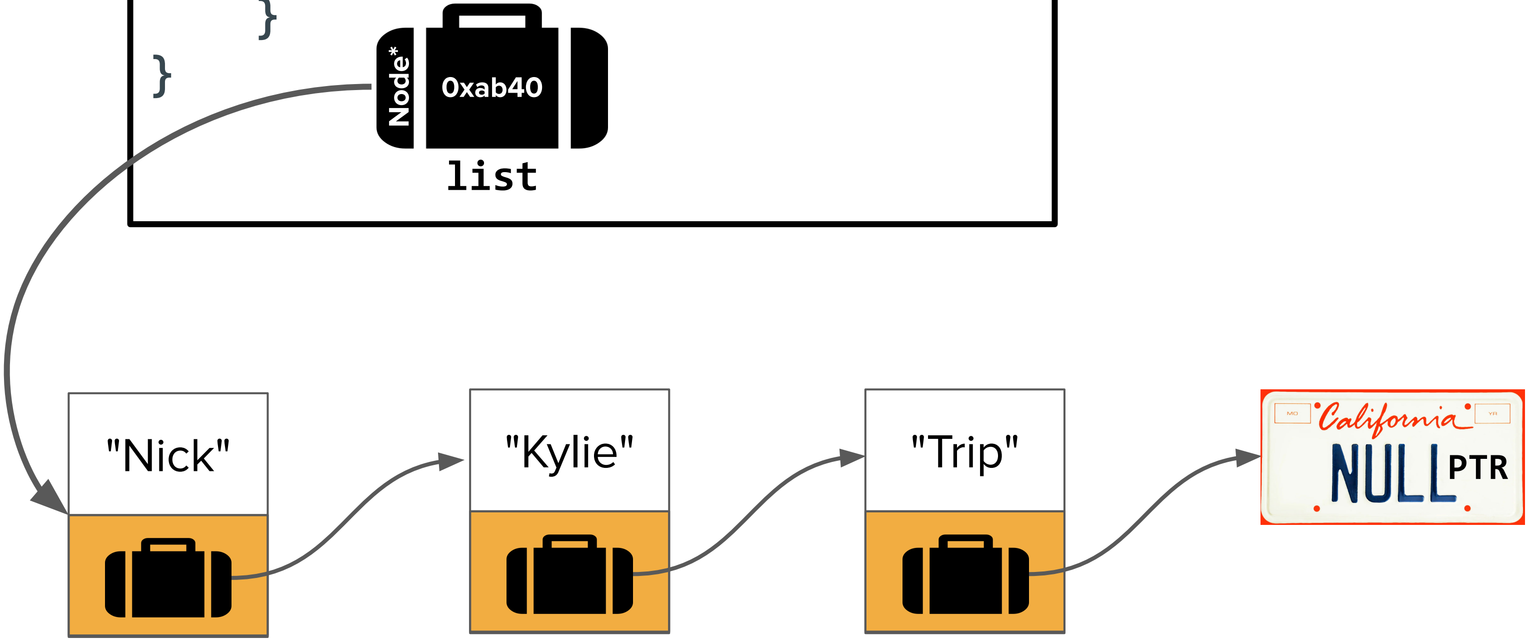
```
void freeList(Node* list) {
    /* WRONG WRONG WRONG WRONG WRONG */
    while (list != nullptr) {
}
```

Undefined Behavior!

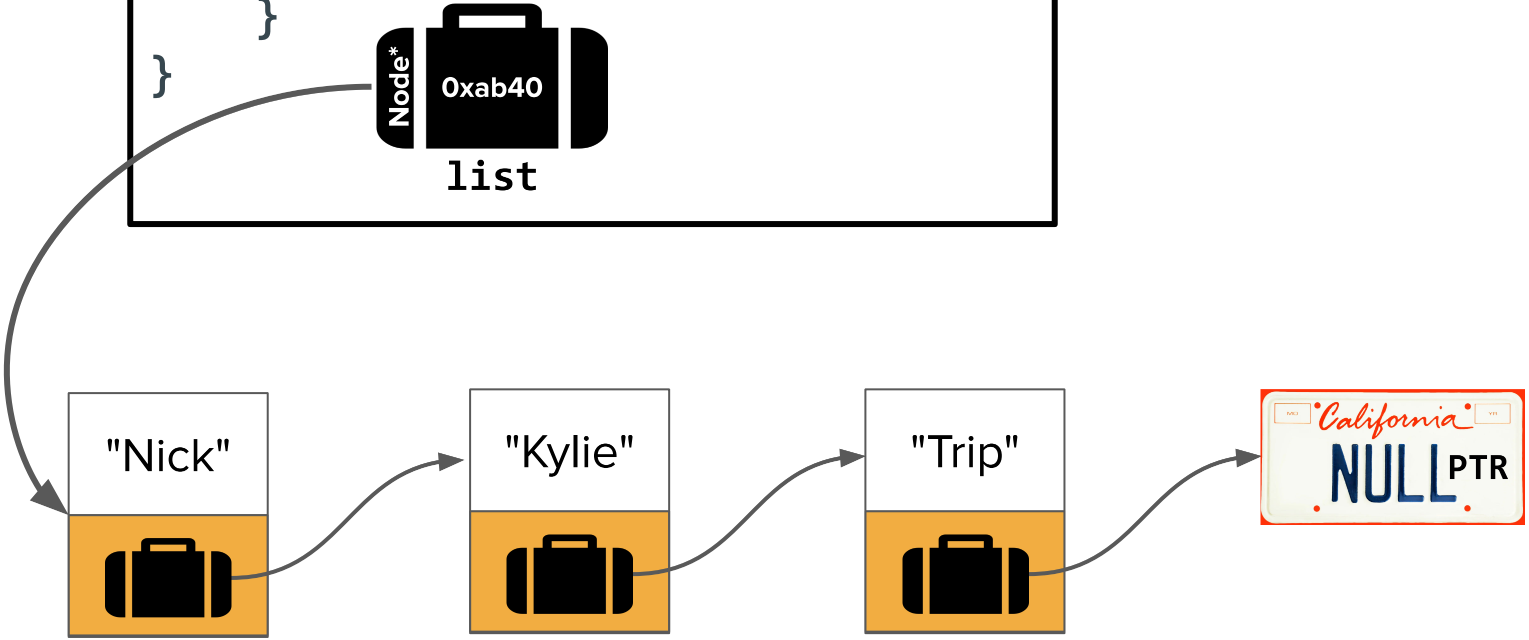


Freeing Linked Lists, the Right Way

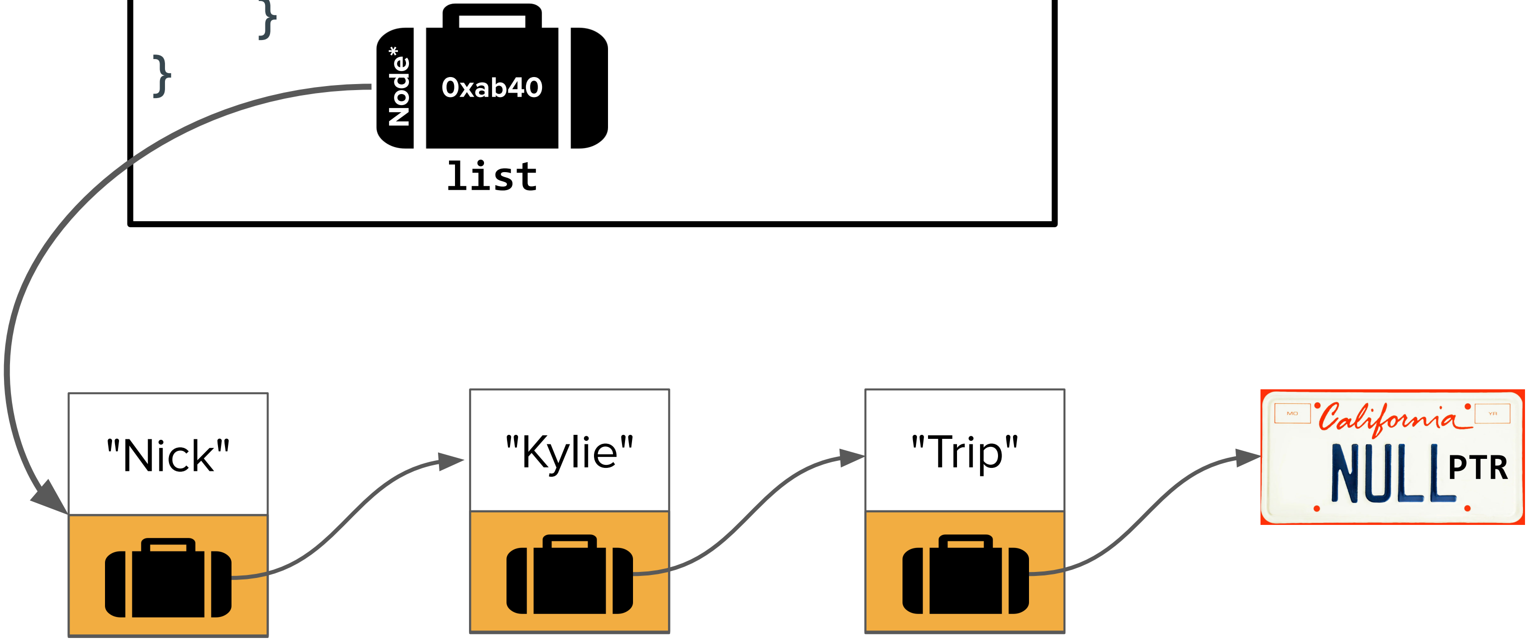
```
void freeList(Node* list) {
    while (list != nullptr) {

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



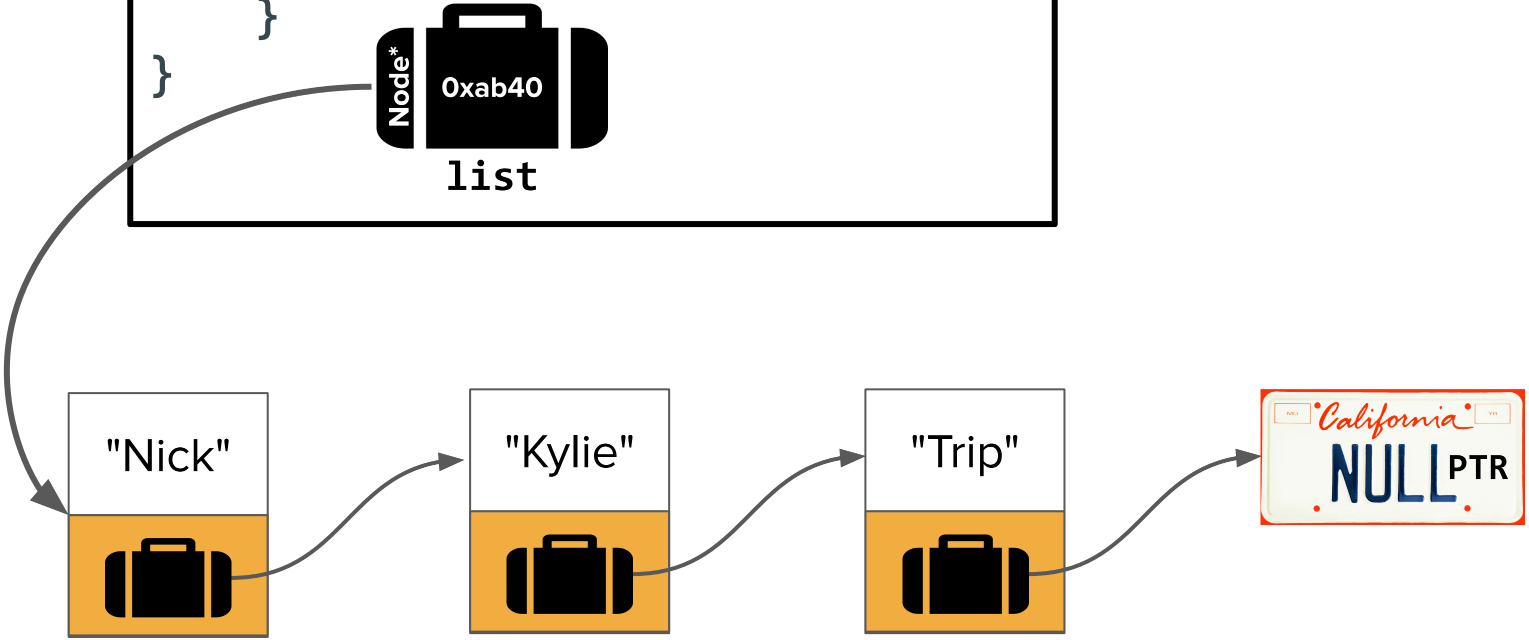
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = list->next;
    }
}
```



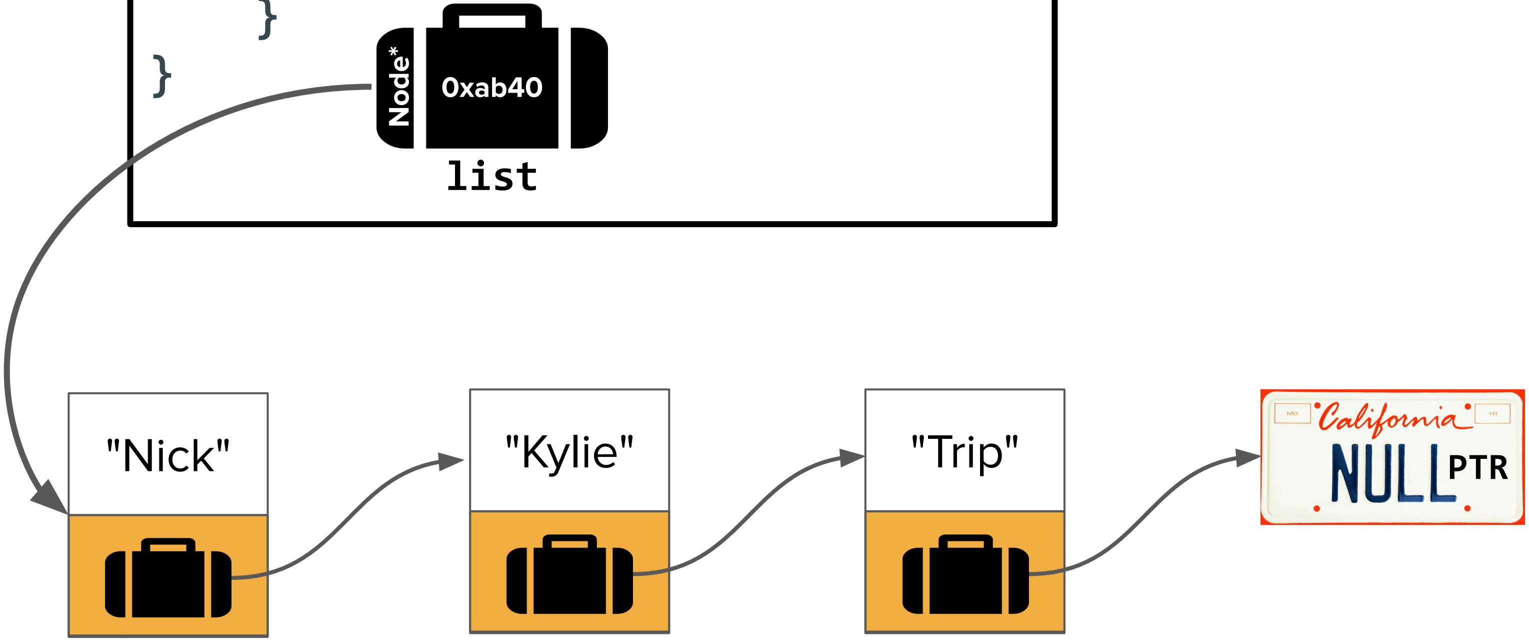
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



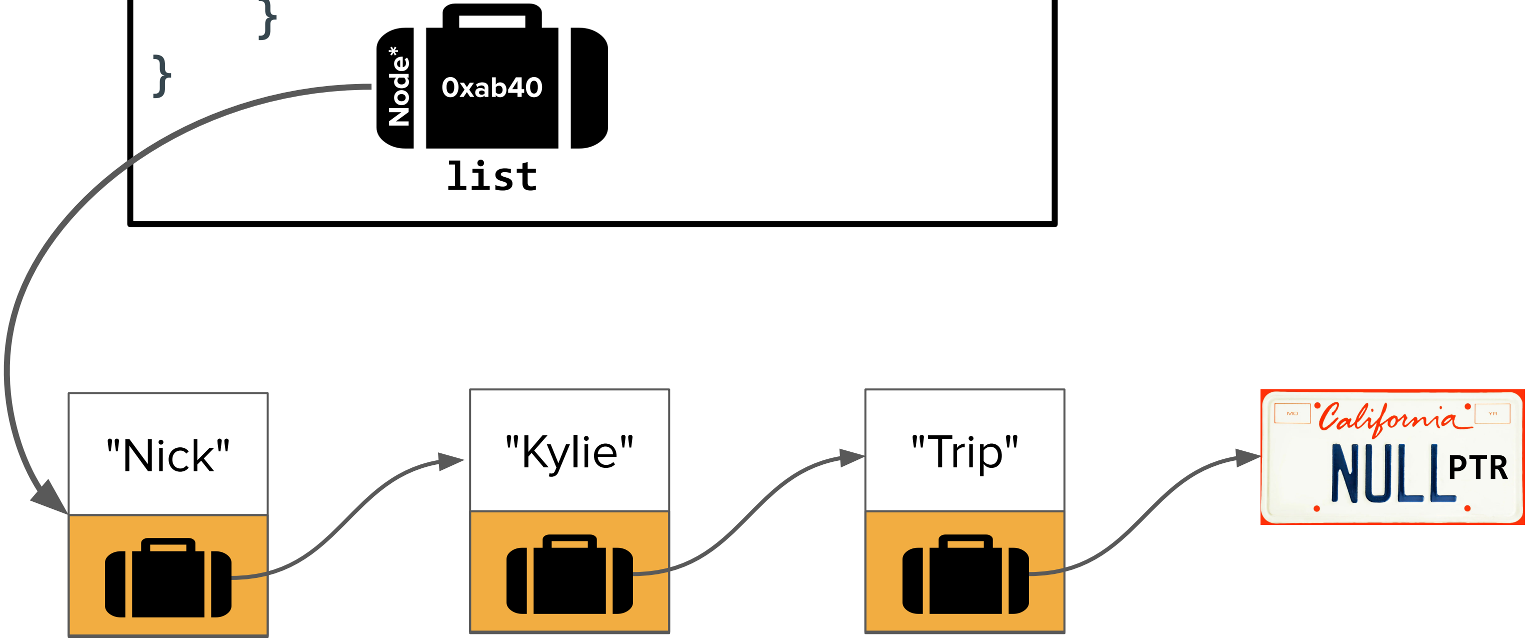
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



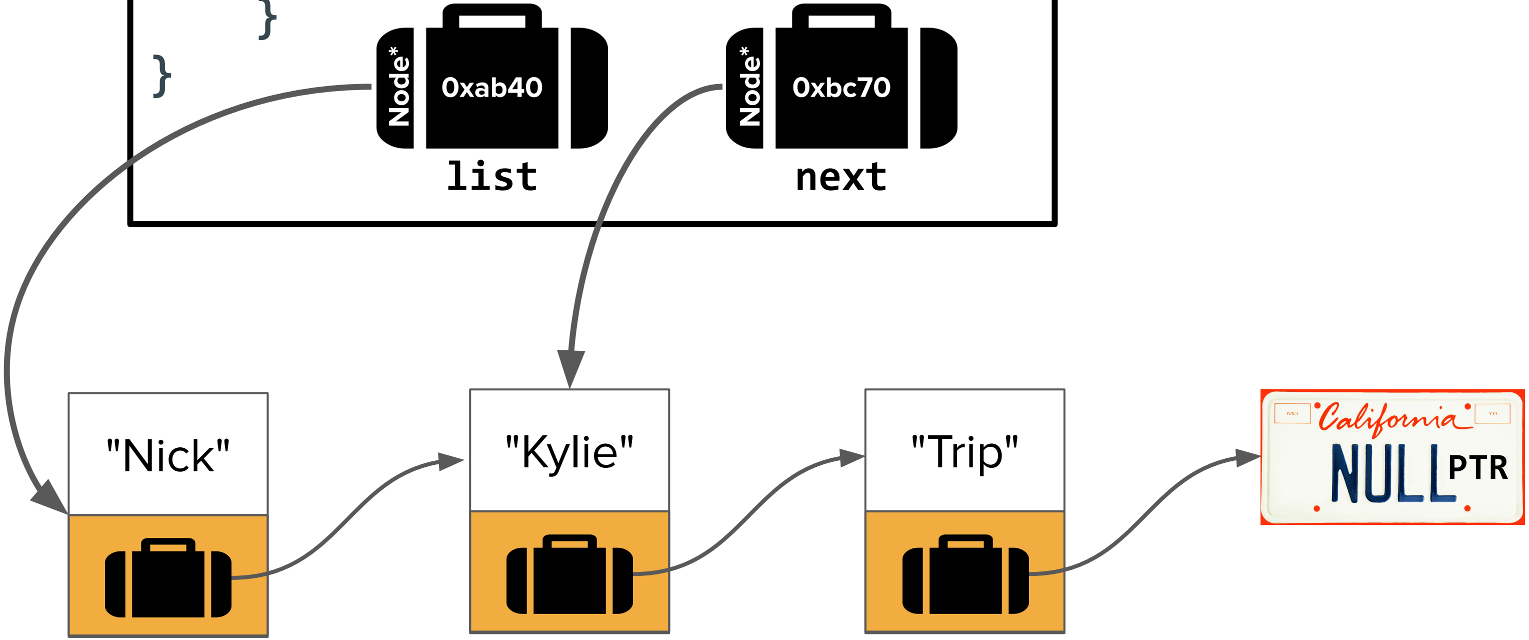
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



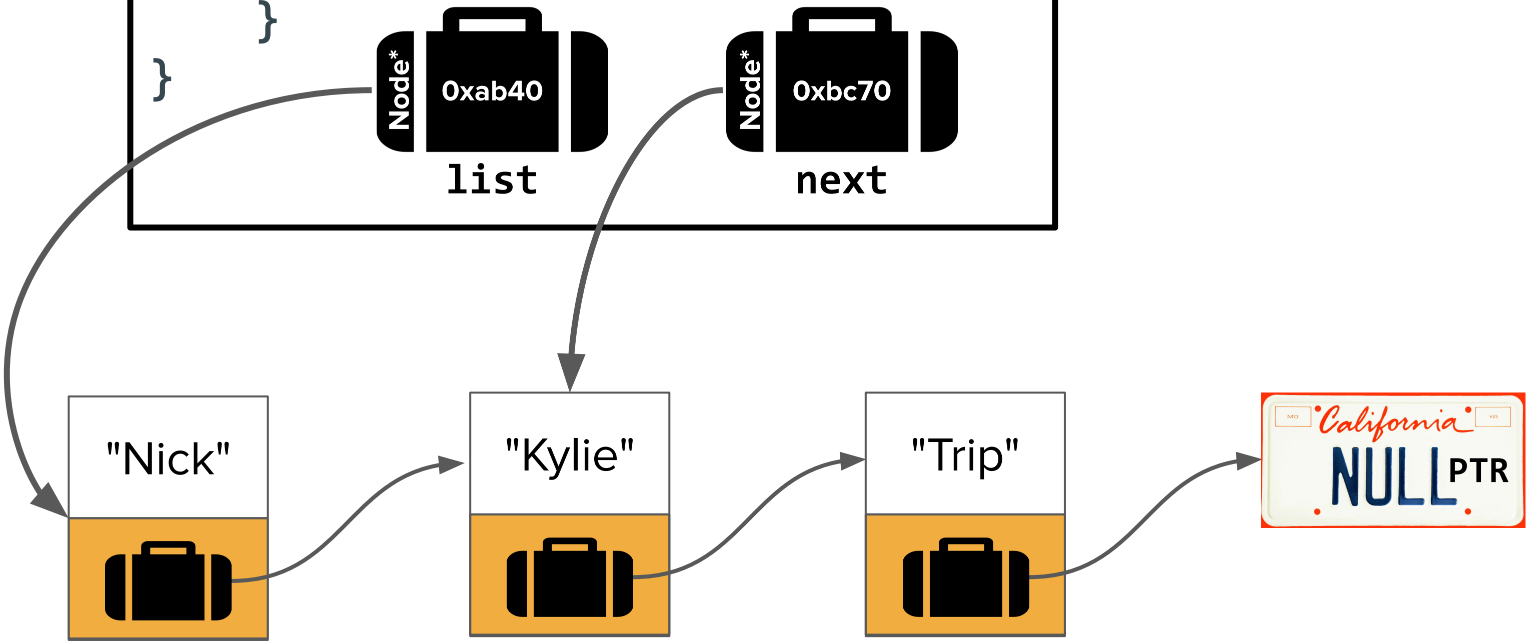
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



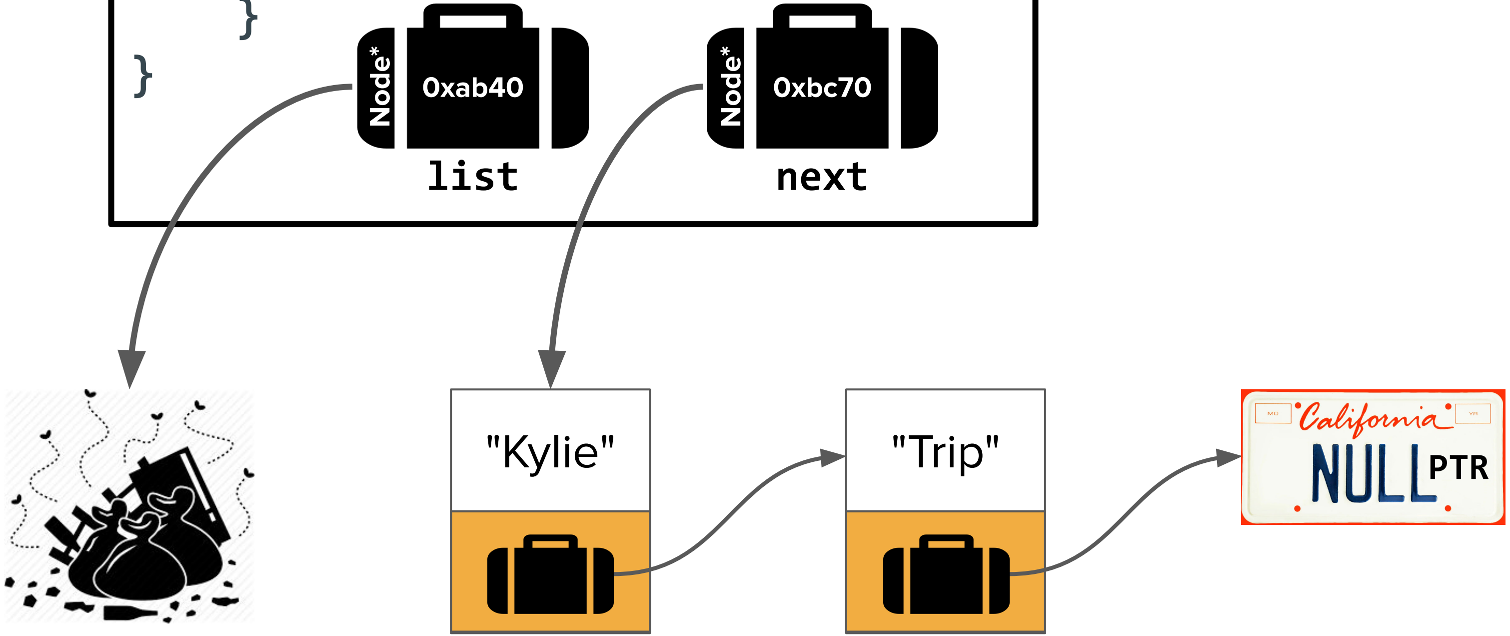
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



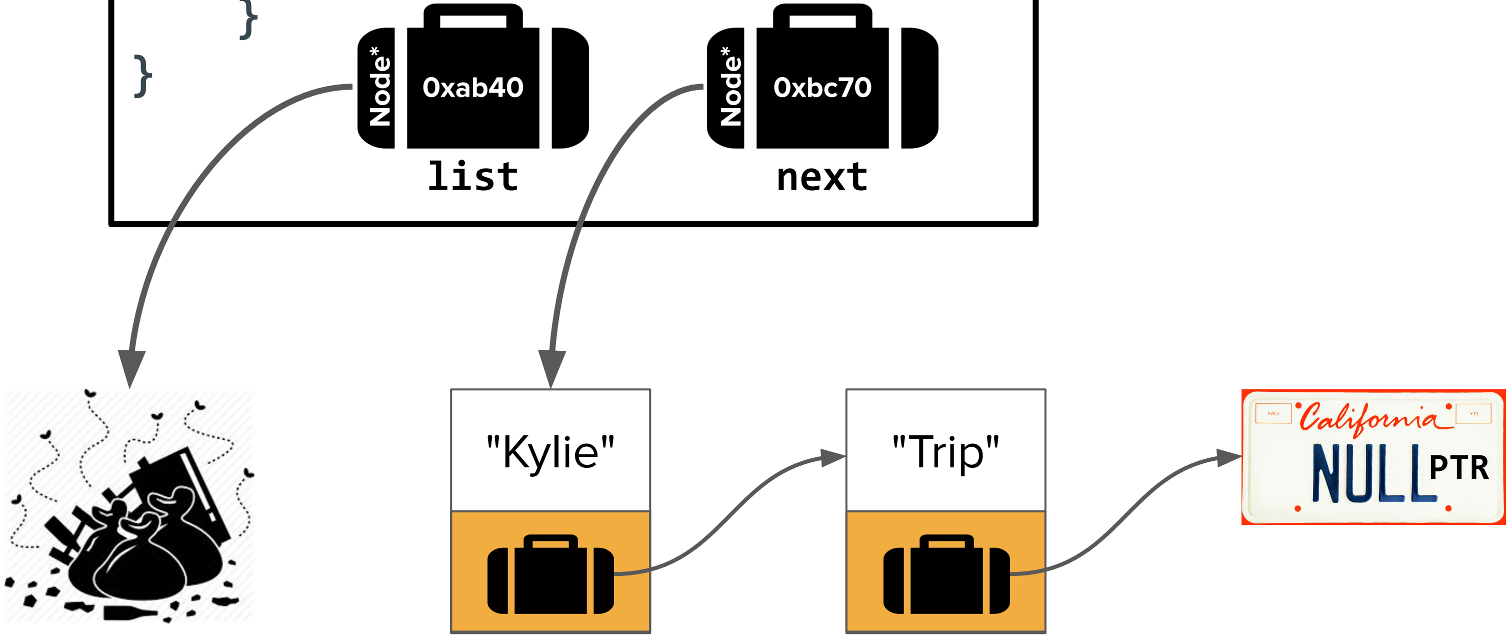
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



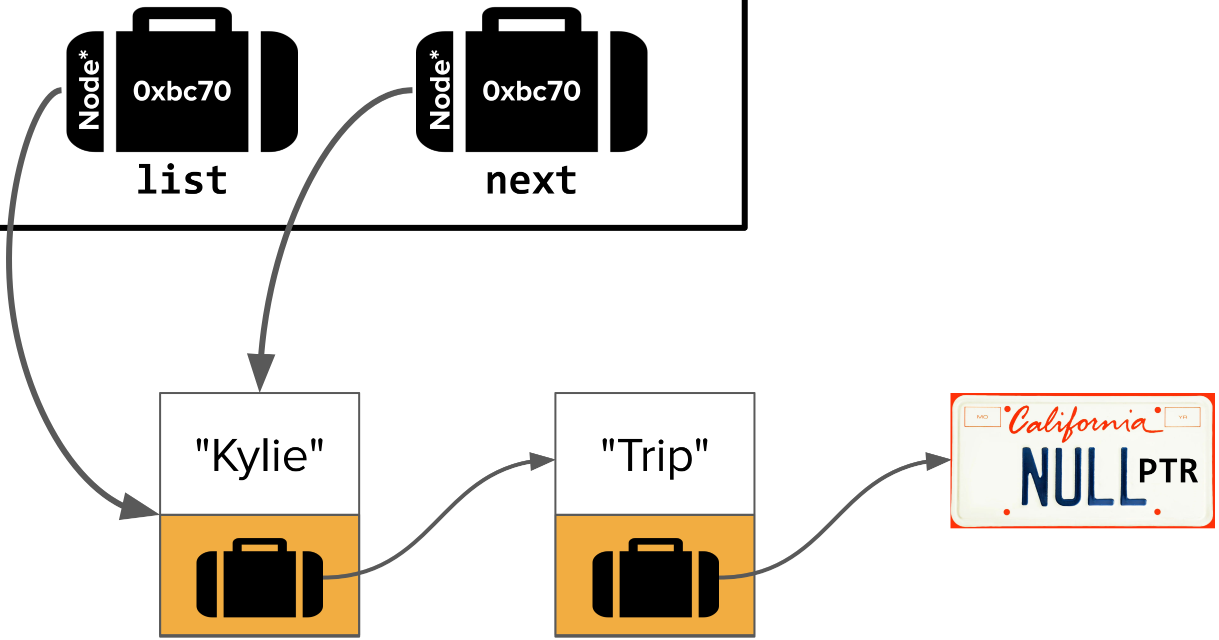
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



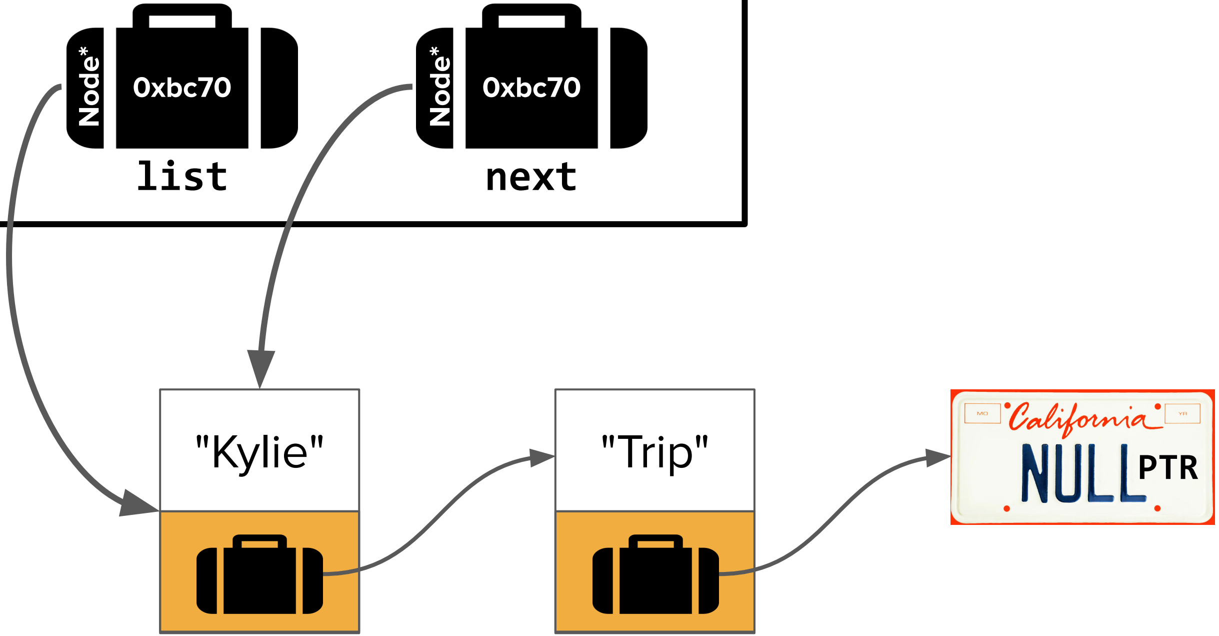
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



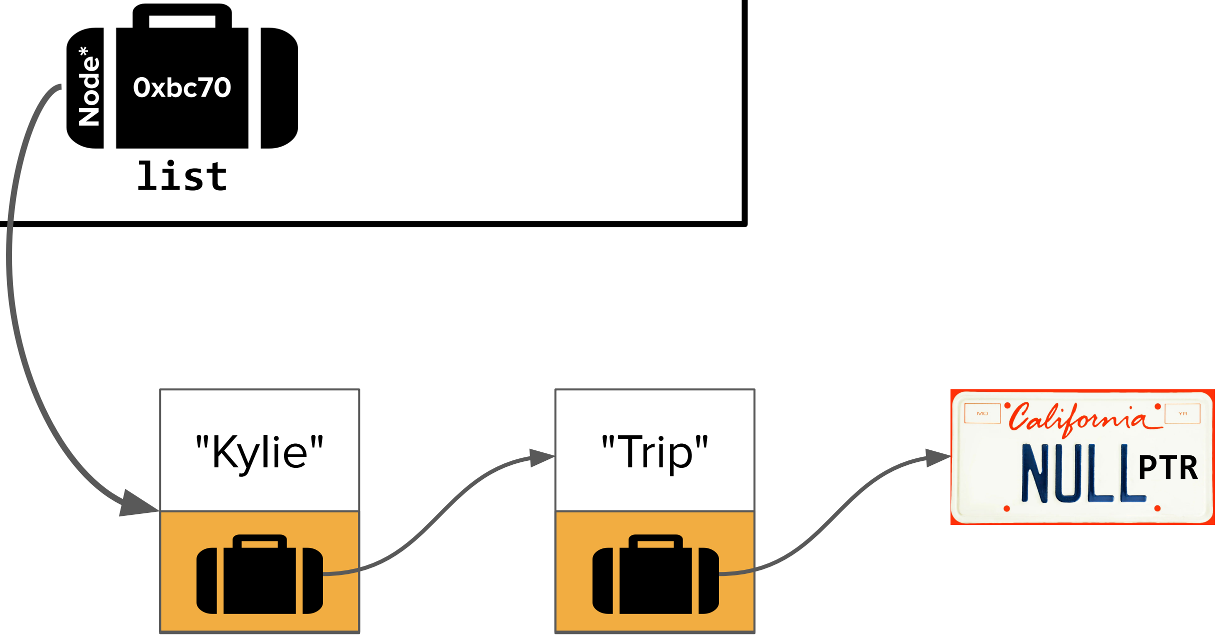
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



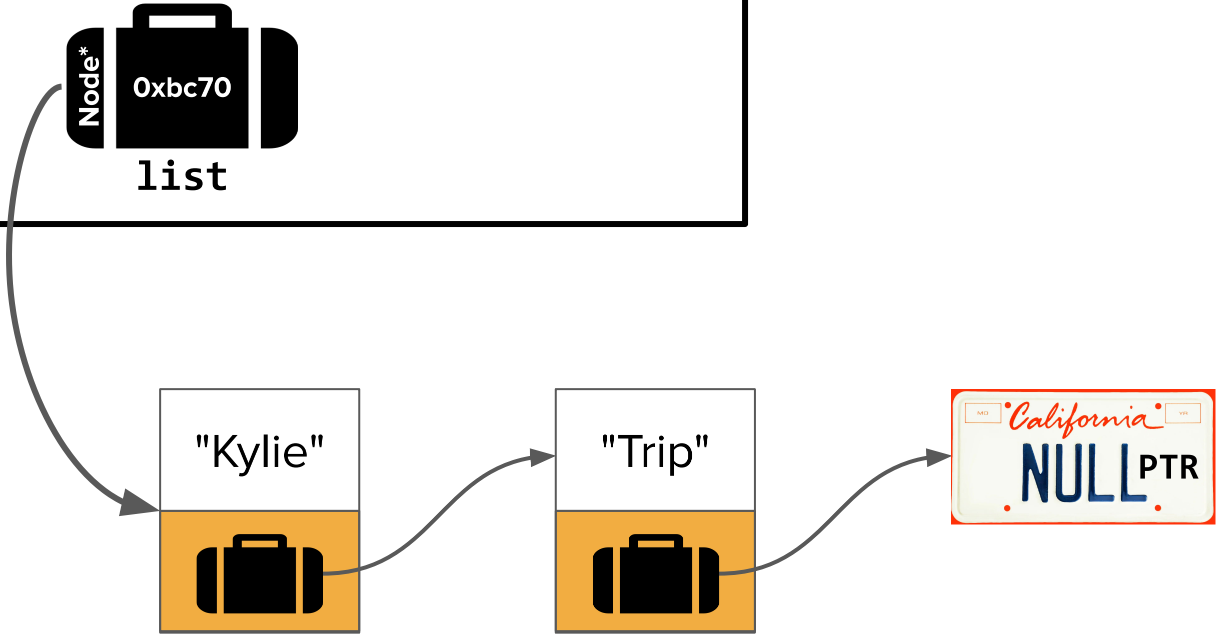
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



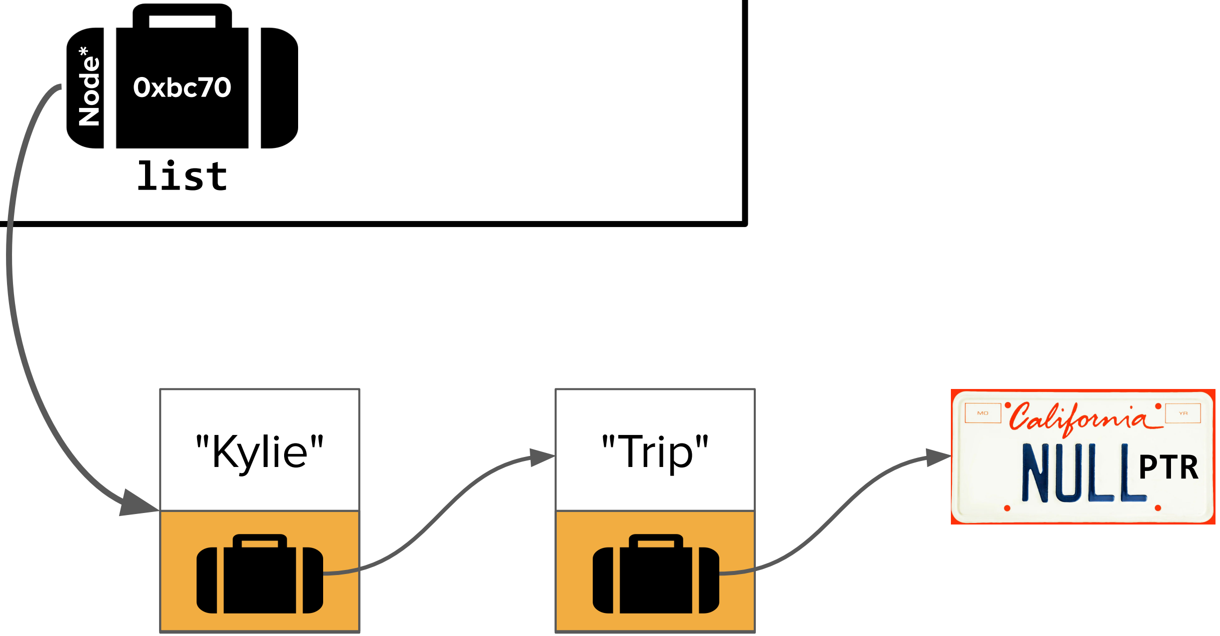
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



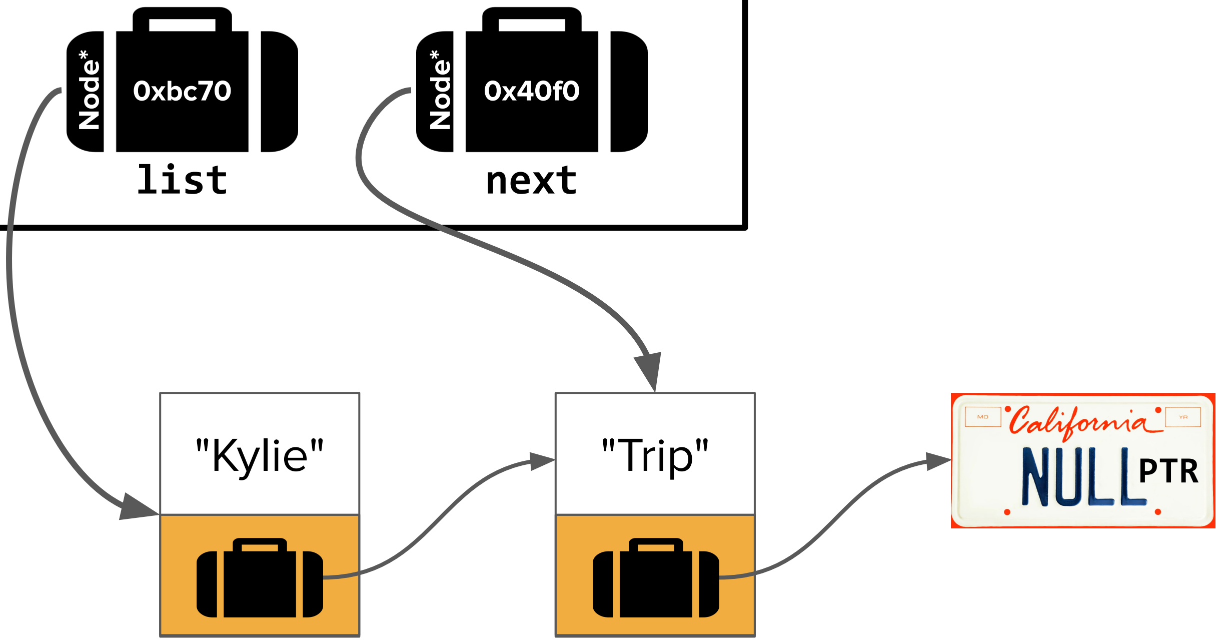
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



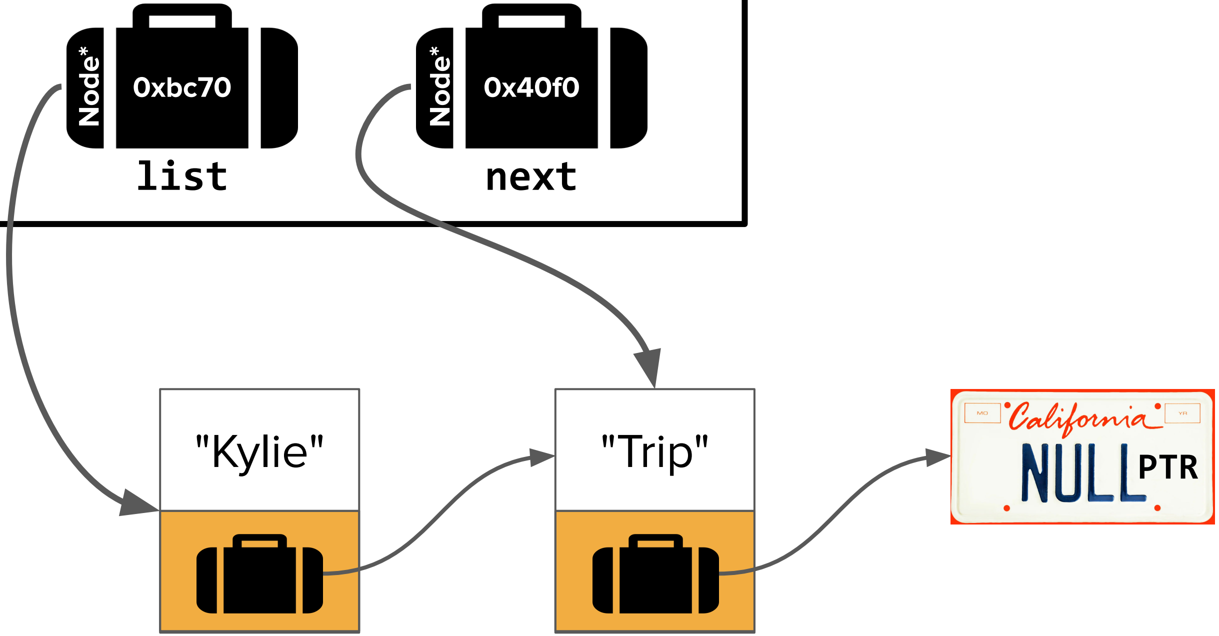
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



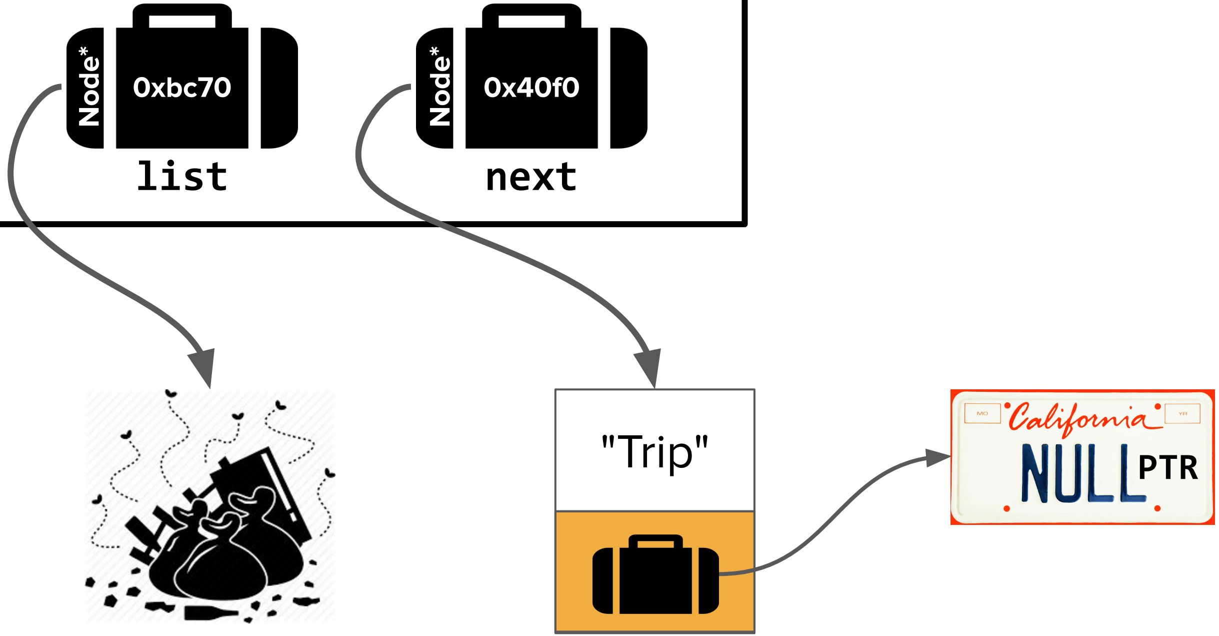
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



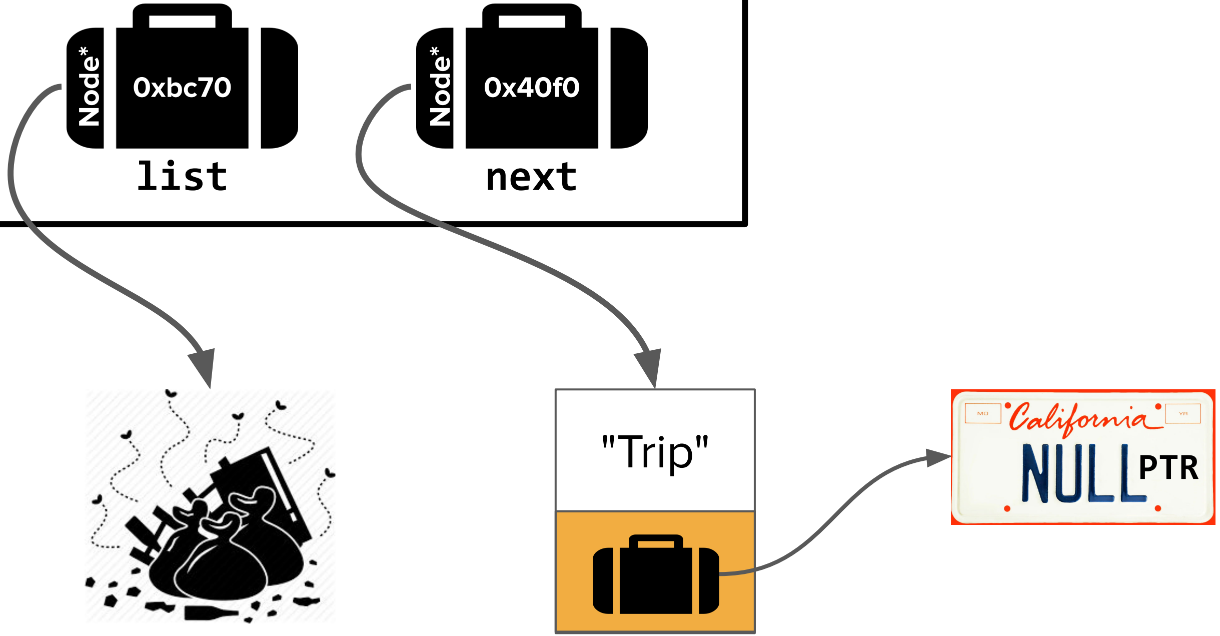
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



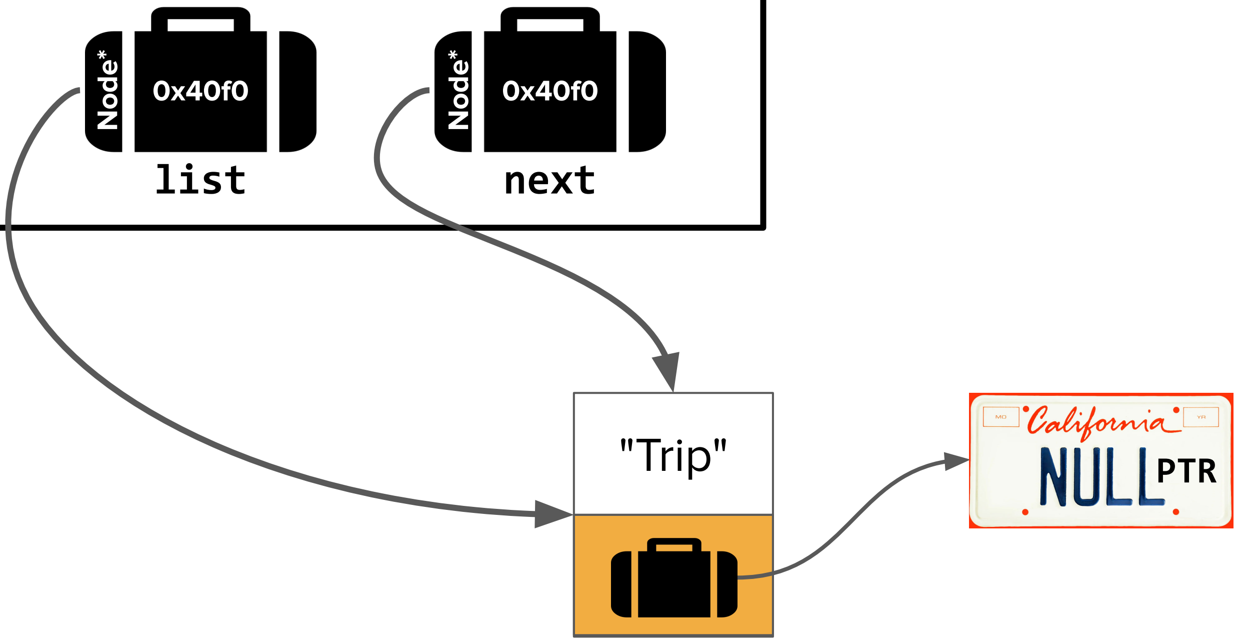
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



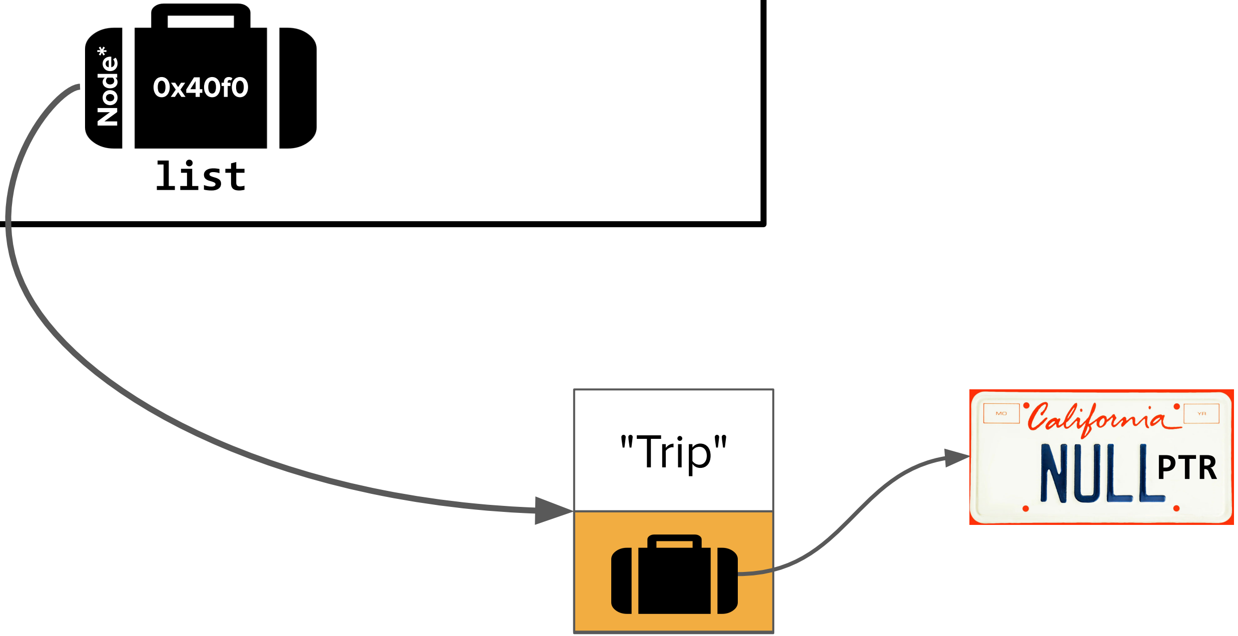
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



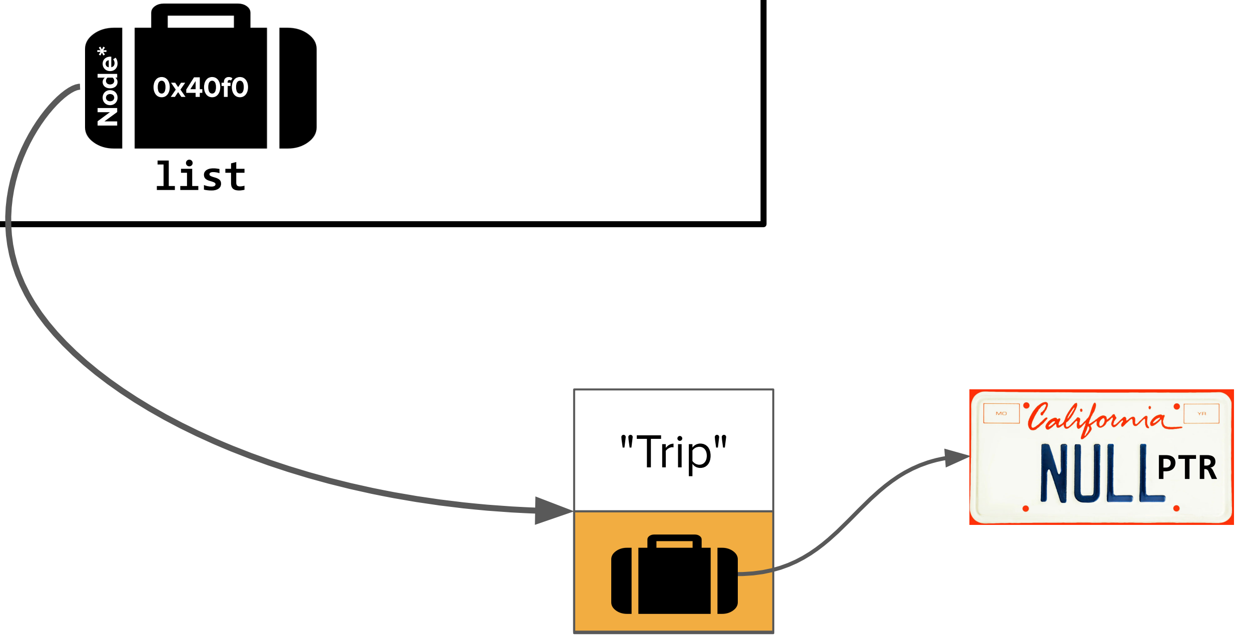
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



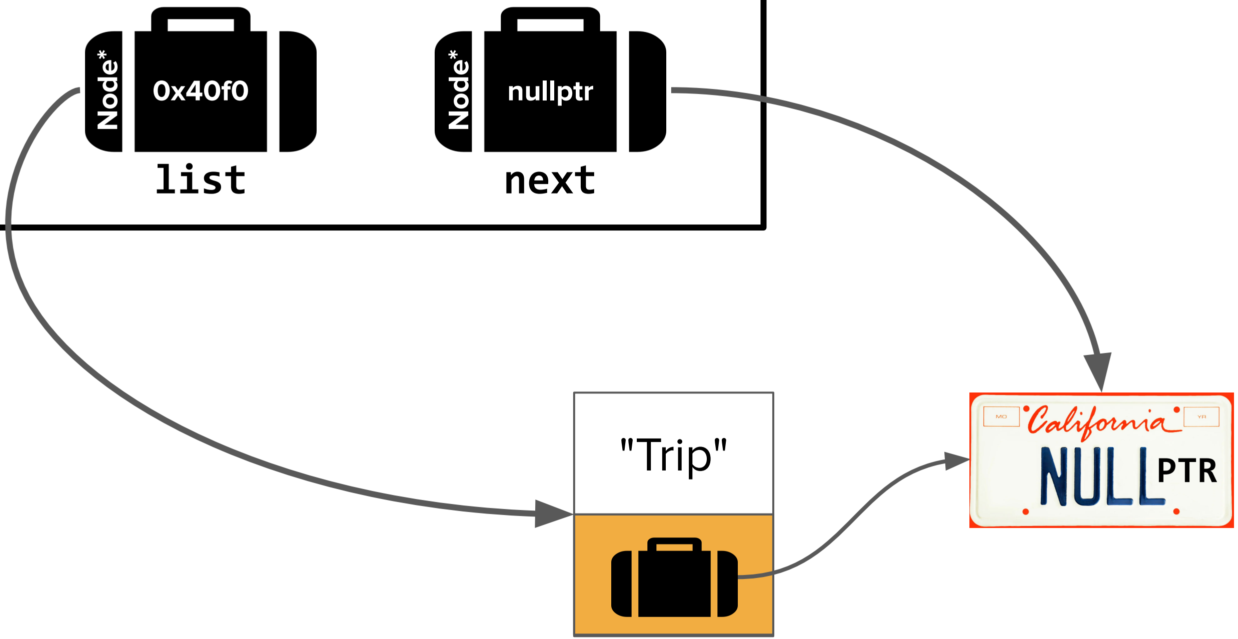
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



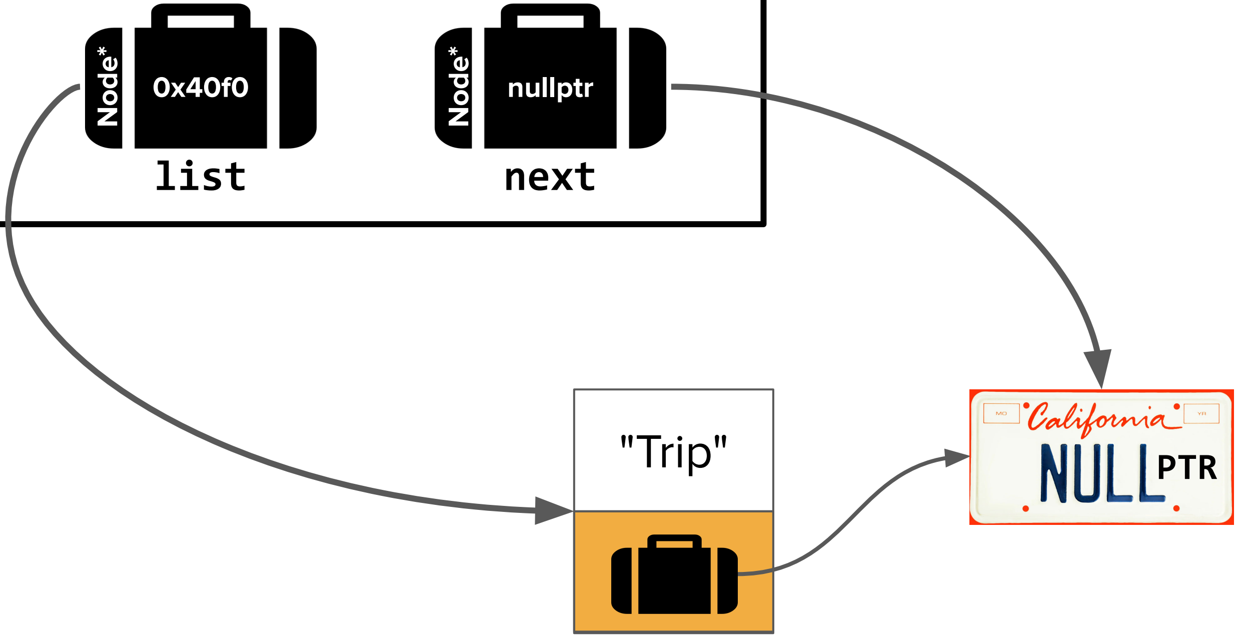
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



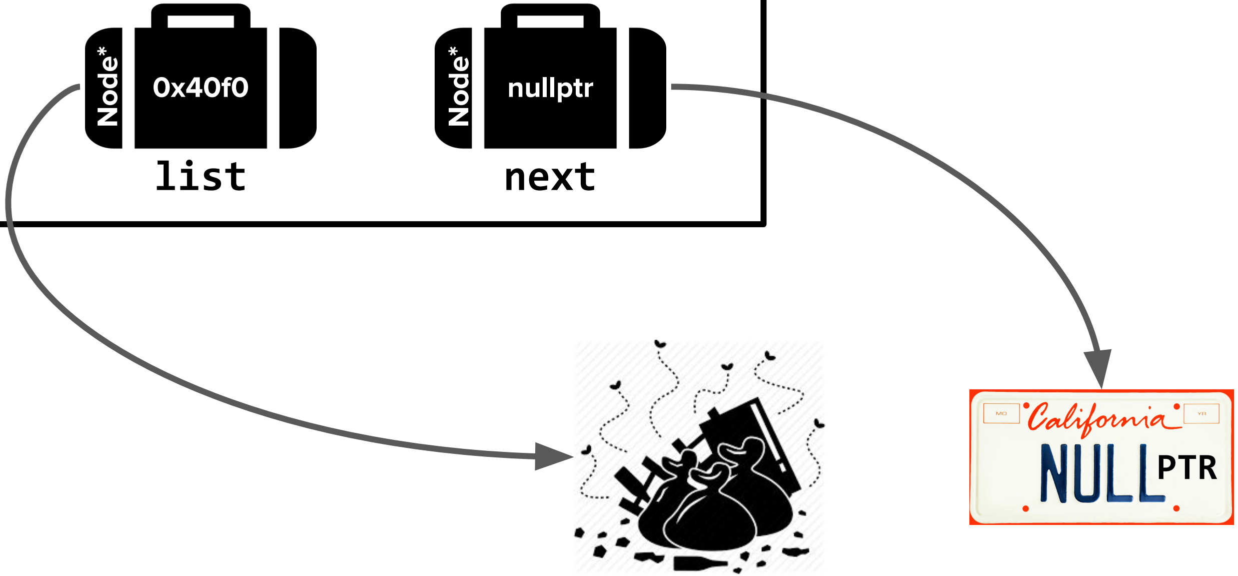
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



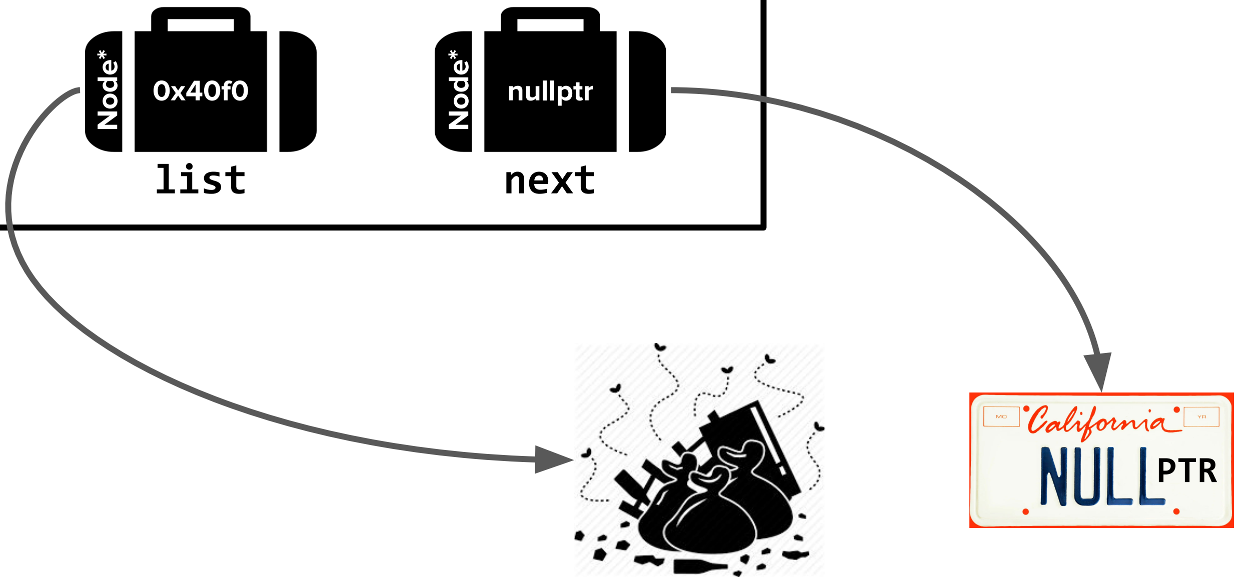
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



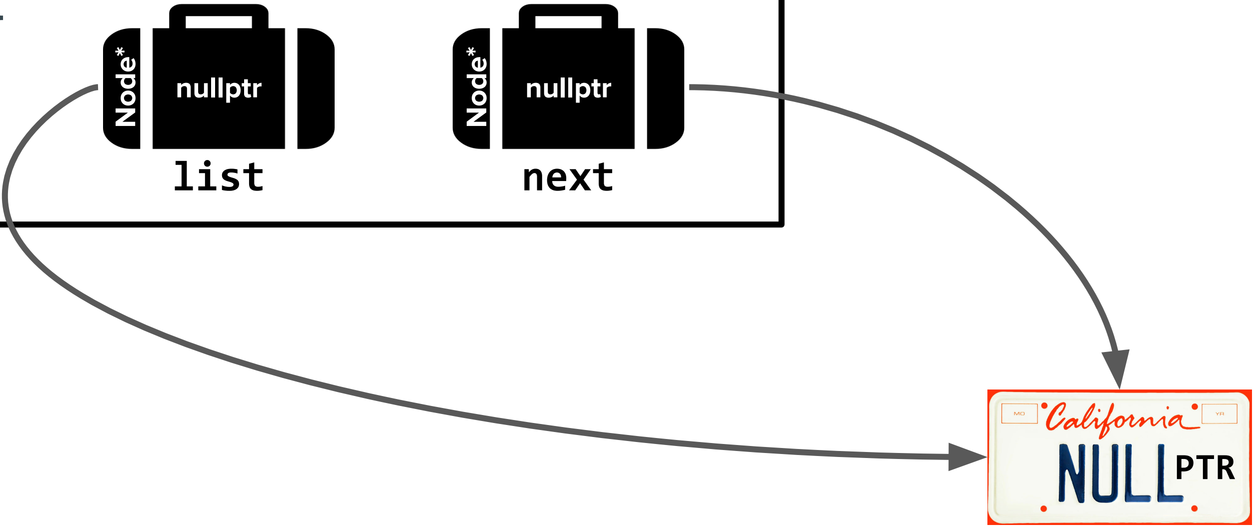
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



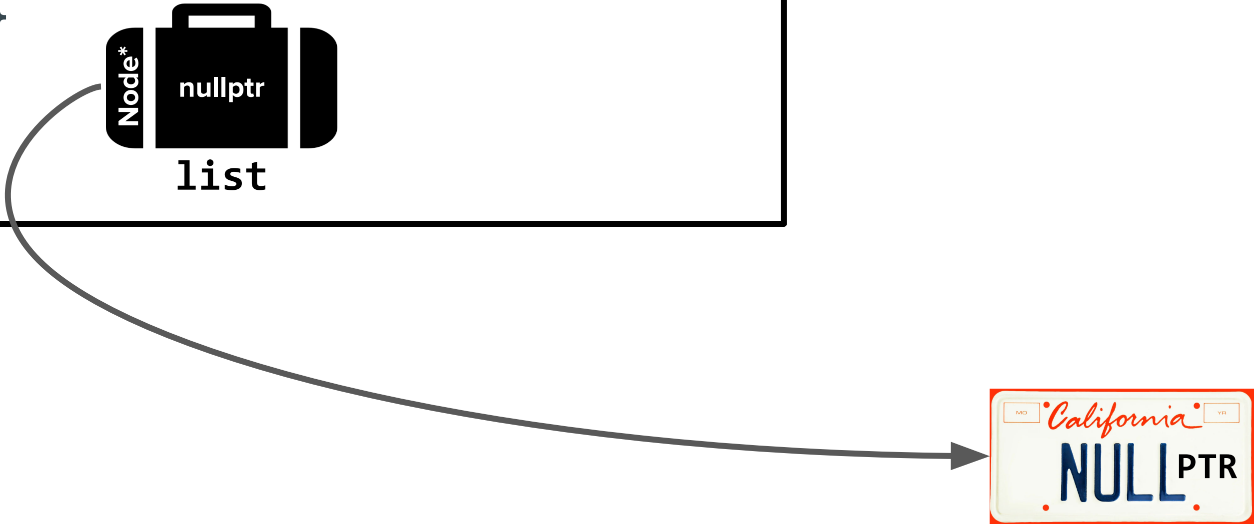
```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* next = list->next;
        delete list;
        list = next;
    }
}
```



```
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!

Linked Lists and Recursion

Rethinking Linked Lists

- On Thursday, we mentioned that the Node struct that defined the contents of a linked list was define **recursively**.

Rethinking Linked Lists

- On Thursday, we mentioned that the Node struct that defined the contents of a linked list was define **recursively**.

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

Rethinking Linked Lists

- On Thursday, we mentioned that the Node struct that defined the contents of a linked list was define **recursively**.

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

- This struct definition gives us some insight into the fact that the overall concept of a linked list can be expressed recursively.

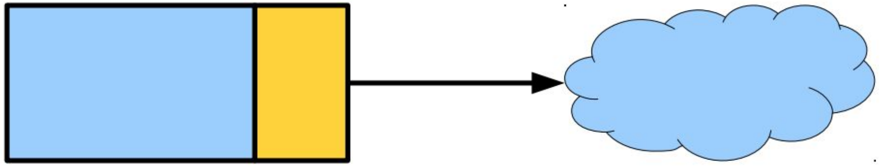
A Linked List is Either…

A Linked List is Either…

…an empty list,
represented by
nullptr, or…

A Linked List is Either…

…an empty list, represented by **`nullptr`**, or…



a single linked list cell that points…

… at another linked list.

Printing a List Revisited

Printing a List Revisited

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

Printing a List Revisited

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

```
void printListRec(Node* list) {
    /* Base Case: There's nothing
to print if the list is empty. */
    if (list == nullptr) return;

    /* Recursive Case: Print the
first node, then the rest of the
list. */
    cout << list->data << endl;
    printListRec(list->next);
}
```

Pitfalls of Recursive List Traversal

- Recursion can be a really elegant way to write code for a list traversal! However, recursion is not always the optimal problem-solving strategy...

Pitfalls of Recursive List Traversal

- Recursion can be a really elegant way to write code for a list traversal! However, recursion is not always the optimal problem-solving strategy...

- Note that the recursive solution generates one recursive call for every element in the list, meaning that a list with **`n`** elements would require **`n`** stack frames.

Pitfalls of Recursive List Traversal

- Recursion can be a really elegant way to write code for a list traversal! However, recursion is not always the optimal problem-solving strategy...

- Note that the recursive solution generates one recursive call for every element in the list, meaning that a list with `n` elements would require `n` stack frames.

- What is the stack frame limit on most computers?
    - You explored this on assignment 3 – for most computers it is somewhere in the range of 16-64K

Pitfalls of Recursive List Traversal

- Recursion can be a really elegant way to write code for a list traversal! However, recursion is not always the optimal problem-solving strategy...

- Note that the recursive solution generates one recursive call for every element in the list, meaning that a list with `n` elements would require `n` stack frames.

- What is the stack frame limit on most computers?
  - You explored this on assignment 3 – for most computers it is somewhere in the range of 16-64K

- With a recursive strategy, the size of the list we're able to process is limited by the stack frame capacity – we can't process lists longer than 16-64K elements!

Pitfalls of Recursive List Traversal

- Recursion can be a really elegant way to write code for a list traversal! However, ... ategy...
- Note that ... every element in the list, ... ck frames.
- What is th...
  - You ex... e range of 16-64K
- With a rec... s is limited by the stack frame capacity – we can't process lists longer than 16-64K elements!

Takeaway: Any linked list operations involving traversal of the whole list are better done *iteratively*! This holds especially true on the assignment – don't try to implement any of the list helper functions recursively!

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.

- Iterative traversal offers the most flexible, scalable way to write utility functions that are able to handle all different sizes of linked lists.

Announcements

Announcements (Part 1)

- Revisions for Assignment 3 opened today and will be due on **Thursday, July 30 at 11:59pm PDT.**
- Assignment 4 is due tonight at **Monday, July 27 at 11:59pm PDT.**
- Assignment 5 will be released by the end of the day tomorrow and will be due on **Tuesday, August 4 at 11:59pm PDT**.

Announcements (Part 2)

- Nick’s and Kylie's group office hour times this week will be slightly modified.
  - Kylie will be hosting group OHs from **2-3:30pm PDT on Monday (today!).**
  - Nick will be hosting group OHs from **12:30-3pm PDT on Tuesday.**

- Come talk to us about your ideas for the final project during office hours!

- Diagnostic grades were released earlier today. Overall, everyone did really well! Regrade requests are now open through Gradescope and must be submitted by **Wednesday, July 29 at 11:59pm**.
  - These requests should only be submitted if you think the posted criteria has been misapplied to your submission, **not** if you think the criteria are unfair.

Announcements (Part 3)

- Common diagnostic questions: What letter grade did I get? Is it curved?
    - We don’t want you to think about this as an exam! There won’t be a curve so you can think of the 45 total points as making up the 10% of your overall grade.
    - BUT since the emphasis for the diagnostic was to help you understand areas for improvement, we’re also going to give you opportunities to demonstrate growth and make up part of that 10%.

- You can receive the points back for **one problem of your choice** by showing that you’ve mastered that concept through your final project.
    - You don’t have to only focus on that topic in your project, but it should be incorporated into the problem you design.
    - **Deliverable**: As an add-on to the final project write-up, you’ll include a section titled “Diagnostic Reflection” that discusses how you improved in that topic, how your final project demonstrates your improvement, and how you would now approach the diagnostic problem differently from a problem-solving standpoint (this does not mean reproducing the correct solution!).

Linked List Insertion

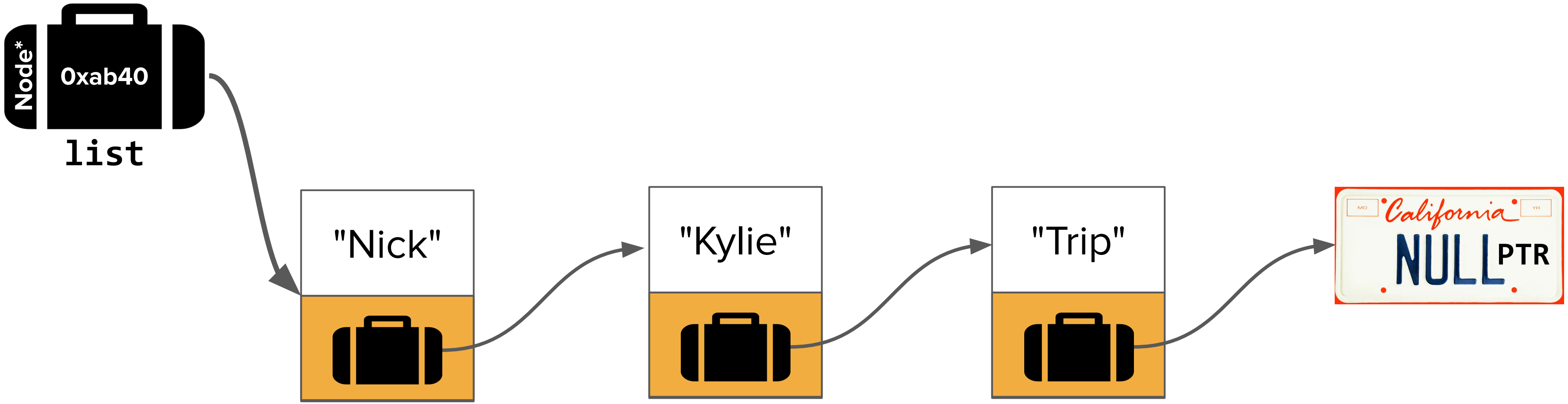
Insertion at the front (prepend)

Prepending an Element

- Suppose we wanted to write a function to insert an element at the front of a linked list.

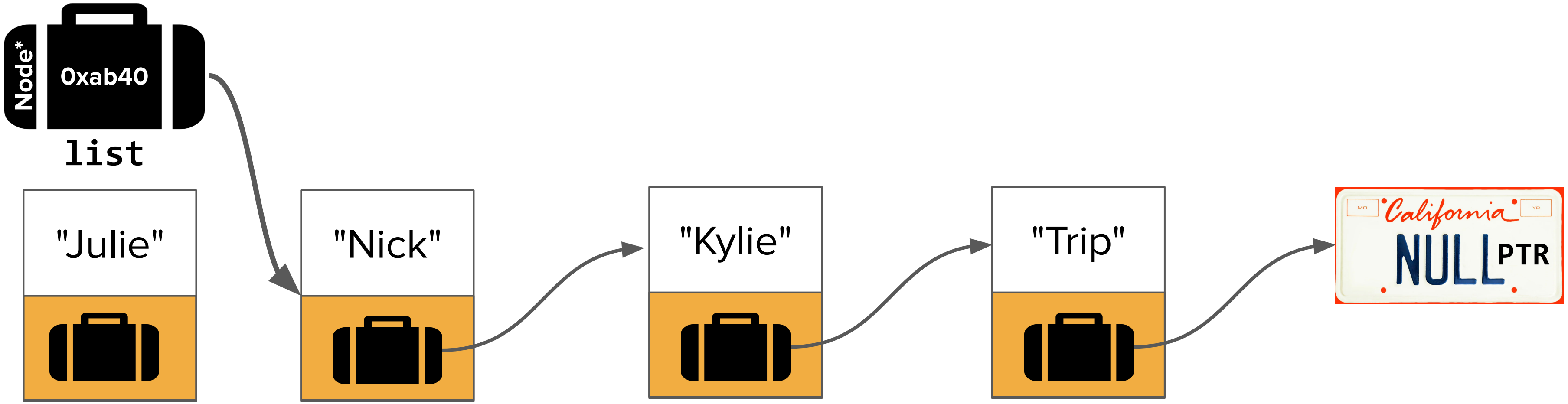
Prepending an Element

- Suppose we wanted to write a function to insert an element at the front of a linked list.



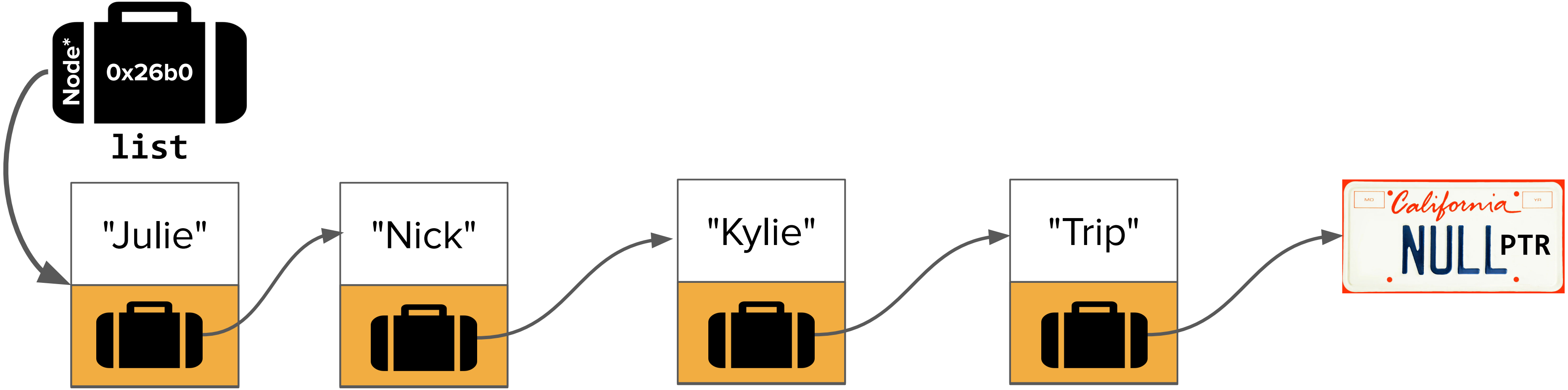
Prepending an Element

- Suppose we wanted to write a function to insert an element at the front of a linked list.



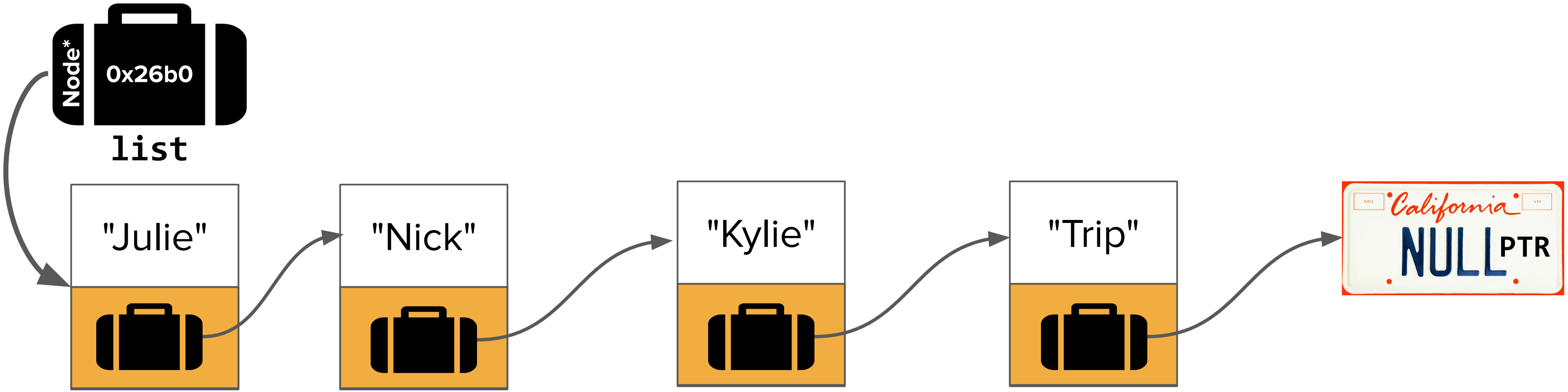
Prepending an Element

- Suppose we wanted to write a function to insert an element at the front of a linked list.



Prepending an Element

- Suppose we wanted to write a function to insert an element at the front of a linked list.
- This is similar to the **`push()`** function we implemented on Thursday, but now we're writing a standalone function to do this on an arbitrary list. Let's code it!



`prependTo()`

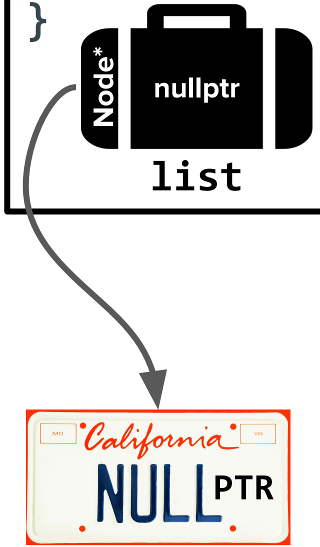
Let's code it!

What went wrong?

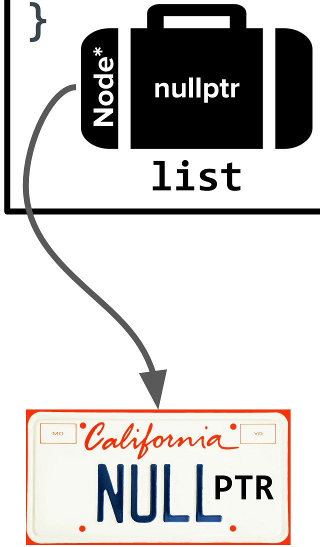
```
int main() {
    Node* list = nullptr;
    prependTo(list, "Trip");
    prependTo(list, "Kylie");
    prependTo(list, "Nick");
    return 0;
}
```

```
int main() {
    Node* list = nullptr;
    prependTo(list, "Trip");
    prependTo(list, "Kylie");
    prependTo(list, "Nick");
    return 0;
}
```

```
int main() {
    Node* list = nullptr;
    prependTo(list, "Trip");
    prependTo(list, "Kylie");
    prependTo(list, "Nick");
    return 0;
}
```

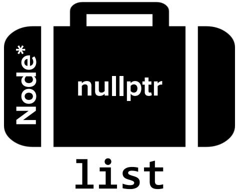


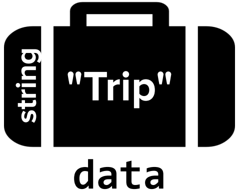
```
int main() {
    Node* list = nullptr;
    prependTo(list, "Trip");
    prependTo(list, "Kylie");
    prependTo(list, "Nick");
    return 0;
}
```



```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

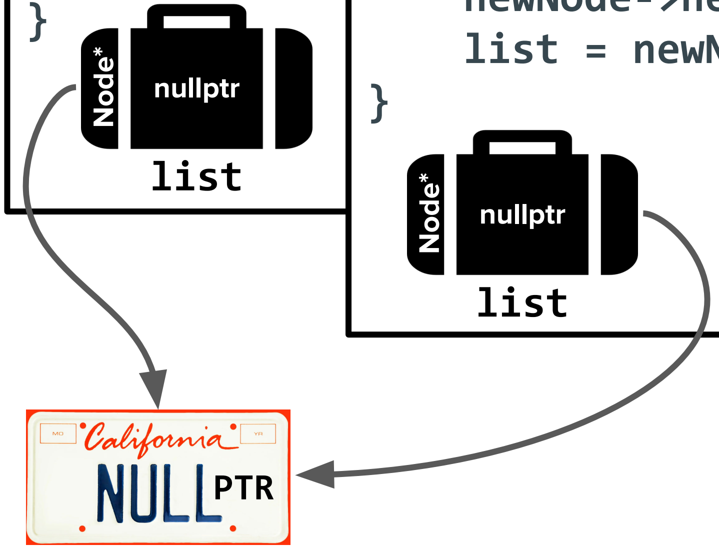

```
void prependTo(Node* list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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

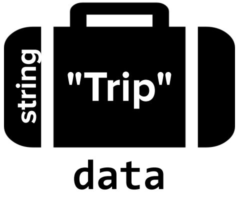




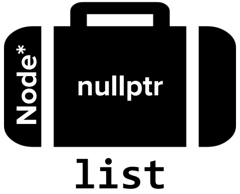
```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

```
void prependTo(Node* list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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

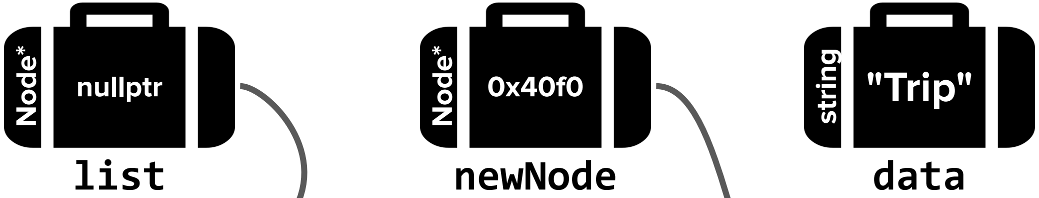


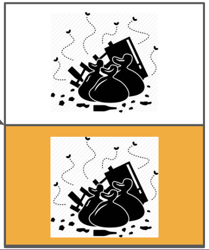


```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```



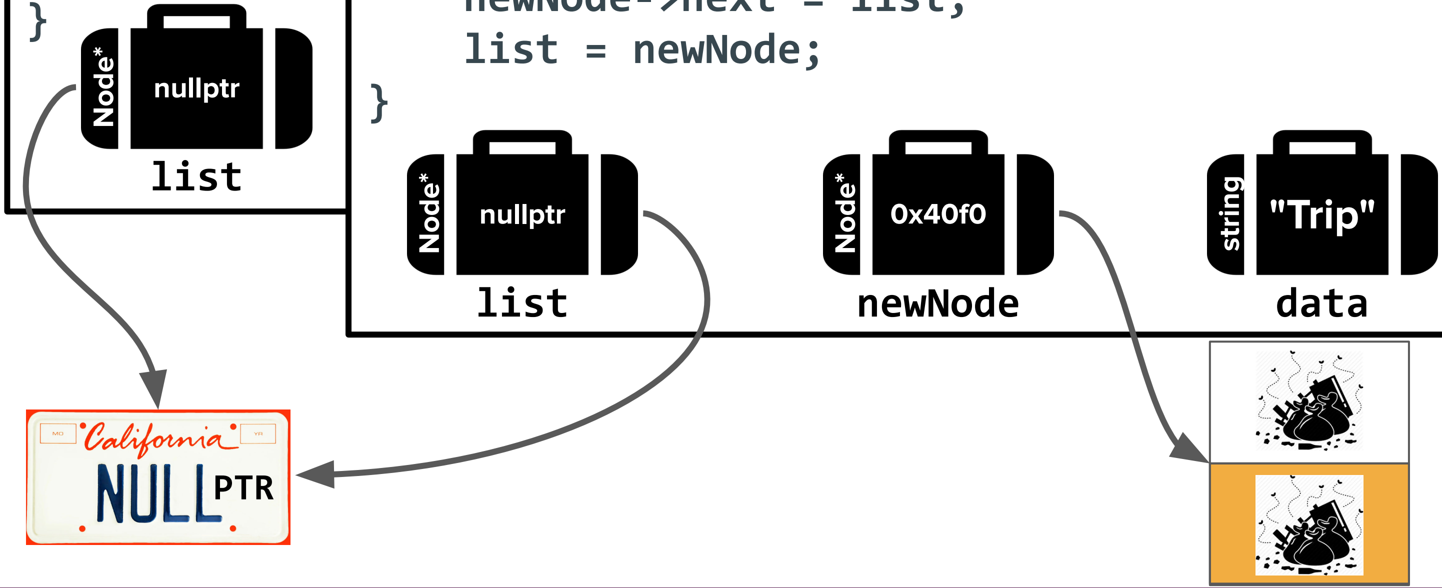

```
void prependTo(Node* list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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





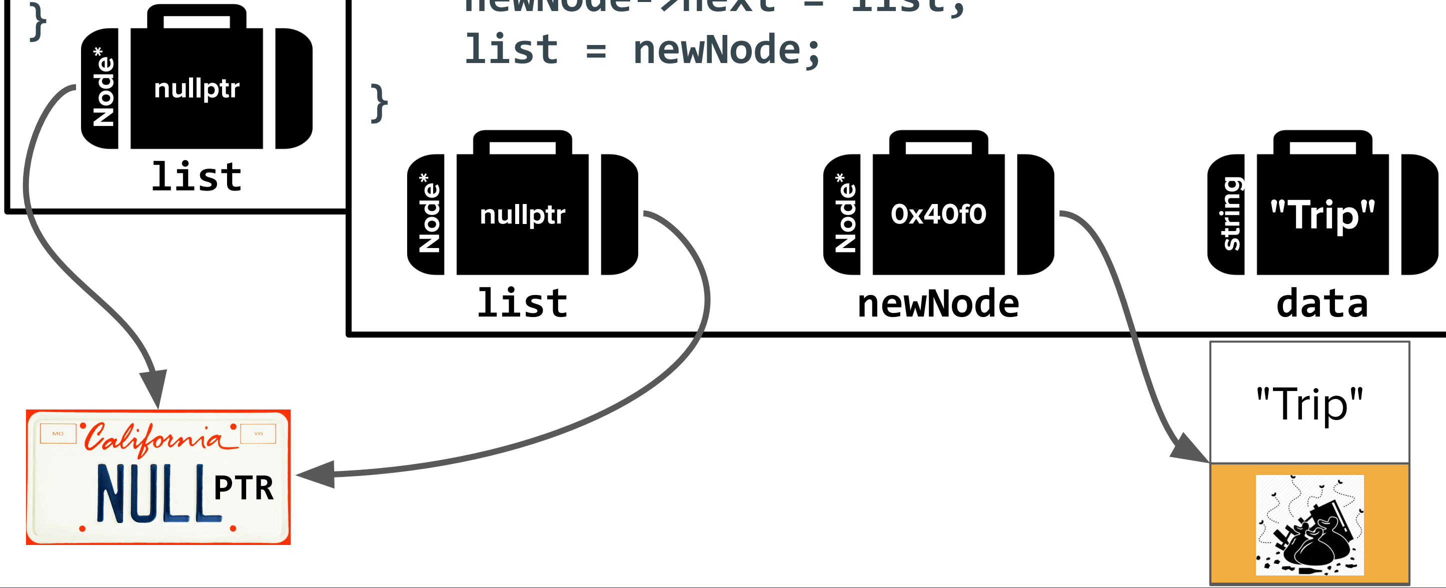
```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

```
void prependTo(Node* list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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



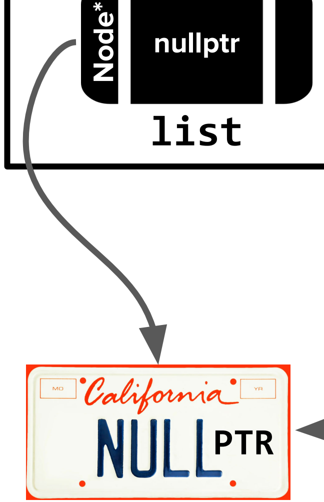
```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

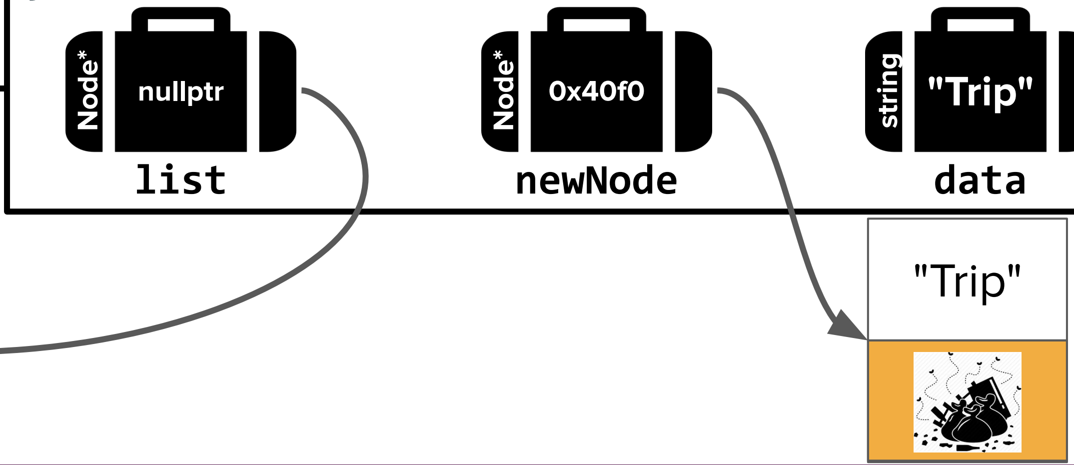
```
void prependTo(Node* list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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



```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

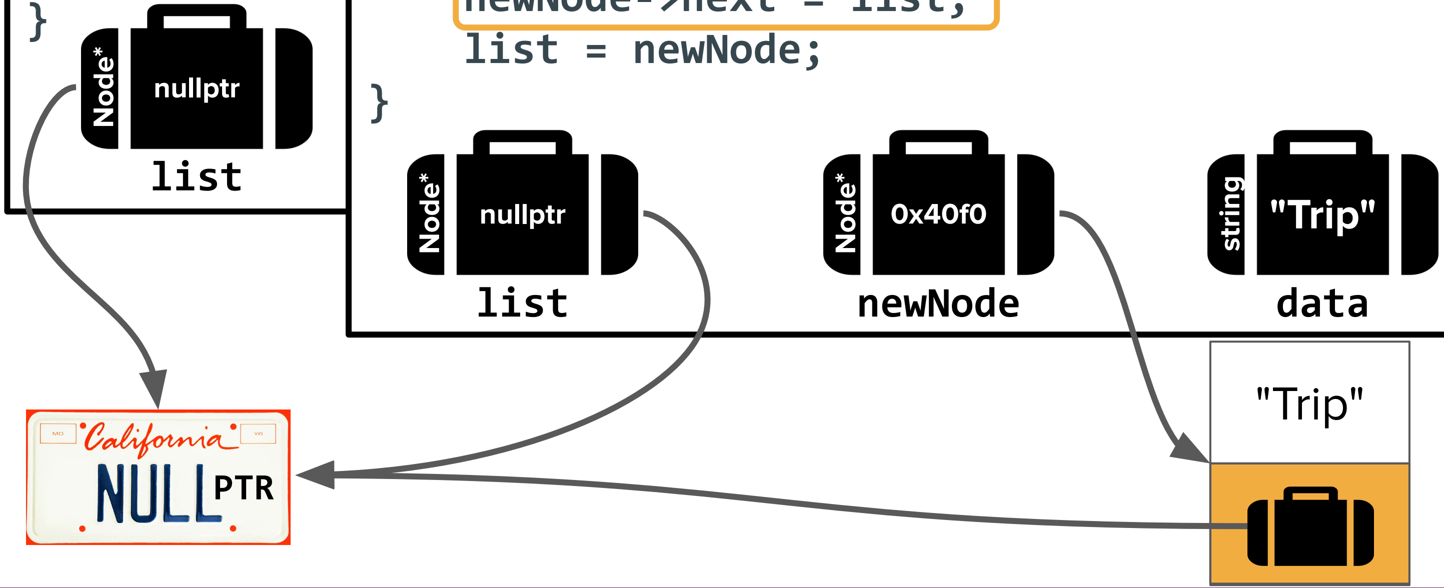
```
void prependTo(Node* list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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





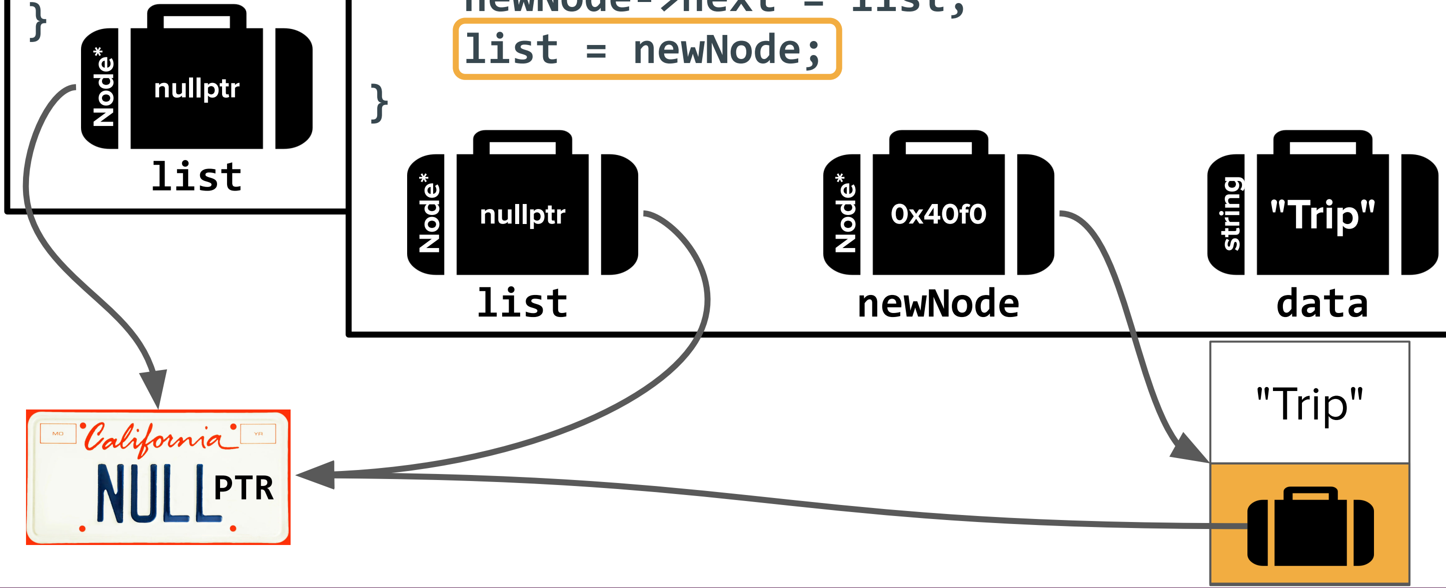
```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

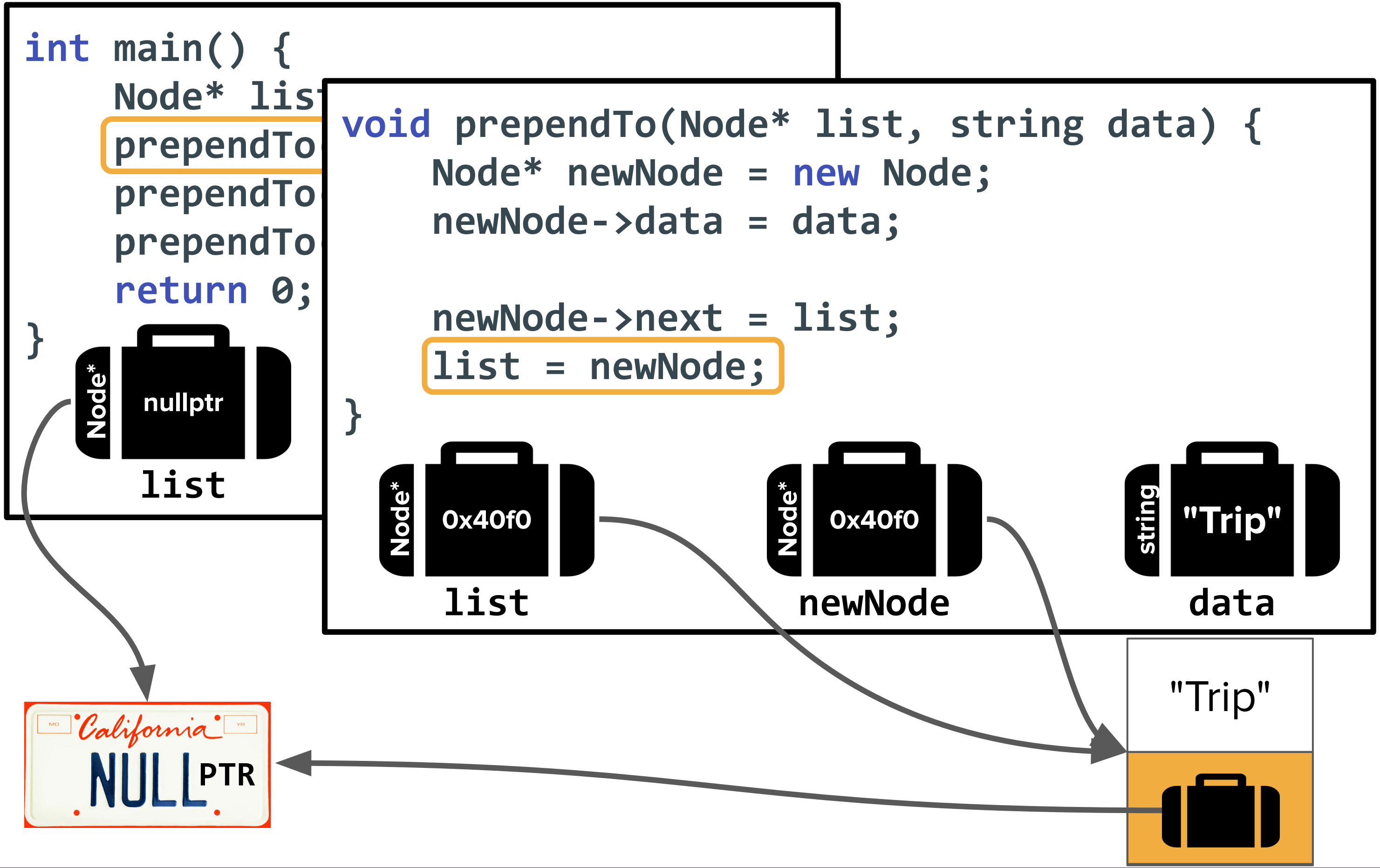
```
void prependTo(Node* list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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



```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

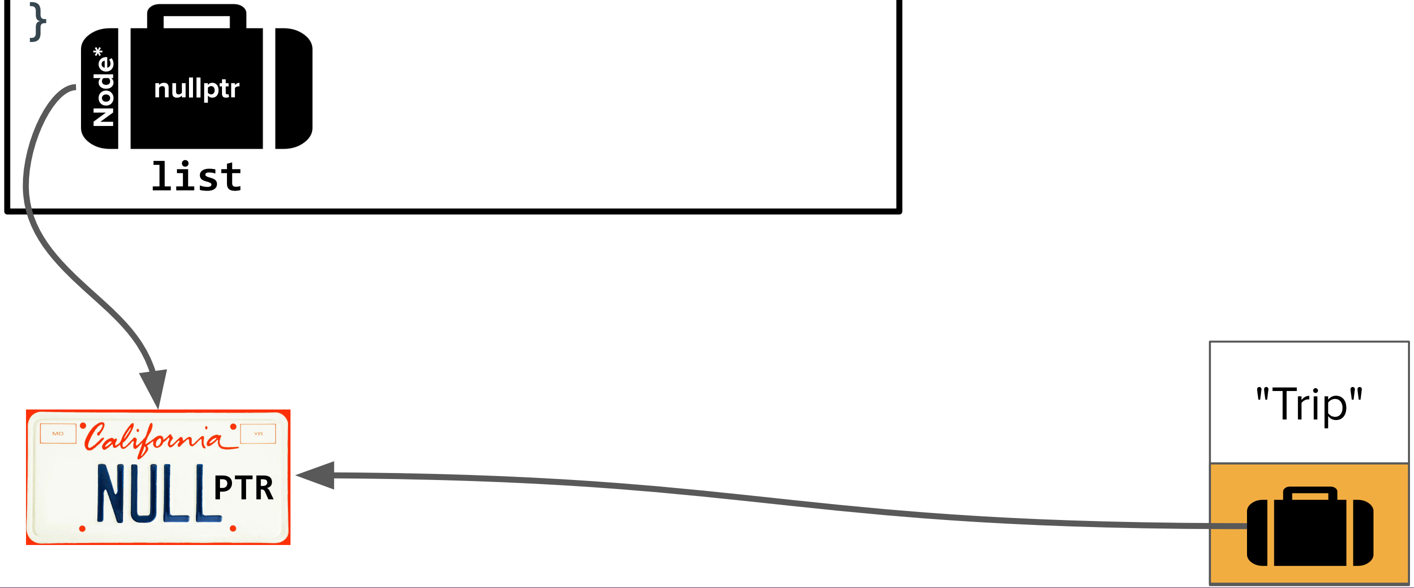
```
void prependTo(Node* list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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



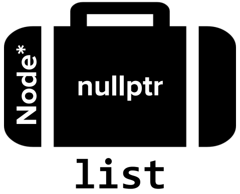


int main() {
Node* lis
prependTo
prependTo
prependTo
return 0;
}
Node*
nullptr
list
void prependTo(Node* list, string data) {
Node* newNode = new Node;
newNode->data = data;
newNode->next = list;
list = newNode;
}
Node*
0x40f0
list
Node*
0x40f0
newNode
string
"Trip"
data
"Trip"
California
NULL PTR

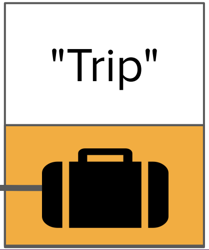
```
int main() {
    Node* list = nullptr;
    prependTo(list, "Trip");
    prependTo(list, "Kylie");
    prependTo(list, "Nick");
    return 0;
}
```



```
int main() {
    Node* list = nullptr;
    prependTo(list, "Trip");
    prependTo(list, "Kylie");
    prependTo(list, "Nick");
    return 0;
}
```



I just got yeeted into the land of leaked memory...

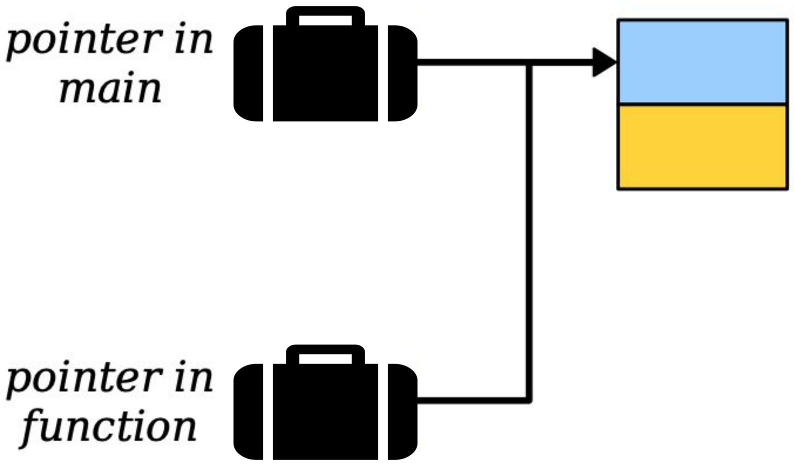


Pointers by Value

- Unless specified otherwise, function arguments in C++ are passed by value – this includes pointers!

- A function that takes a pointer as an argument gets a copy of the pointer.

- We can change where the copy points, but not where the original pointer points.



Pointers by Reference

Pointers by Reference

- To solve our earlier problem, we can **pass the linked list pointer by reference.**

Pointers by Reference

- To solve our earlier problem, we can **pass the linked list pointer by reference.**

- Our new function:

```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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

Pointers by Reference

- To solve our earlier problem, we can **pass the linked list pointer by reference.**

- Our new function:

```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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

Pointers by Reference

- To solve our earlier problem, we can **pass the linked list pointer by reference.**

- Our new function:

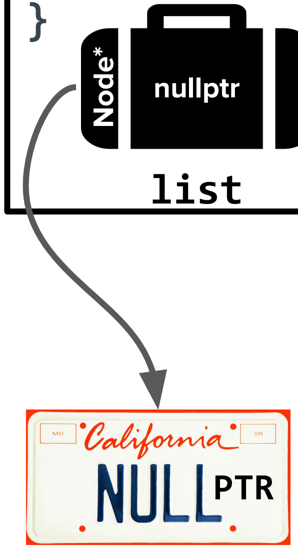
```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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

This is a *reference to a pointer to a Node*. If we change where list points in this function, the changes will stick!

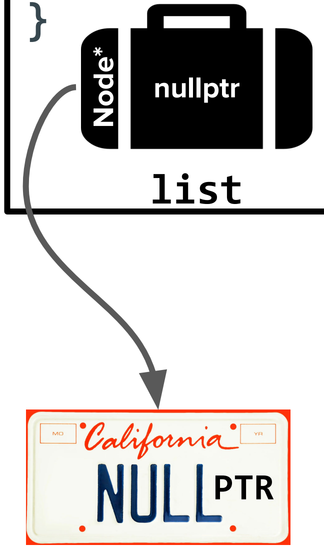
```
int main() {
    Node* list = nullptr;
    prependTo(list, "Trip");
    prependTo(list, "Kylie");
    prependTo(list, "Nick");
    return 0;
}
```

```
int main() {
    Node* list = nullptr;
    prependTo(list, "Trip");
    prependTo(list, "Kylie");
    prependTo(list, "Nick");
    return 0;
}
```

```
int main() {
    Node* list = nullptr;
    prependTo(list, "Trip");
    prependTo(list, "Kylie");
    prependTo(list, "Nick");
    return 0;
}
```

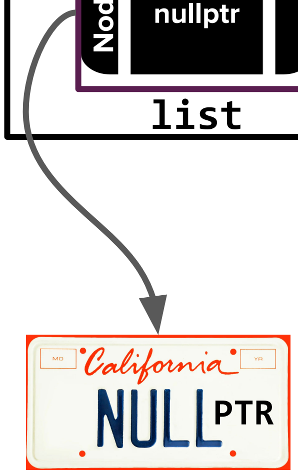


```
int main() {
    Node* list = nullptr;
    prependTo(list, "Trip");
    prependTo(list, "Kylie");
    prependTo(list, "Nick");
    return 0;
}
```



```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

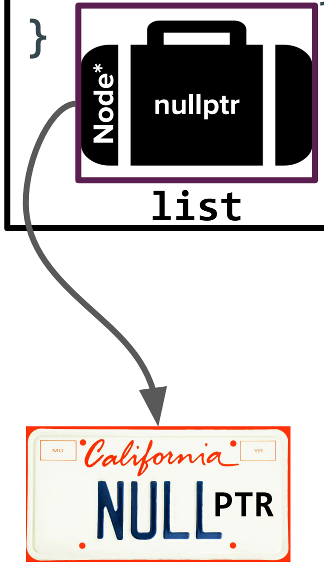
list




```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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

data

```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```



list

```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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

data

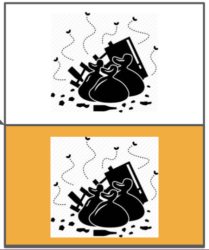
```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

list

```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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

newNode

data



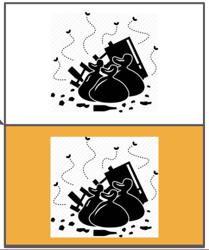
```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

list

```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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

newNode

data



```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

list

```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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

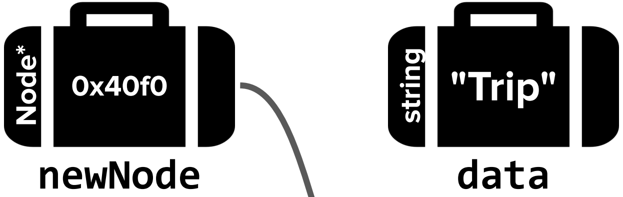
newNode

data

```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

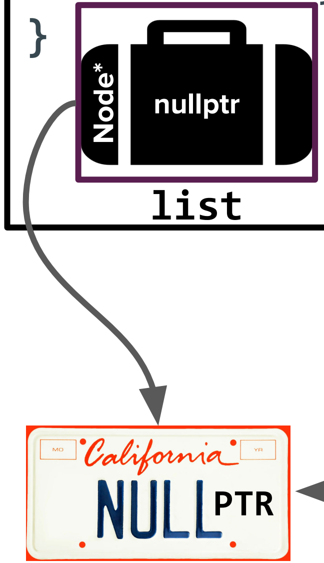
list

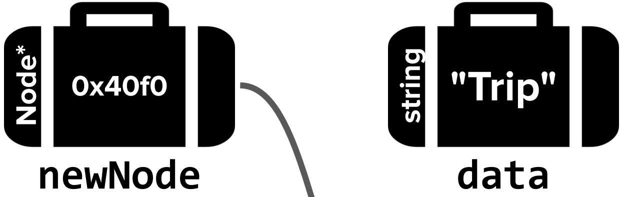

```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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



```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

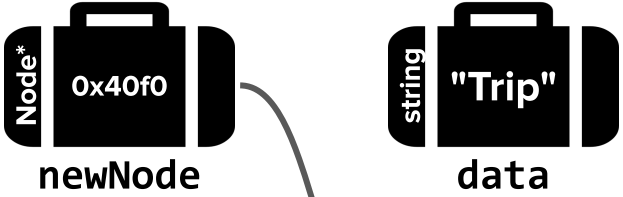
```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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

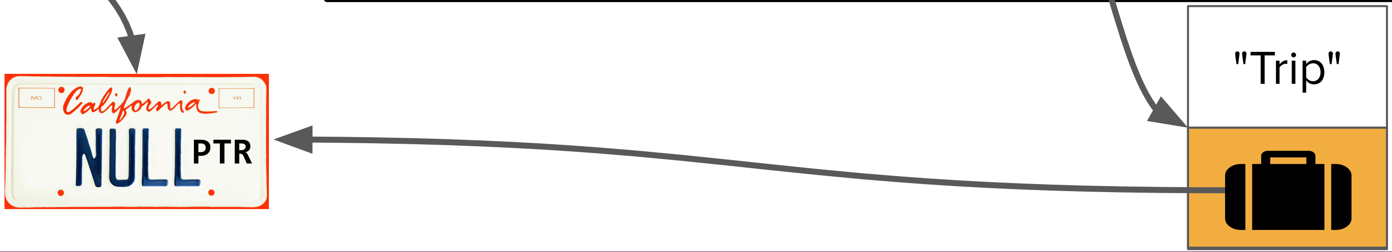




```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

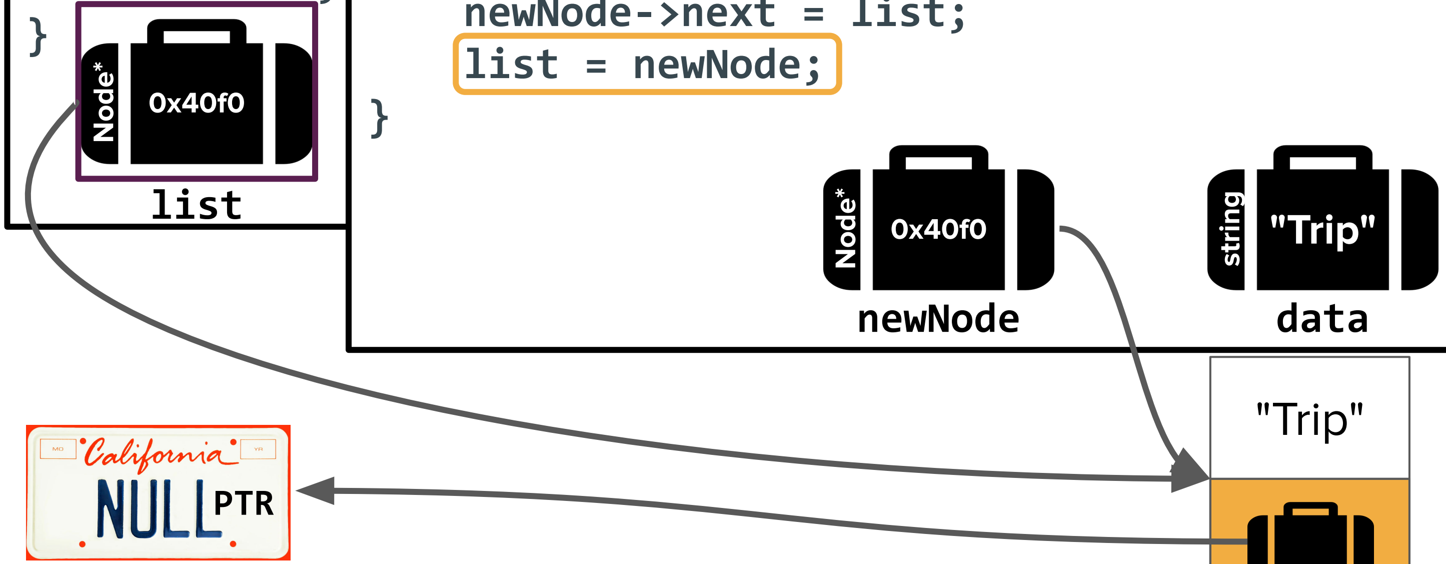

```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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





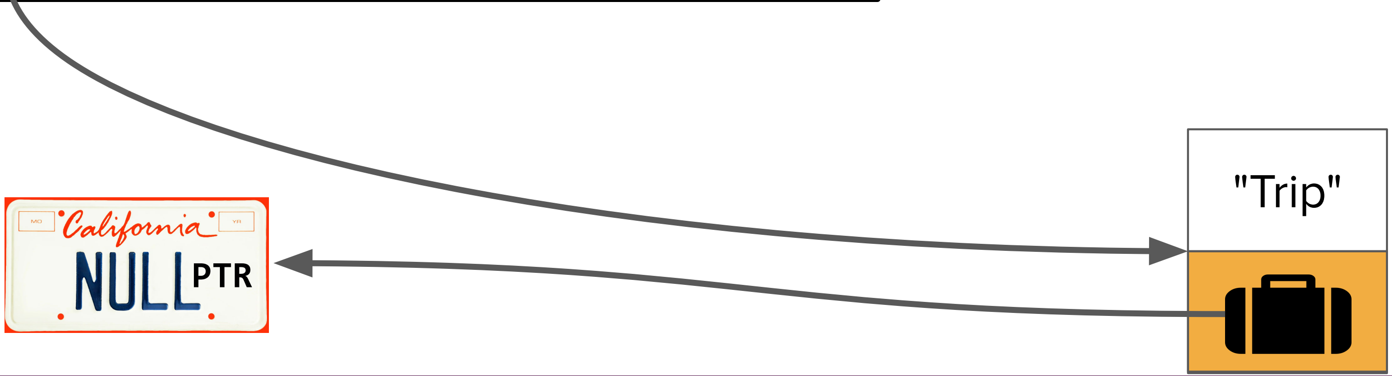
```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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

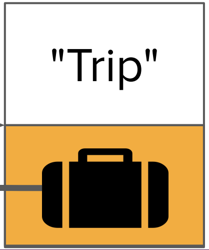


```
int main() {
    Node* list = nullptr;
    prependTo(list, "Trip");
    prependTo(list, "Kylie");
    prependTo(list, "Nick");
    return 0;
}
```

list



```cpp
int main() {
    Node* list = nullptr;
    prependTo(list, "Trip");
    prependTo(list, "Kylie");
    prependTo(list, "Nick");
    return 0;
}
```



Pointers by Reference Summary

- If you pass a pointer into a function by *value*, you can change the contents at the object you point at, but not *which* object you point at.

- If you pass a pointer into a function by *reference*, you can *also* change *which* object is pointed at.

- When passing in pointers by reference, be careful not to change the pointer unless you really want to change where it's pointing!

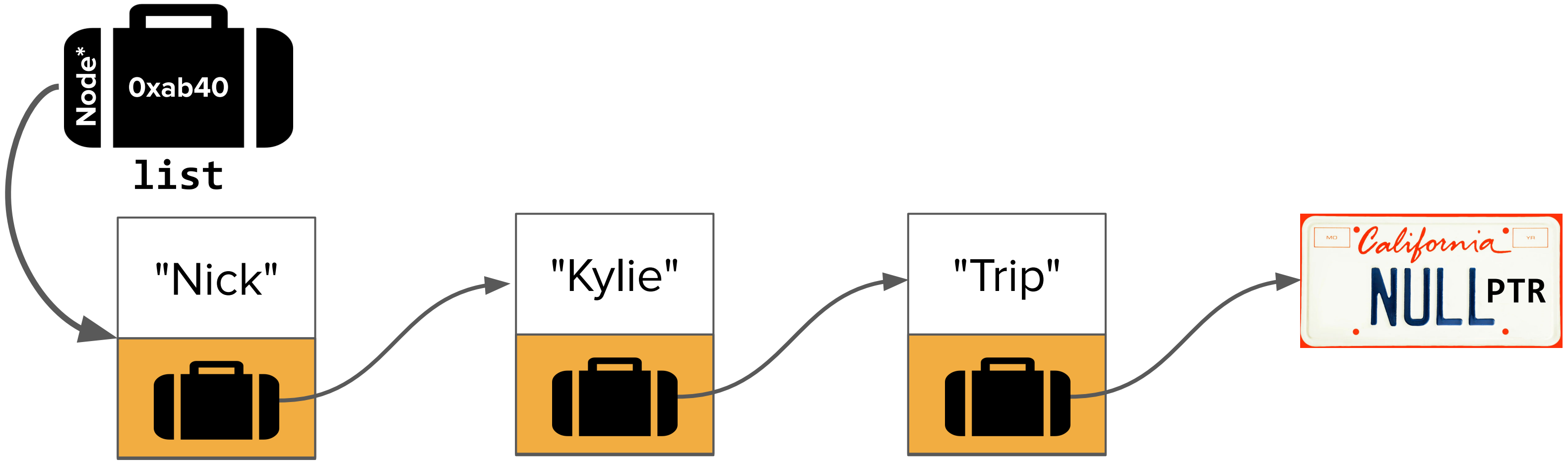
Insertion at the end (append)

Appending an Element

- Suppose we wanted to write a function to add an element to the end of a linked list.

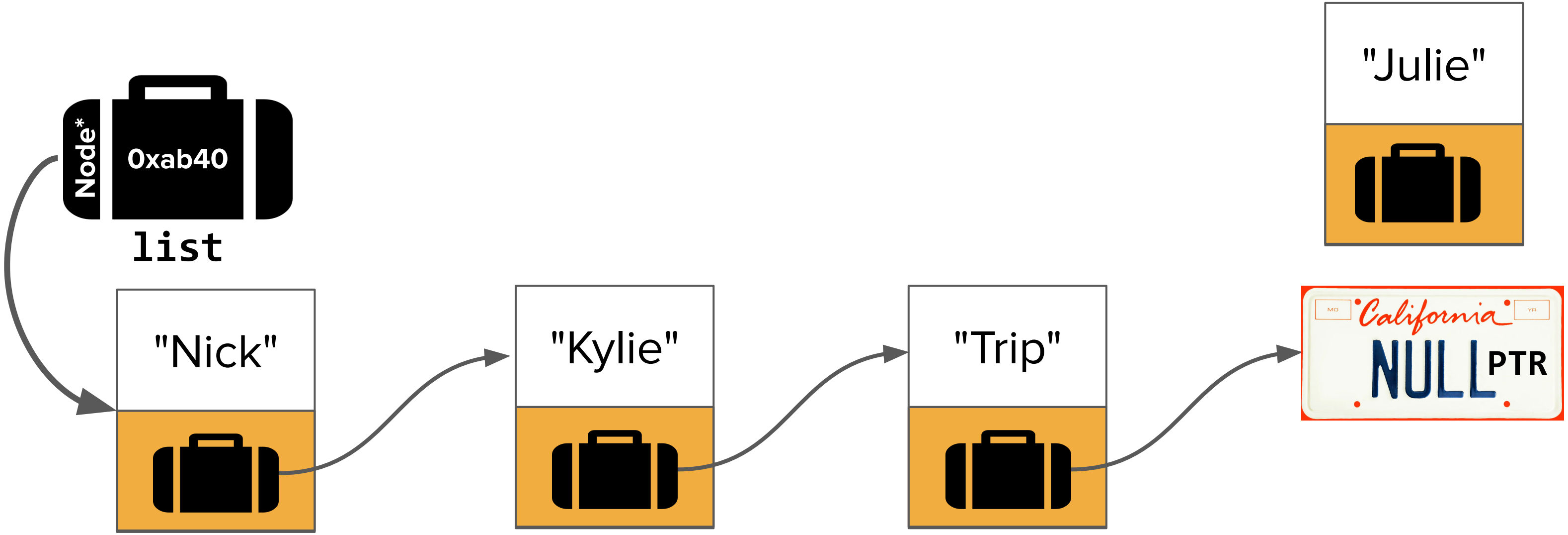
Appending an Element

- Suppose we wanted to write a function to add an element to the end of a linked list.



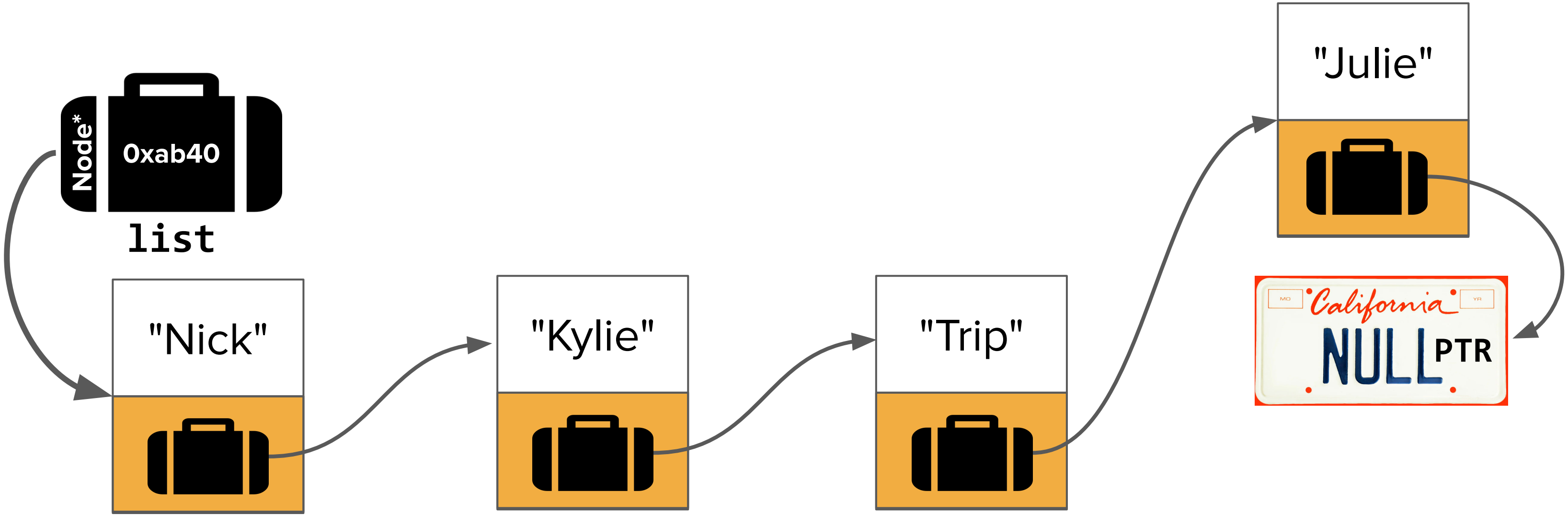
Appending an Element

- Suppose we wanted to write a function to add an element to the end of a linked list.



Appending an Element

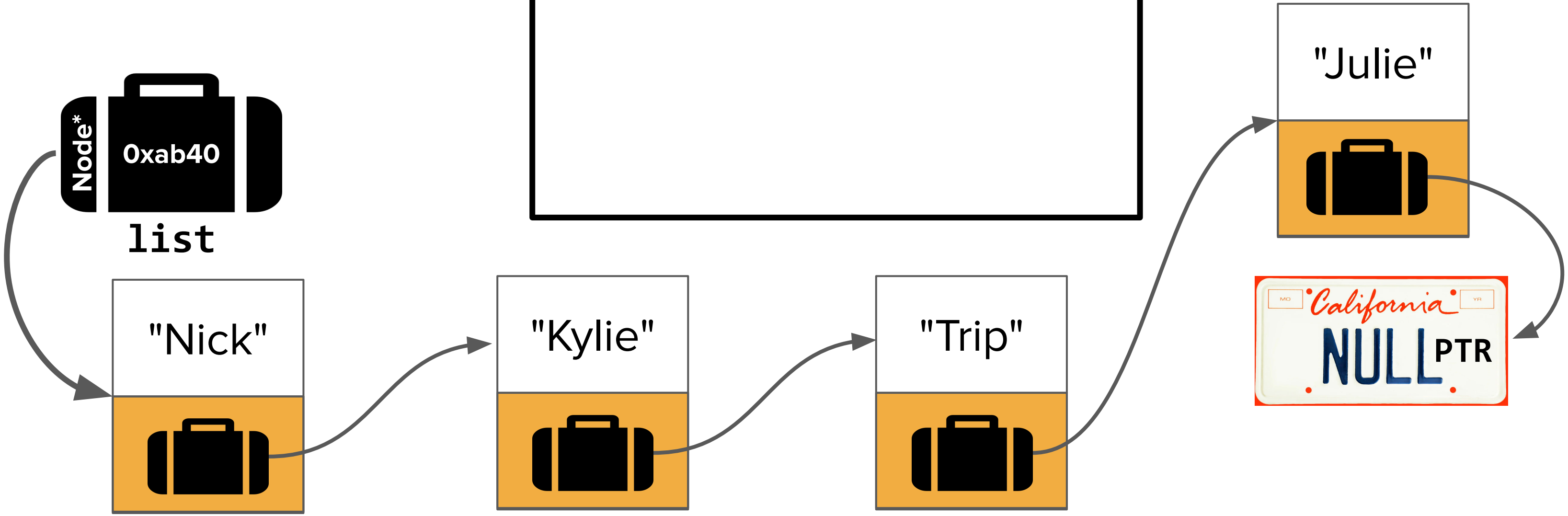
- Suppose we wanted to write a function to add an element to the end of a linked list.



Appending an Element

- Suppose we wanted to write a function to add an element to the end of a linked list.

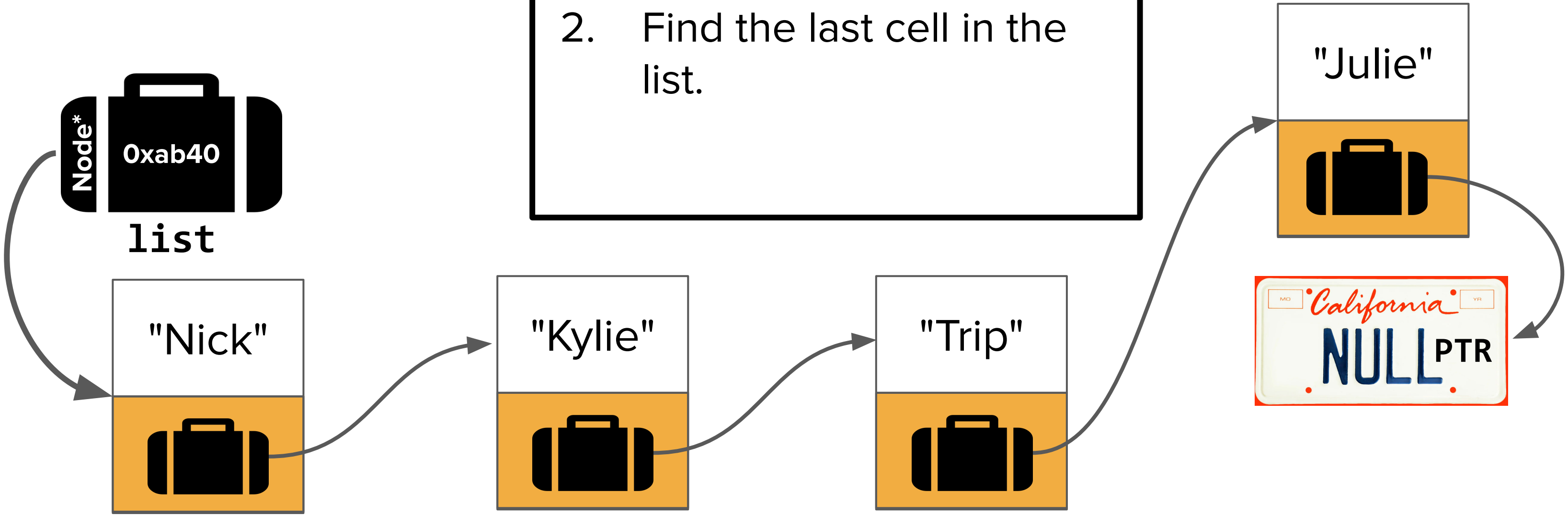
1. Create a cell whose **next** field is nullptr.



Appending an Element

- Suppose we wanted to write a function to add an element to the end of a linked list.

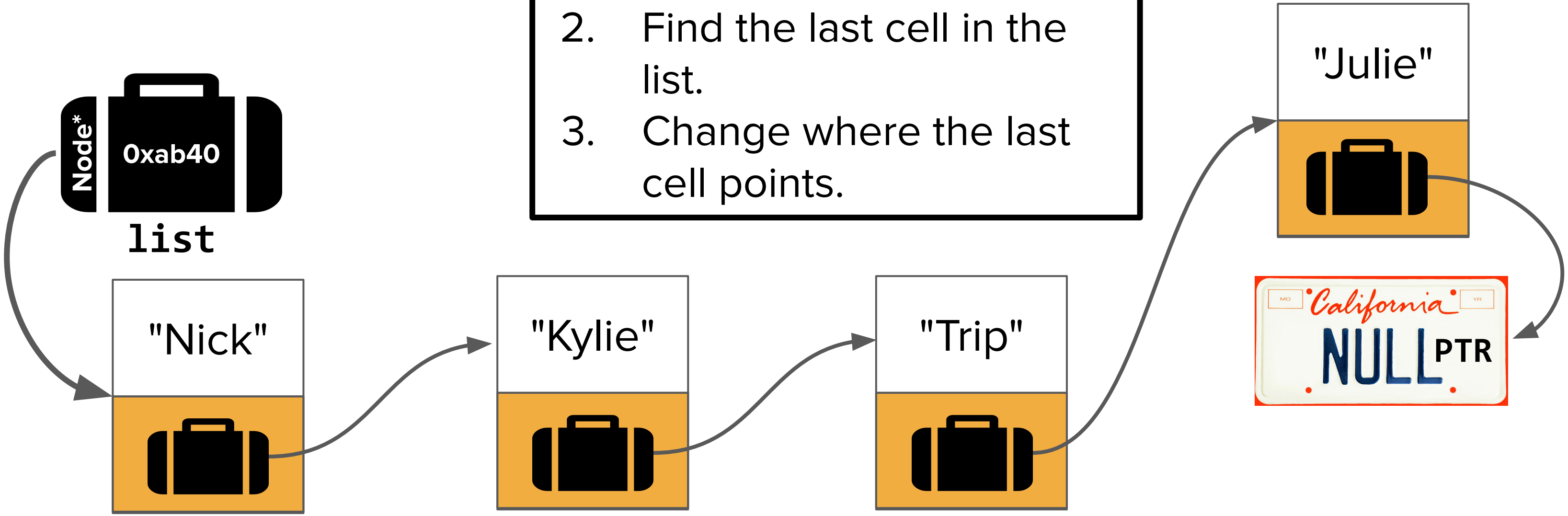
1. Create a cell whose **`next`** field is nullptr.
2. Find the last cell in the list.



Appending an Element

- Suppose we wanted to write a function to add an element to the end of a linked list.

1. Create a cell whose **next** field is nullptr.
2. Find the last cell in the list.
3. Change where the last cell points.



appendTo()

Let's code it!

`appendTo()` Takeaways

- Appending to the end of a linked list has a lot of tricky edge cases!
    - We must pass the pointer by reference to account for the case where we're adding to an empty list and need to update the head pointer.
    - We have to be careful about our while loop condition to make sure that we never dereference a null pointer!
    - We have to be careful with our usage of pointers by reference and make sure to maintain a local iterator pointer to traverse the list.

- Being able to reason about all of these cases becomes much easier if we draw out diagrams and carefully trace the values of different pointers over time.
    - Note: Check out slides 56-124 of this slide deck for visualizations of the right and wrong ways of coding up the append function!

Unresolved Issue

- What is the big-O complexity of appending to the back of a linked list using our algorithm?

Unresolved Issue

- What is the big-O complexity of appending to the back of a linked list using our algorithm?

- **Answer:** `O(n)`, where n is the number of elements in the list, since we have to find the last position each time.

Unresolved Issue

- What is the big-O complexity of appending to the back of a linked list using our algorithm?

- **Answer:** `O(n)`, where n is the number of elements in the list, since we have to find the last position each time.

- This seems suspect – `O(n)` for a single insertion is pretty bad! Can we do better?
    - Find out tomorrow!

Summary

Summary

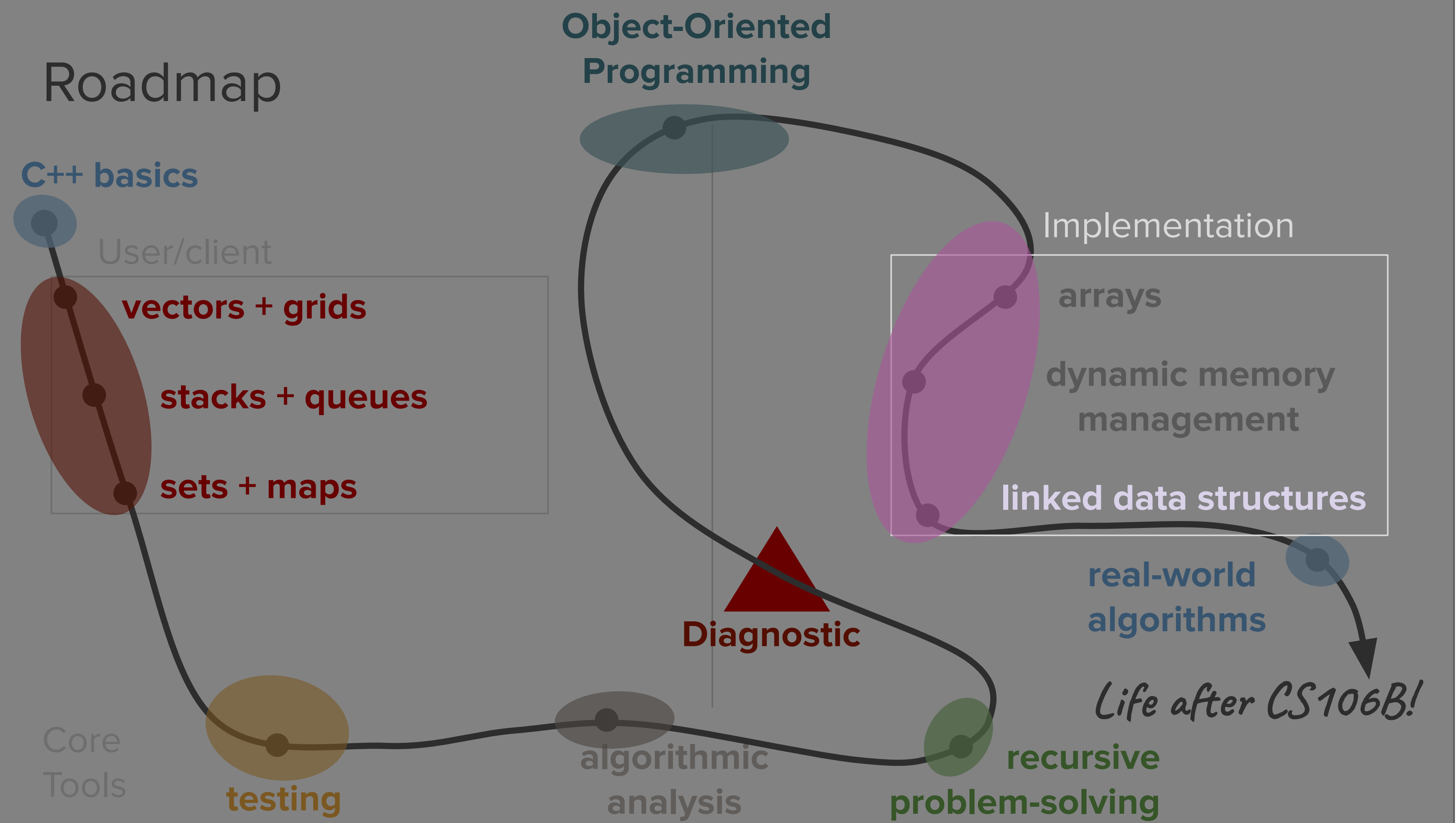
- Linked lists can be used outside classes - you’ll do this on Assignment 5!

- Think about when you want to pass pointers by reference in order to edit the original pointer and to avoid leaking memory.

- We can add to a linked list by either prepending or appending.
  - Prepending is faster but results in a reversed order of items (things added earlier are at the back of the list)
  - Appending (as we’ve learned so far) requires traversing all items but maintains order (things added earlier are at the front of the list)

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

Finish up Linked Lists and start Intro to Sorting!