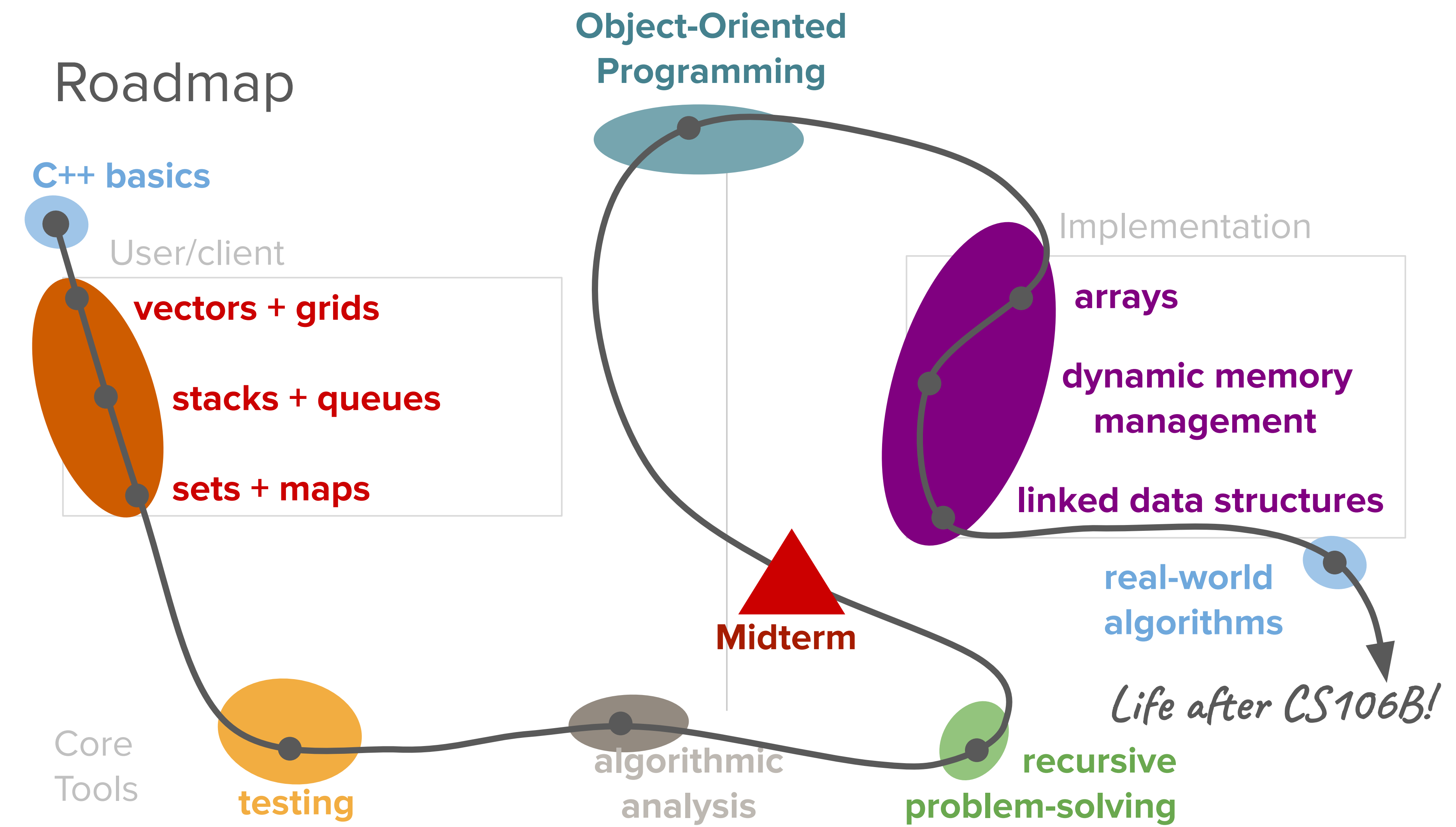
Linked List Operations

Get creative: If you were explaining a linked list to a friend who isn't in CS106B, what would you use as a good analogy?

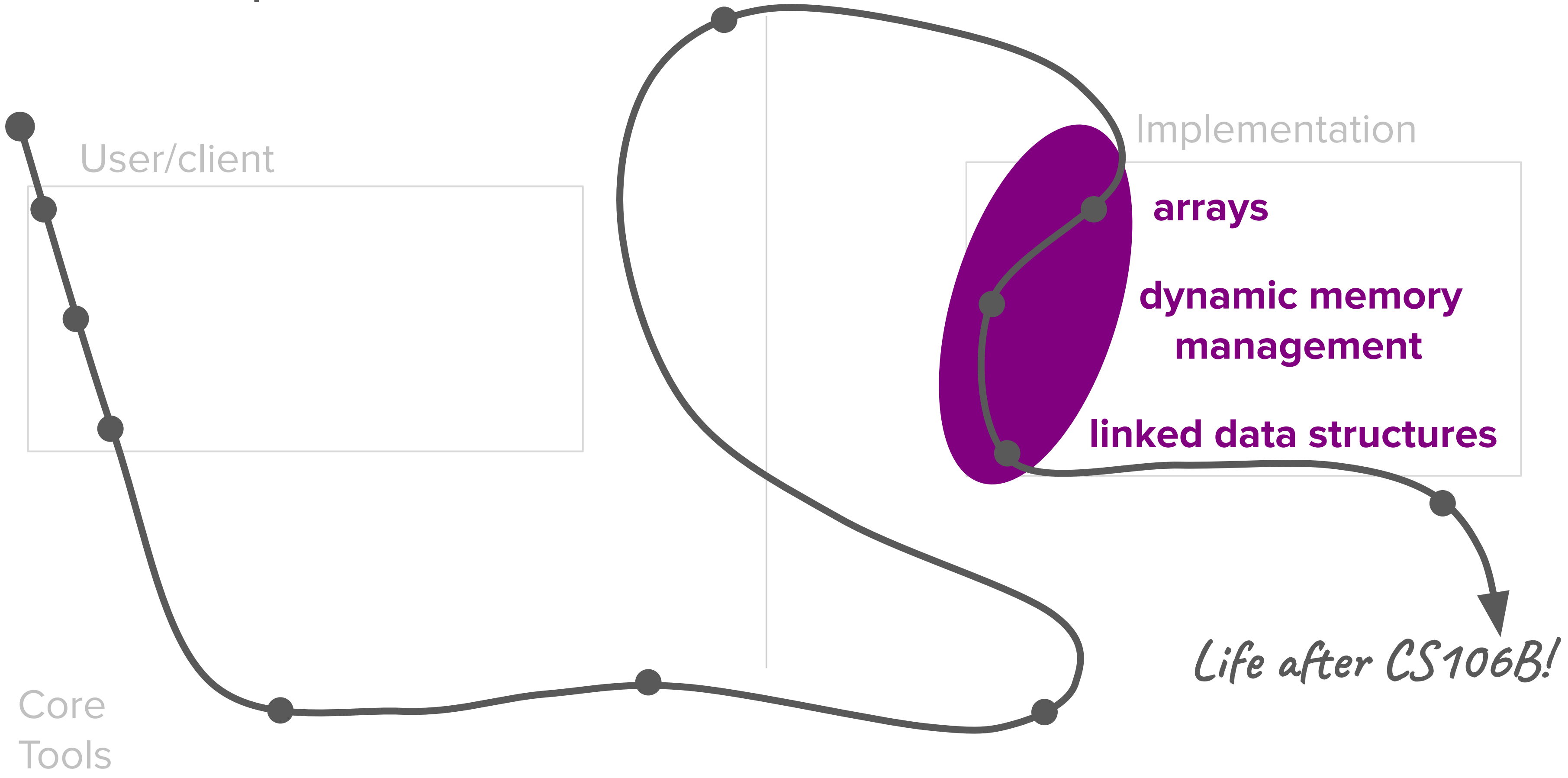
https://pollev.com/cs106bpolls

If you were explaining a linked list to a friend who isn't in CS106B, what would you use as a good analogy?

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
Life after CS106B!
Core Tools

Today's question

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

Today's topics

1. Review
2. Advanced Linked List Operations
    a. Traversal
    b. Insertion (multiple ways!)
    c. Deletion (if time)

Review

[intro to linked lists]

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)

Fundamental C++ Data Storage

Pointers move us across this boundary!

How is data represented electronically?
(RAM)

Computer Hardware

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?

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?

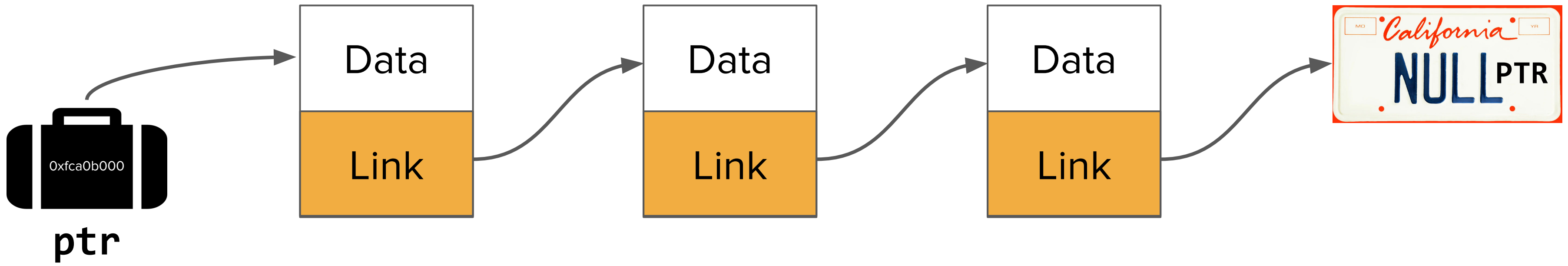
- A linked list is a **chain of nodes**, used to store a sequence of data.

- 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.

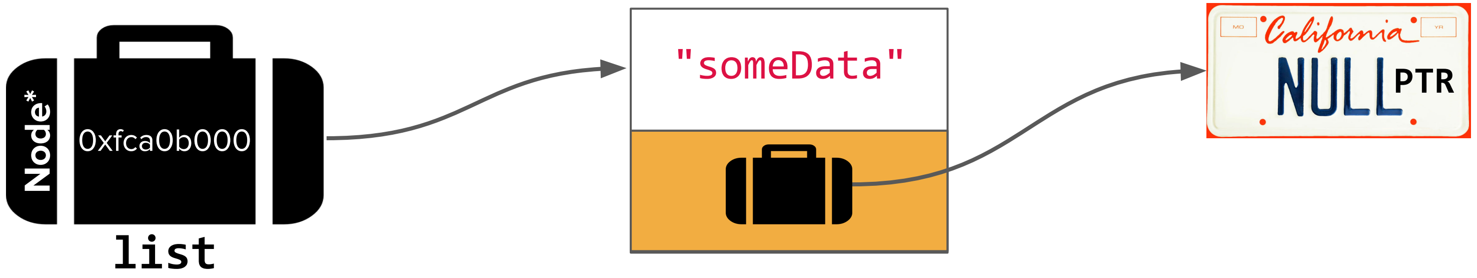
A linked list!



The **Node** struct

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

Pointer to a node

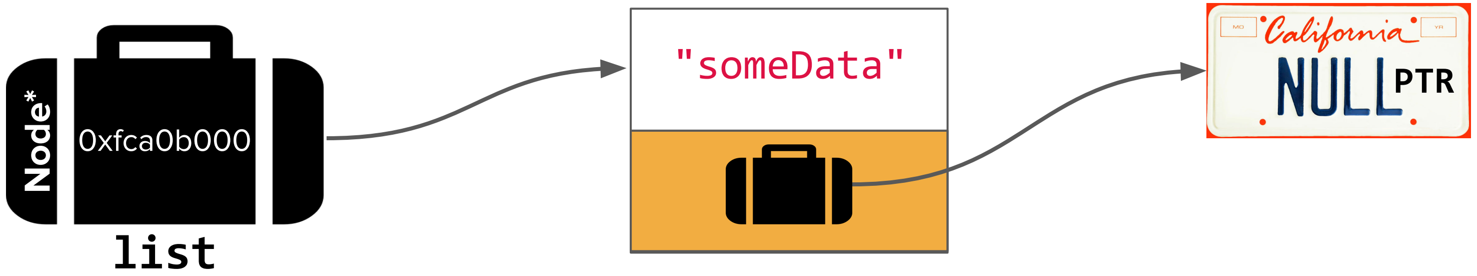


```
Node* list = new Node;
list->data = "someData";
list->next = nullptr;
```

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

Linked list traversal

- 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.

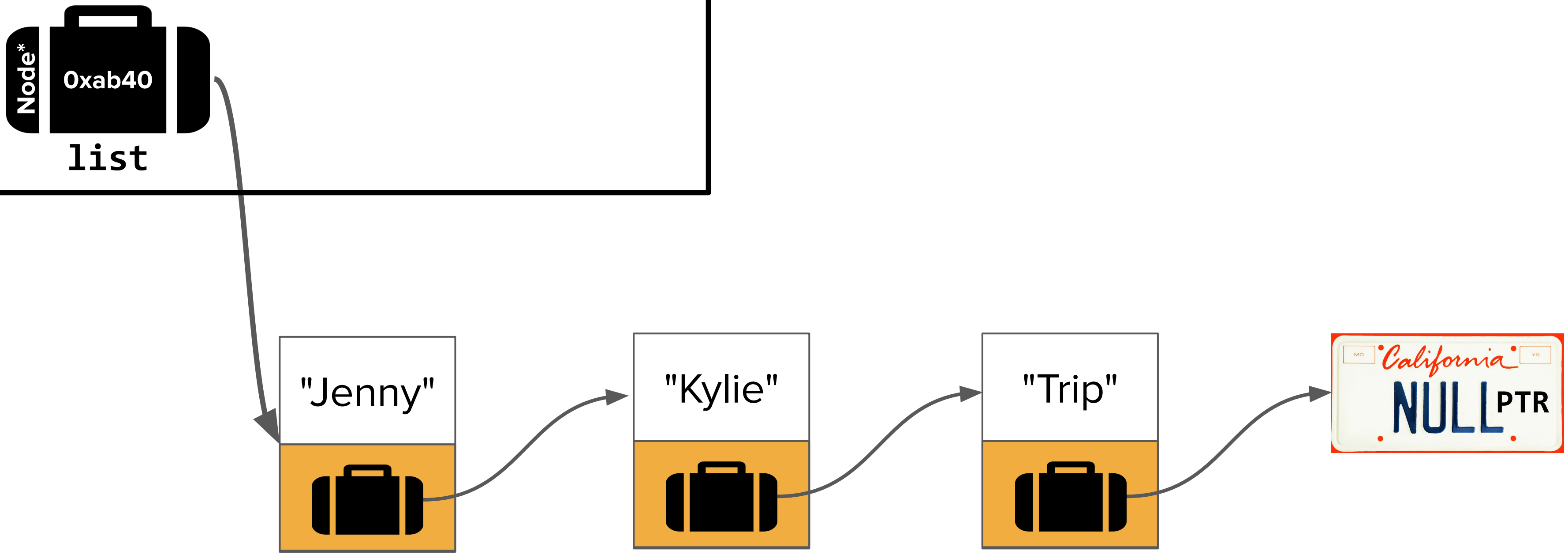
```
printList()
```

How does it work?

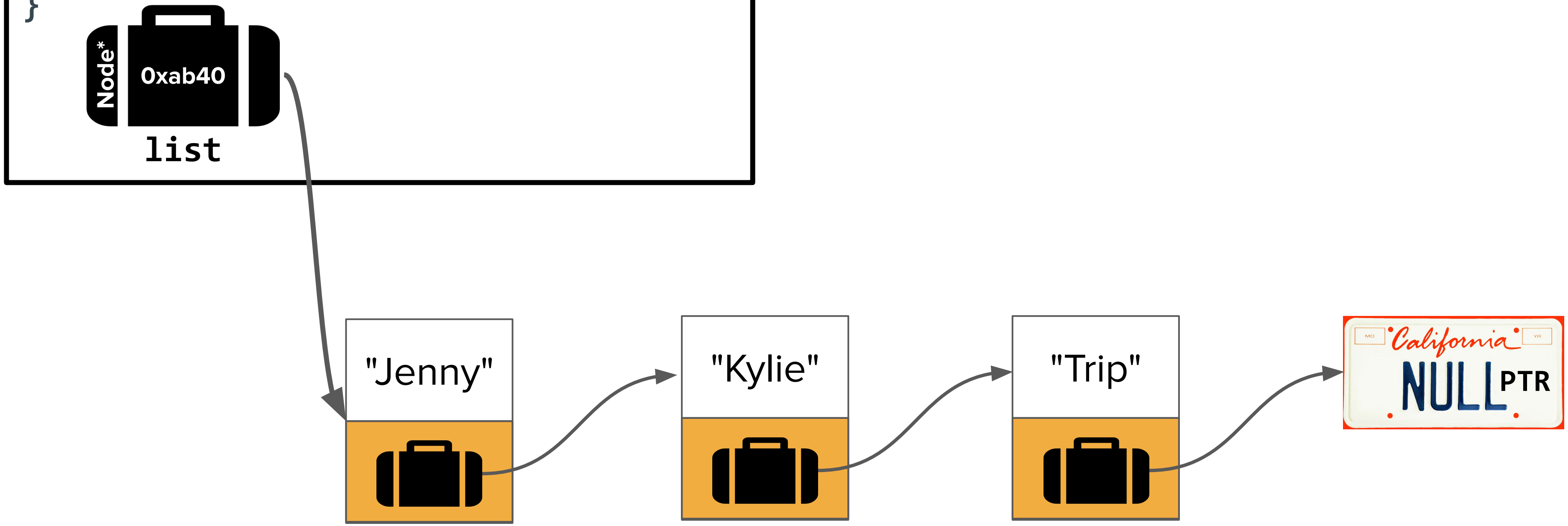
```
int main() {
    Node* list = readList();
    printList(list);

    /* 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... */
}
```

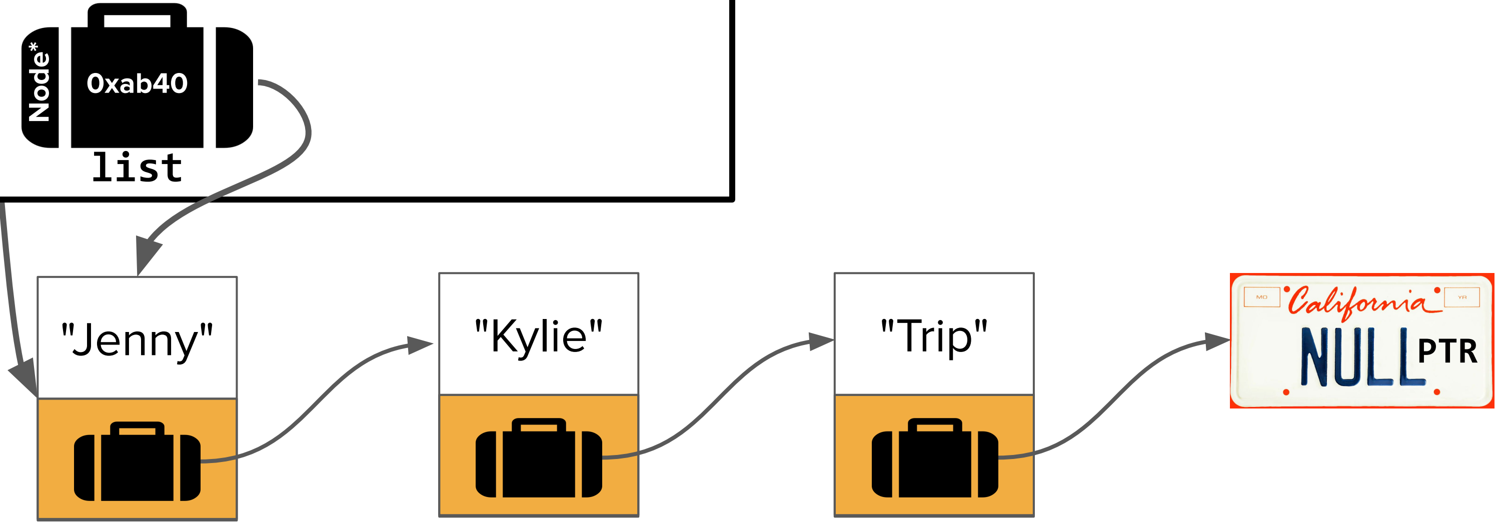


```
int main() {
    Node* list = readList();
    printList(list);

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



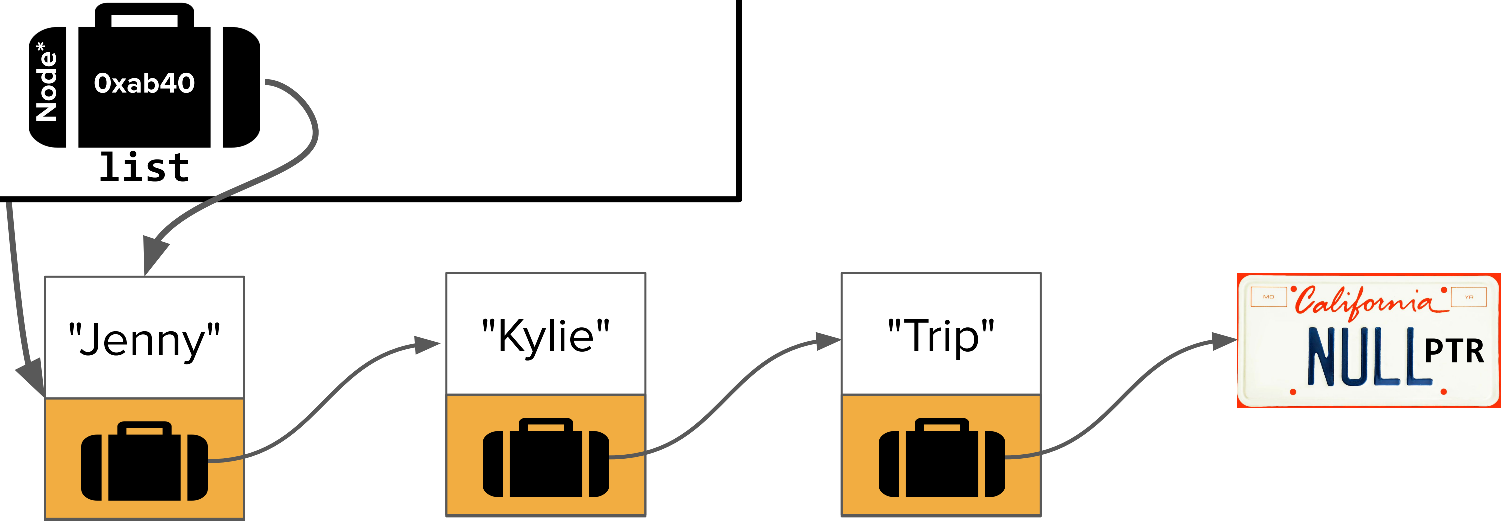
```
int main() {

}
```

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



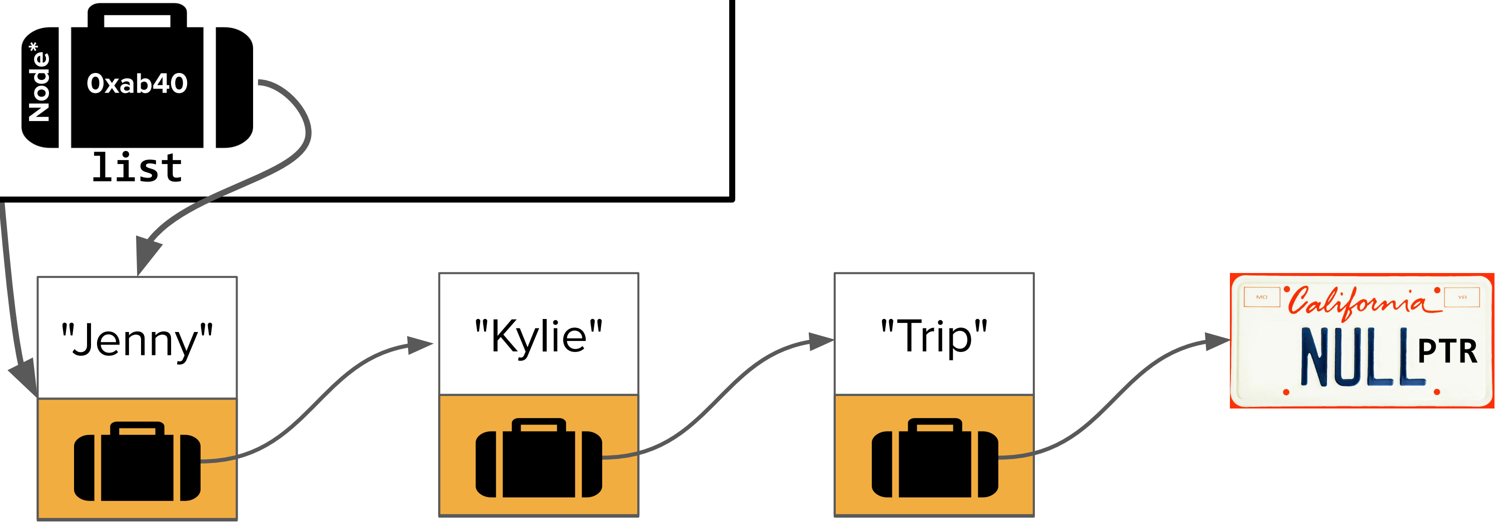
```
int main() {

}
```

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



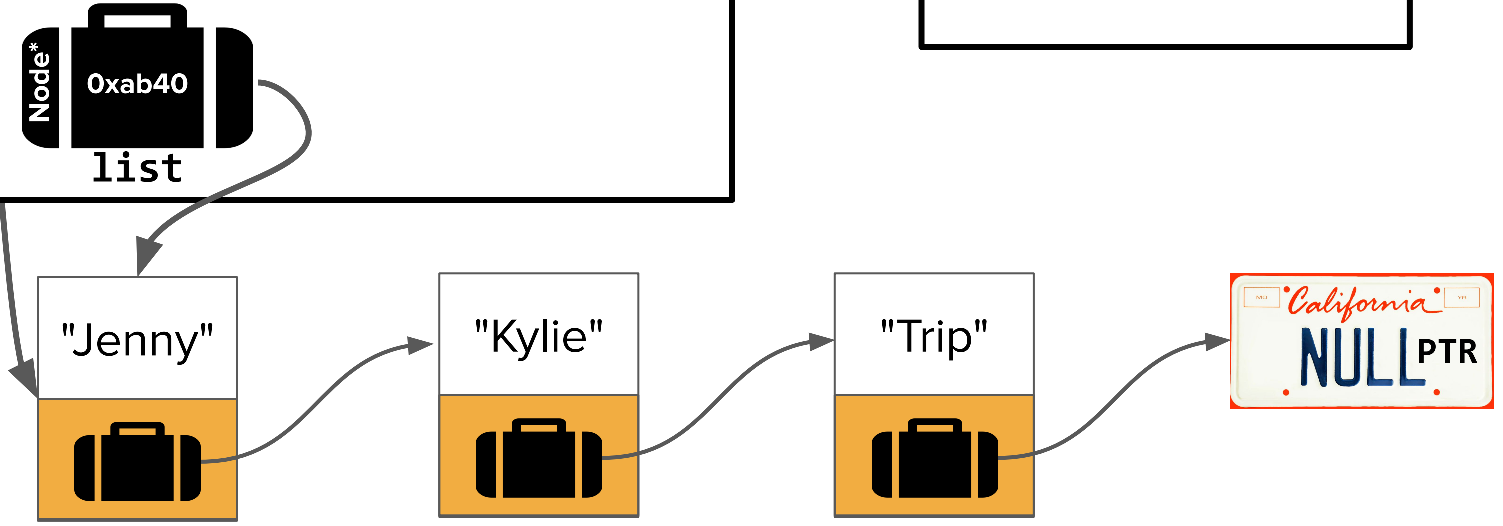
```
int main() {

}
```

```
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;
    }
}
```

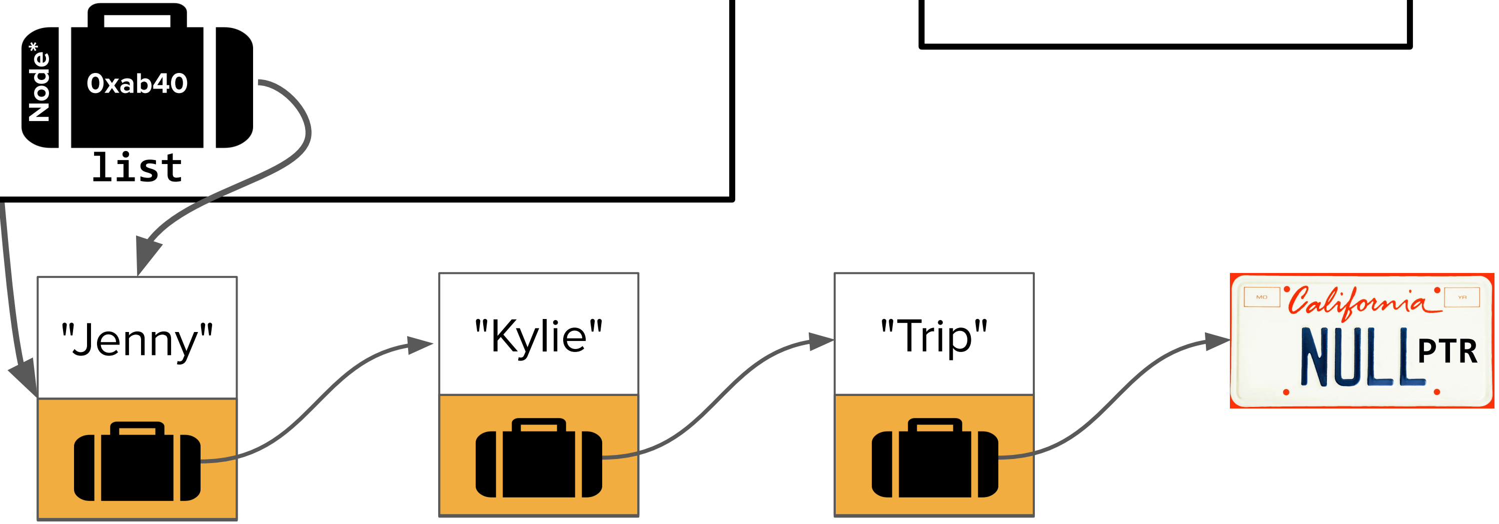


```
Jenny
```

```
int main() {

}
```

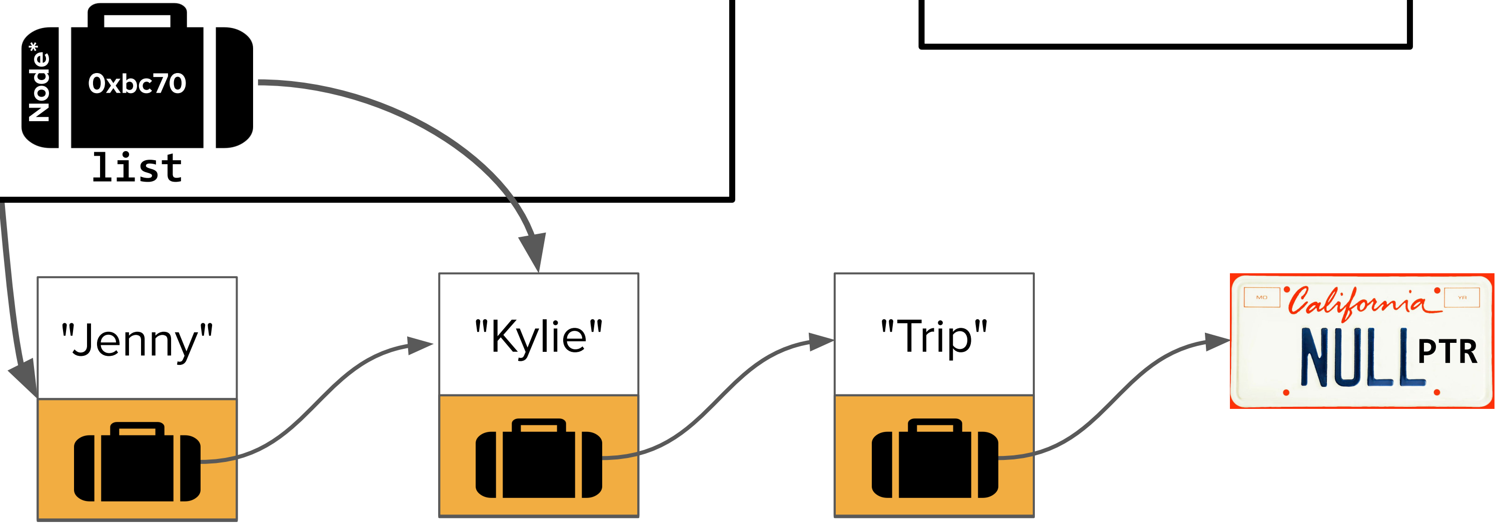
```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```

```
Jenny
```



```
int main() {

}
```

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

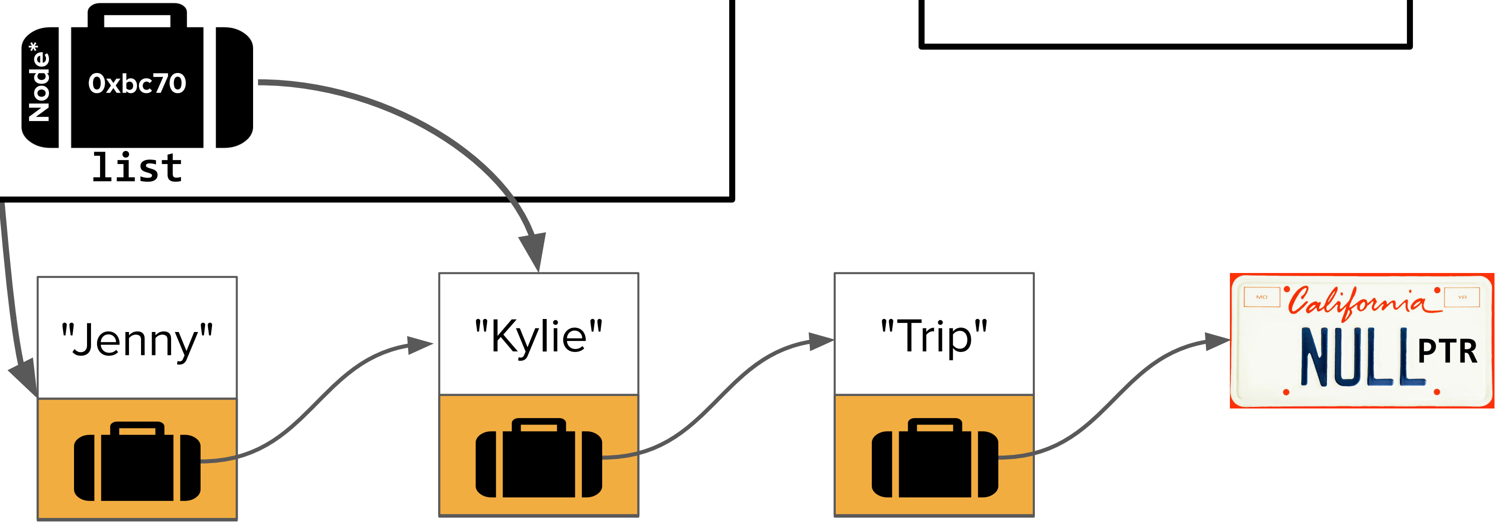


```
Jenny
```

```
int main() {

}
```

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

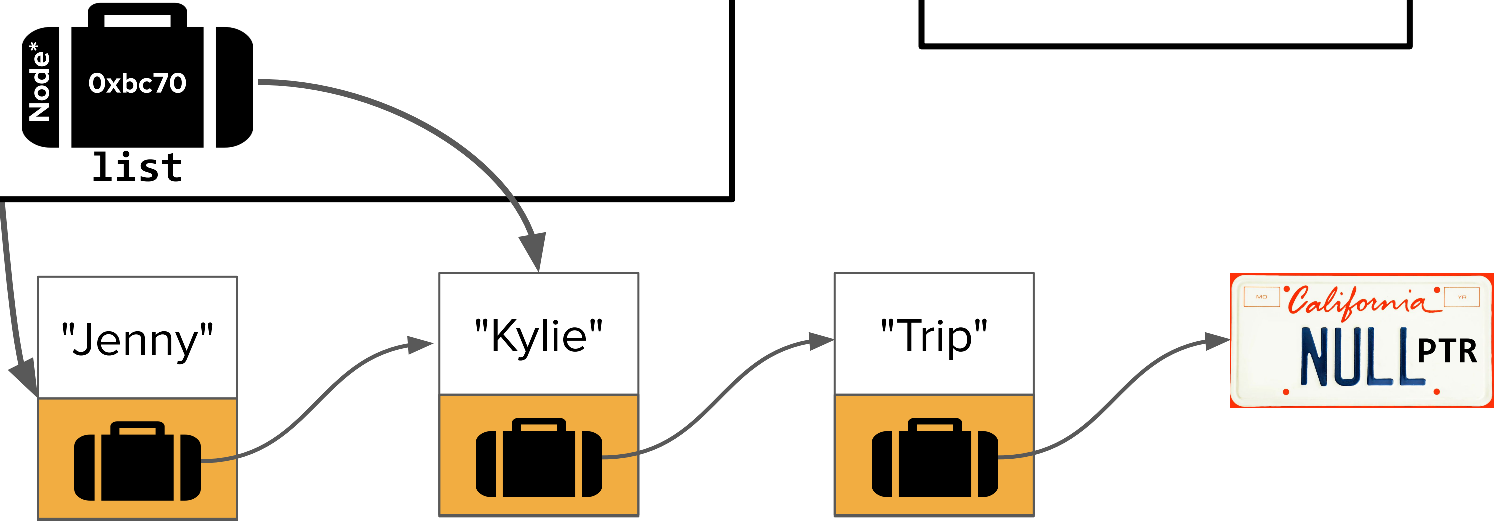
```
Jenny
```



```
int main() {

}
```

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

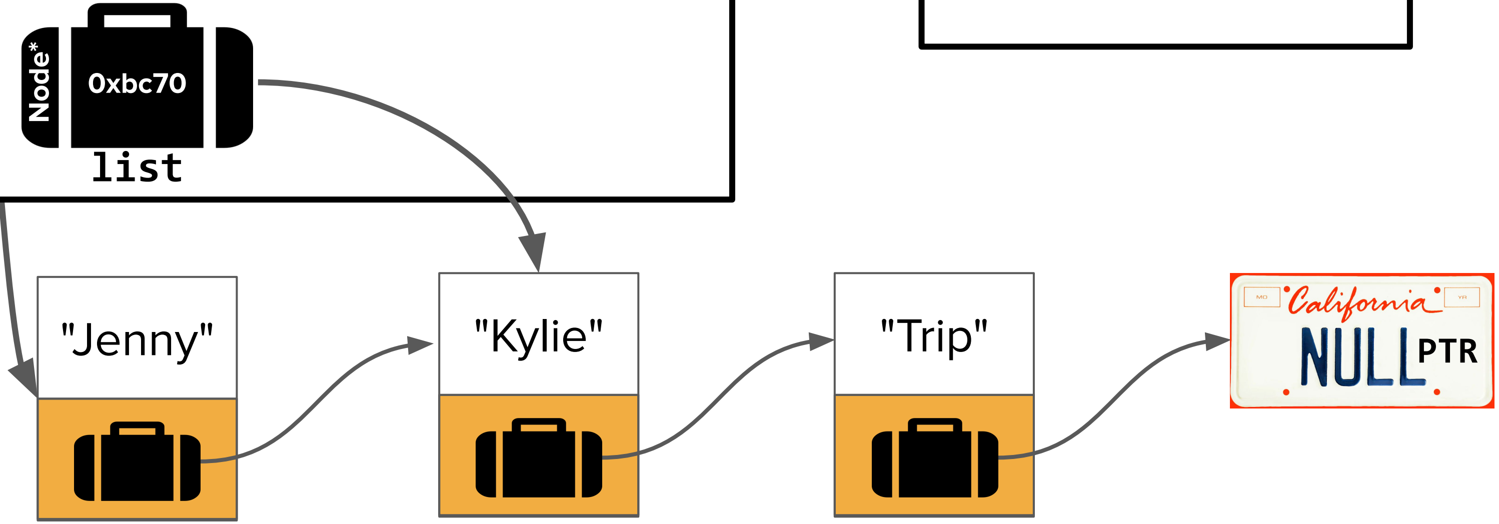
```
Jenny
```



```
int main() {

}
```

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

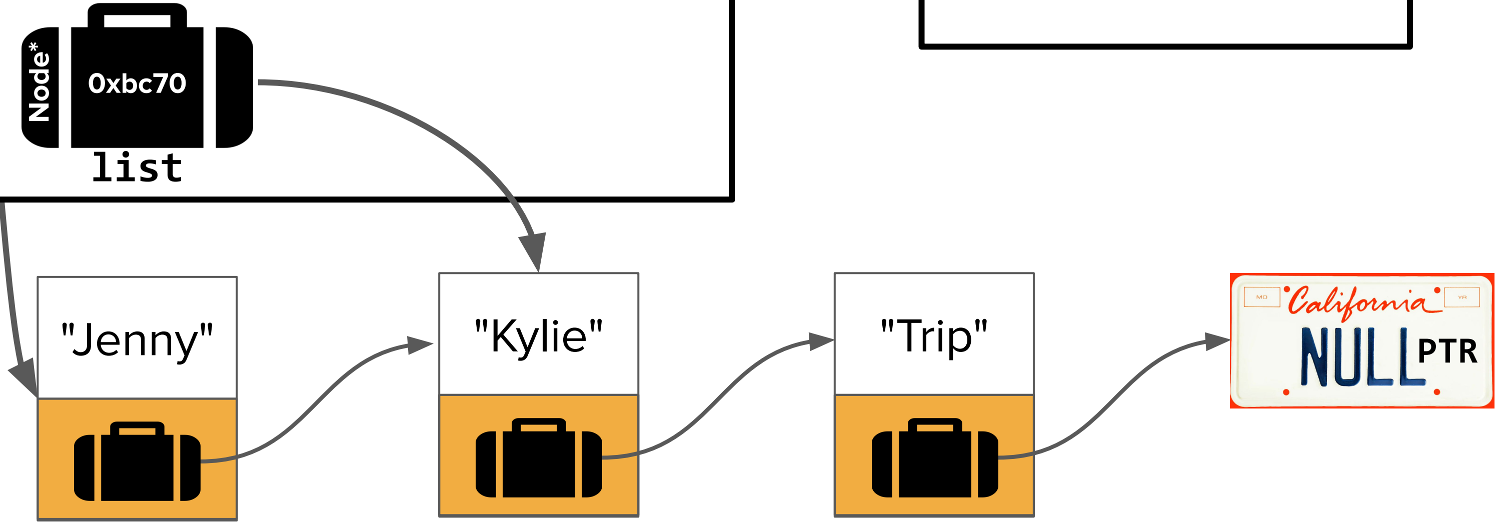
```
Jenny
Kylie
```



```
int main() {

}
```

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

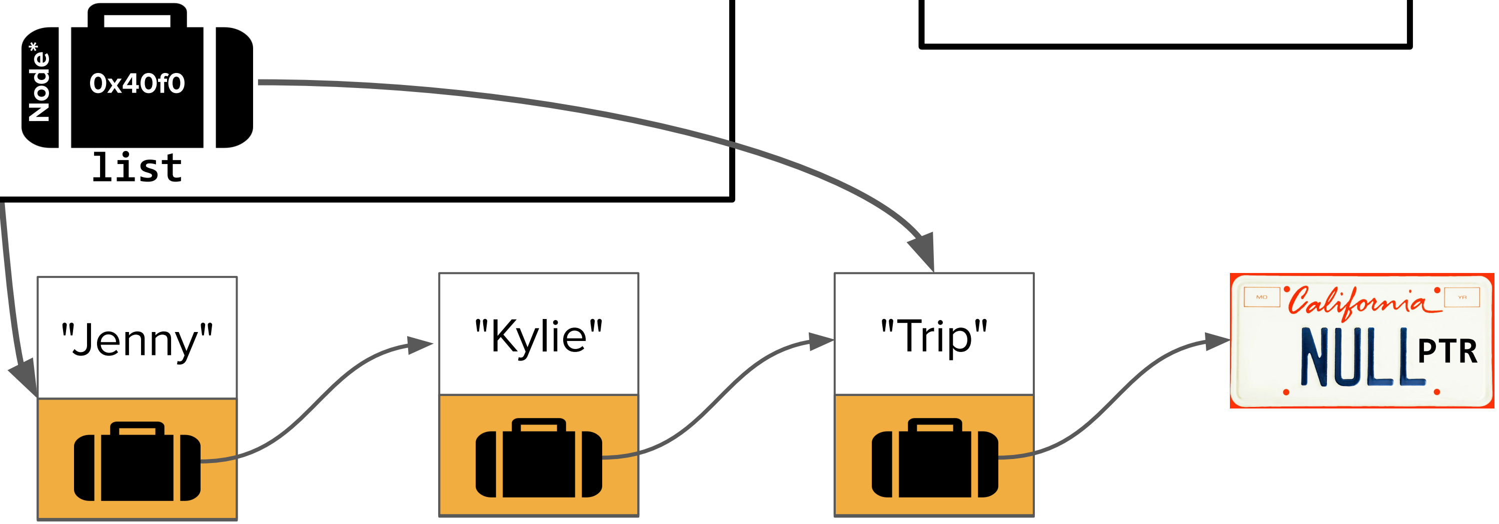
```
Jenny
Kylie
```



```
int main() {

}
```

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

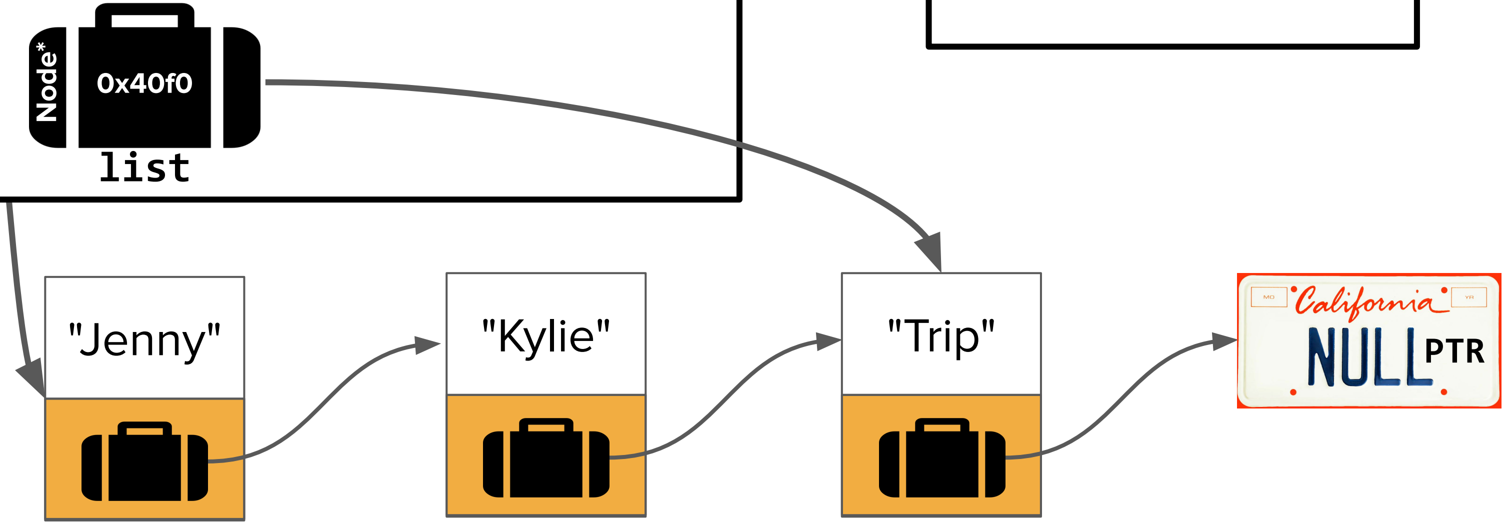
```
Jenny
Kylie
```



```
int main() {

}
```

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

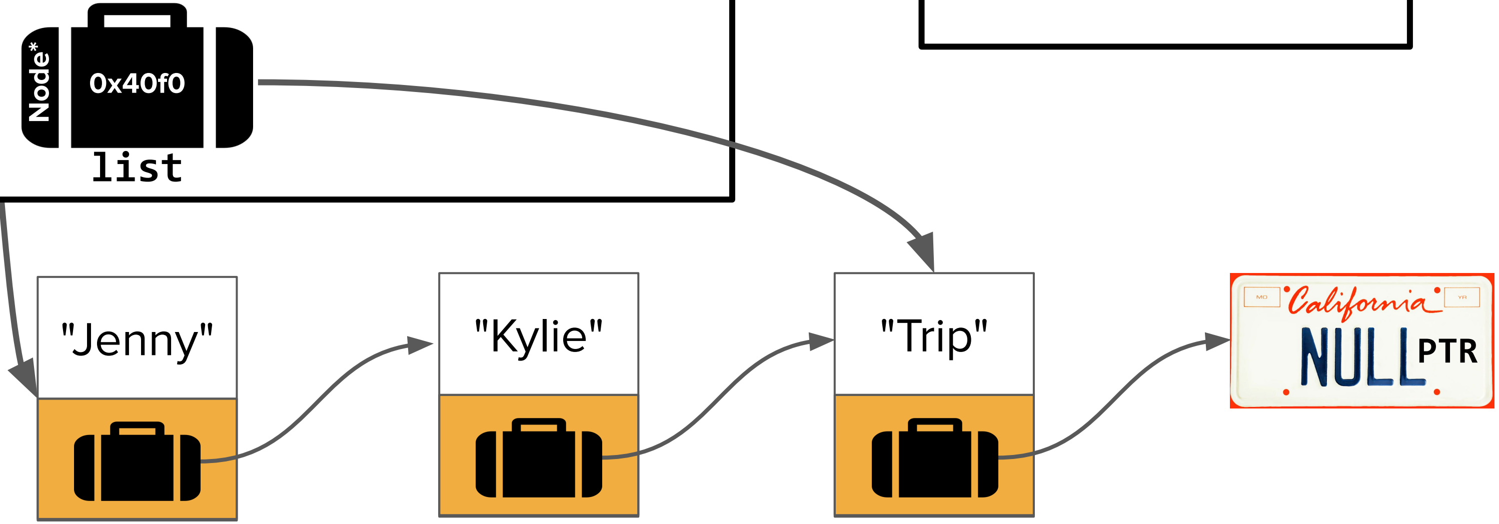
```
Jenny
Kylie
```



```
int main() {

}
```

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

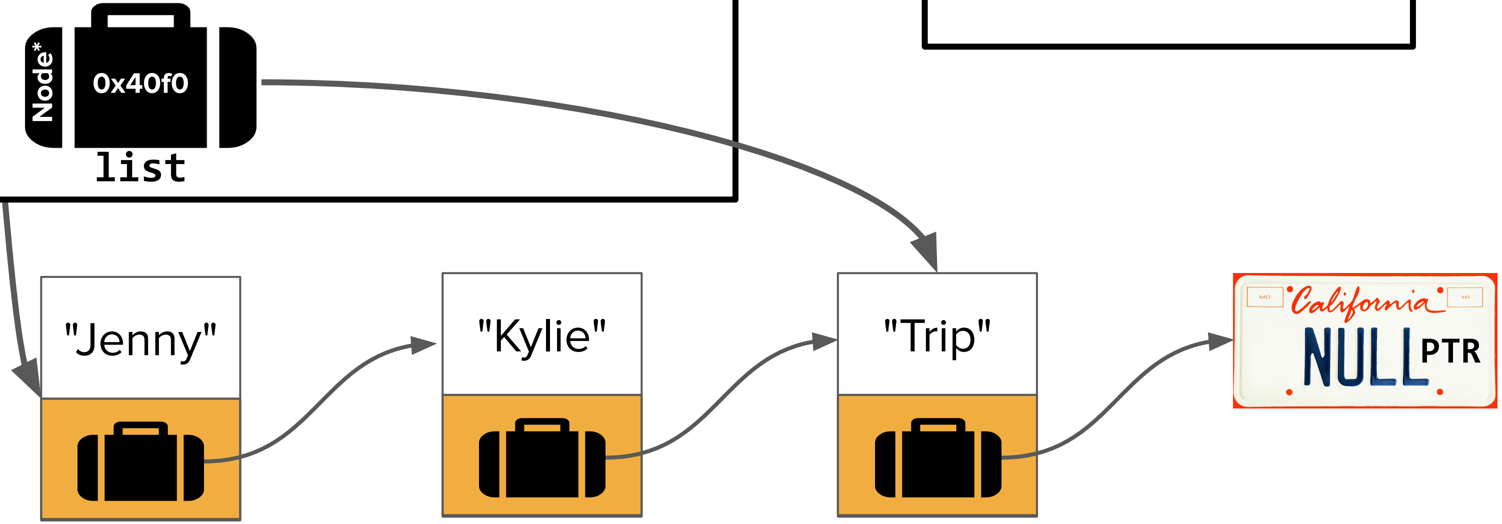
```
Jenny
Kylie
```



```
int main() {

}
```

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

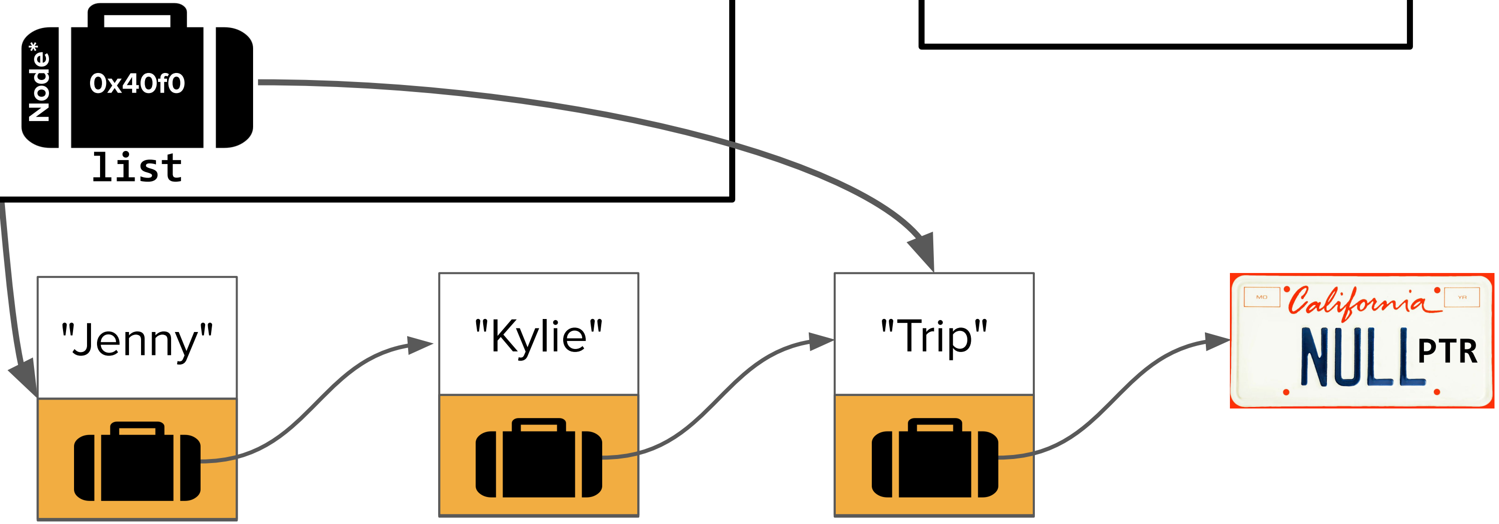
```
Jenny
Kylie
Trip
```



```
int main() {

}
```

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

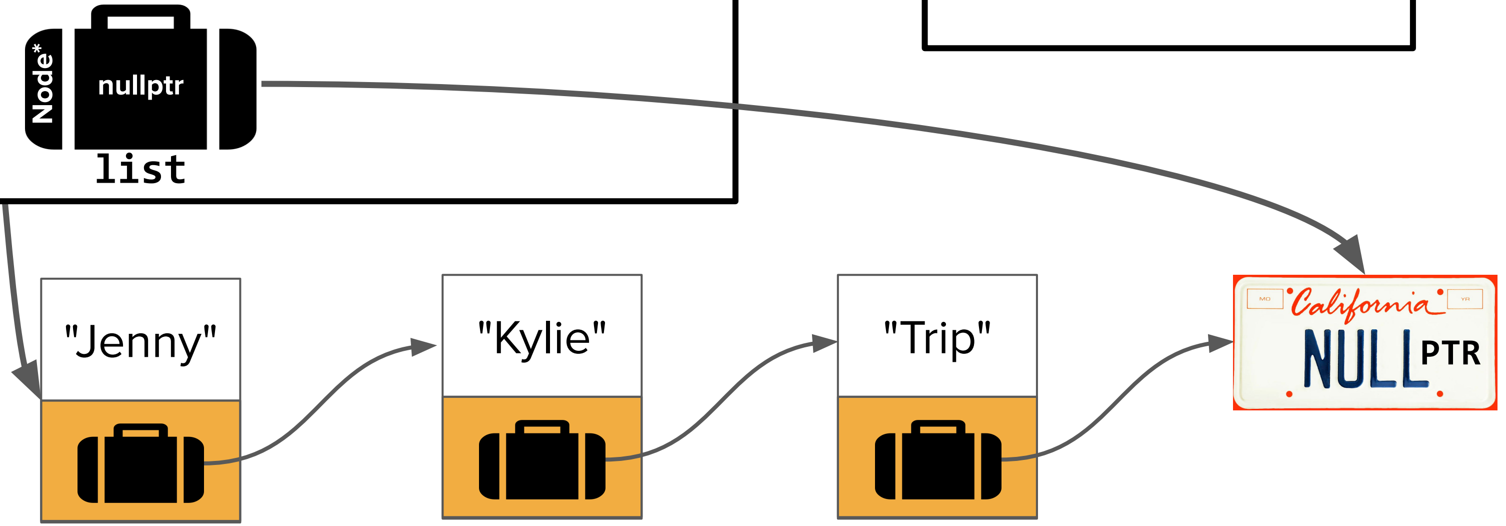
```
Jenny
Kylie
Trip
```



```
int main() {

}
```

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

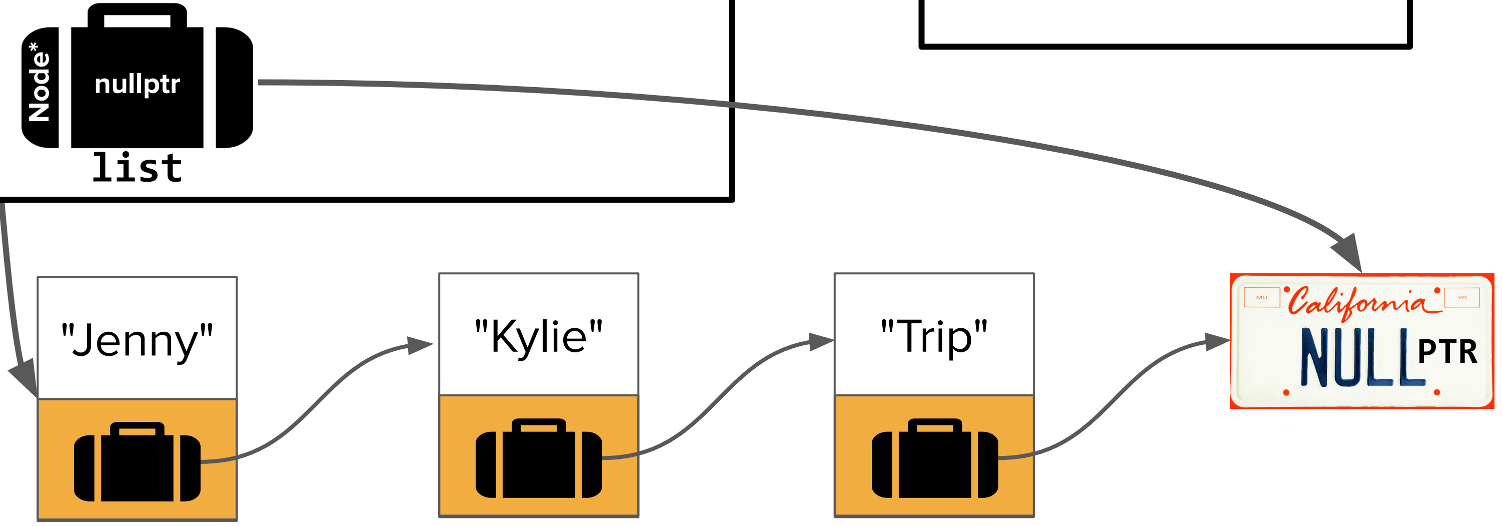
```
Jenny
Kylie
Trip
```



```
int main() {

}
```

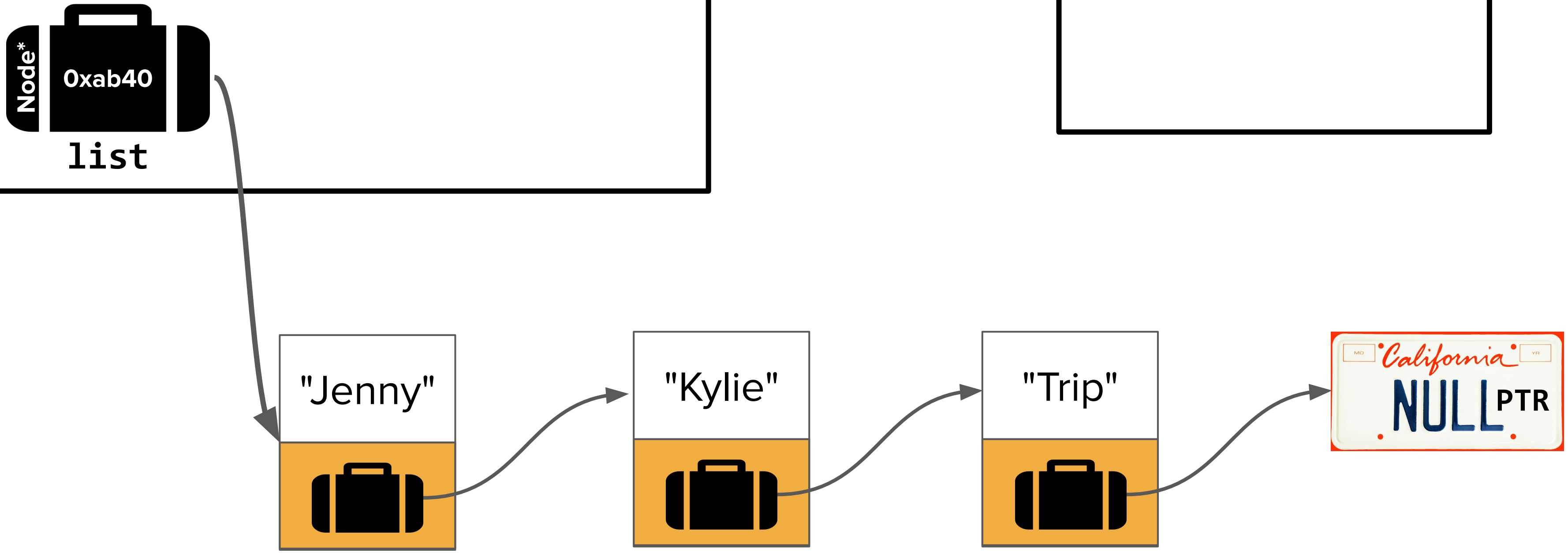
```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```

```
Jenny
Kylie
Trip
```



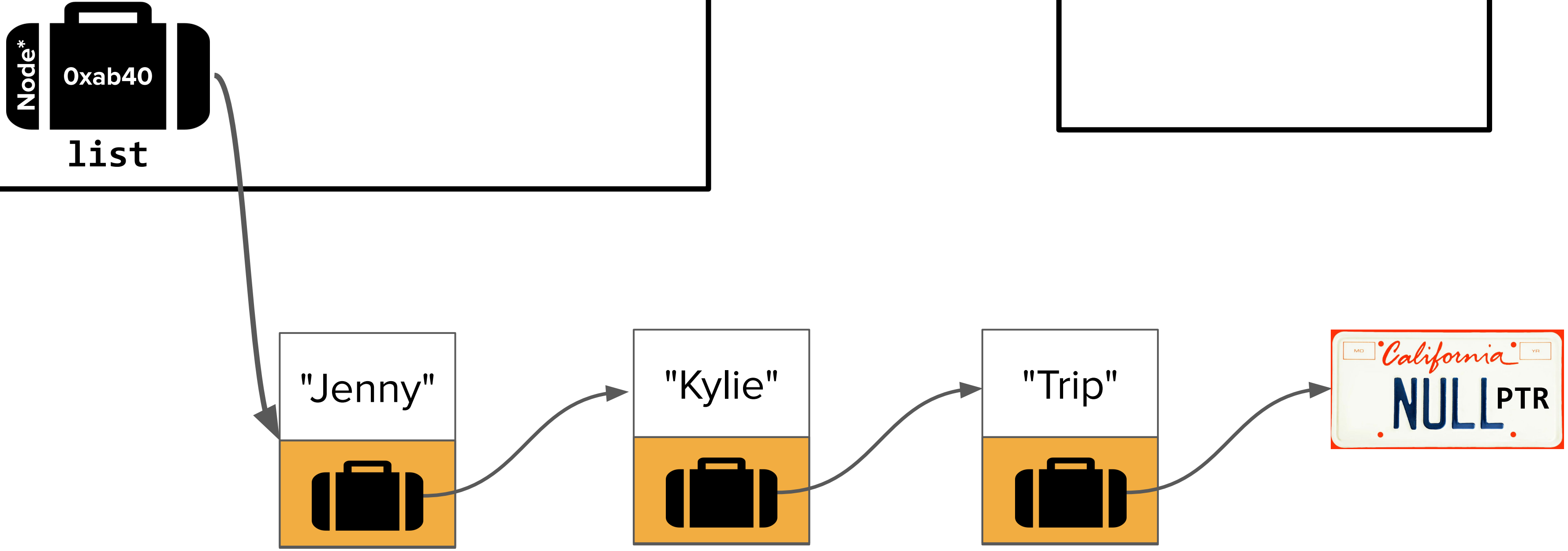
```
int main() {
    Node* list = readList();
    printList(list);

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

```
Jenny
Kylie
Trip
```



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

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

```
Jenny
Kylie
Trip
```



Summary

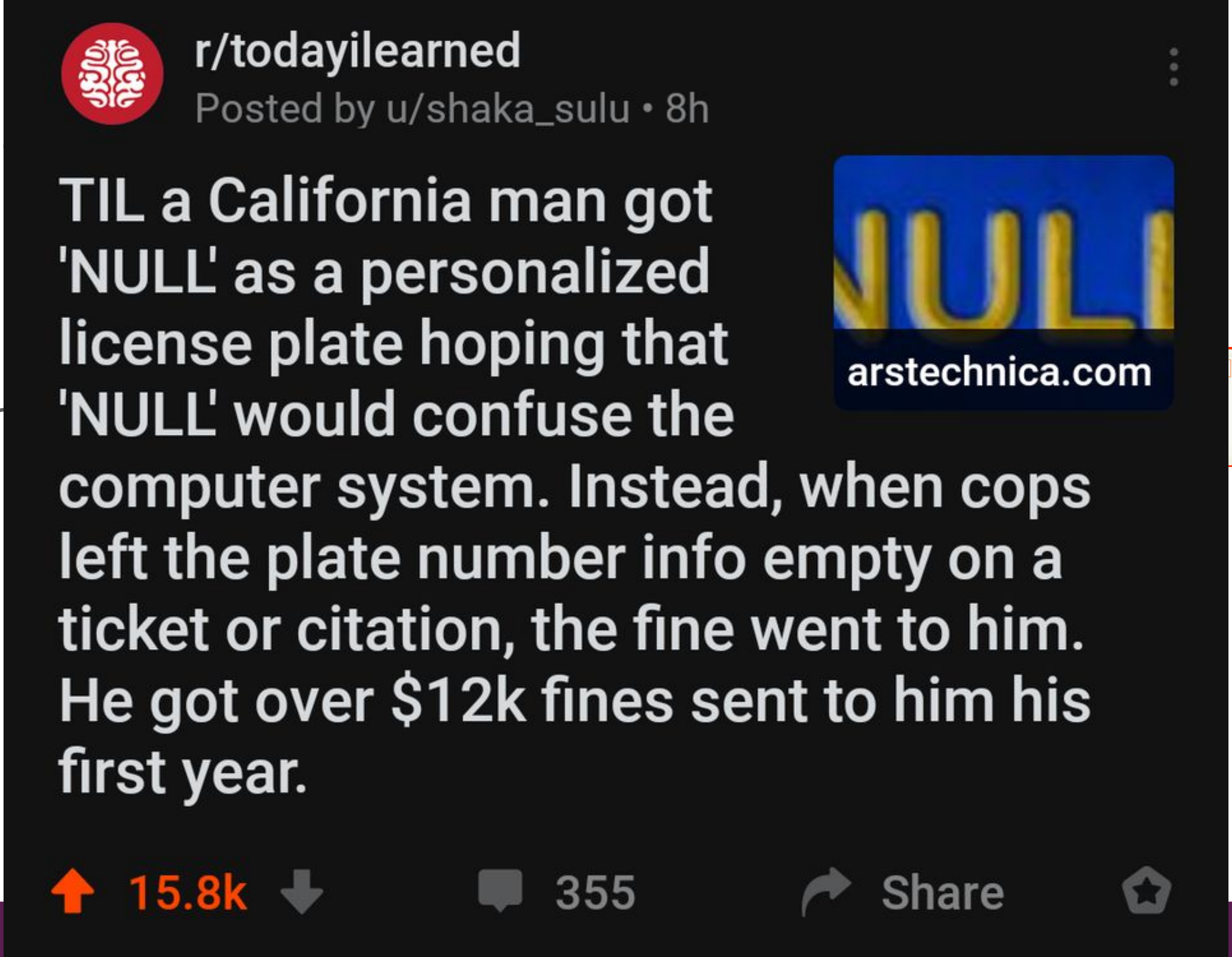
- Linked lists are chains of Node structs, which are connected by pointers.
    - Since the memory is not contiguous, they allow for fast rewiring between nodes (without moving all the other Nodes like an array might).

- 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)
    - Use **`nullptr`** wisely!

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

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

Linked List Operations Revisited

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

(revisited)

Traversal utility functions

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

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!

```
int lengthOf() {...}
```

Attendance ticket:

https://tinyurl.com/lengthOfList

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

`lengthOf()`

Let's code it!

Linked Lists and Recursion

Rethinking Linked Lists

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

Rethinking Linked Lists

- On Monday, 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 Monday, 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.

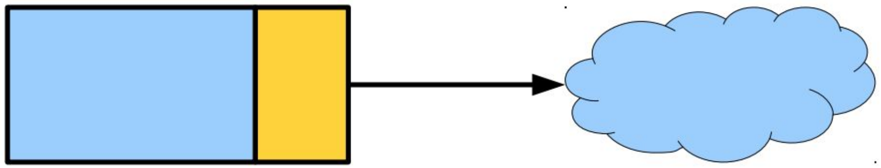
A Linked List is Either…

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

- 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.

- 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 while traversing.

- When traversing but not editing a list, we often pass a pointer parameter by value into a utility function.

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

Linked List Insertion

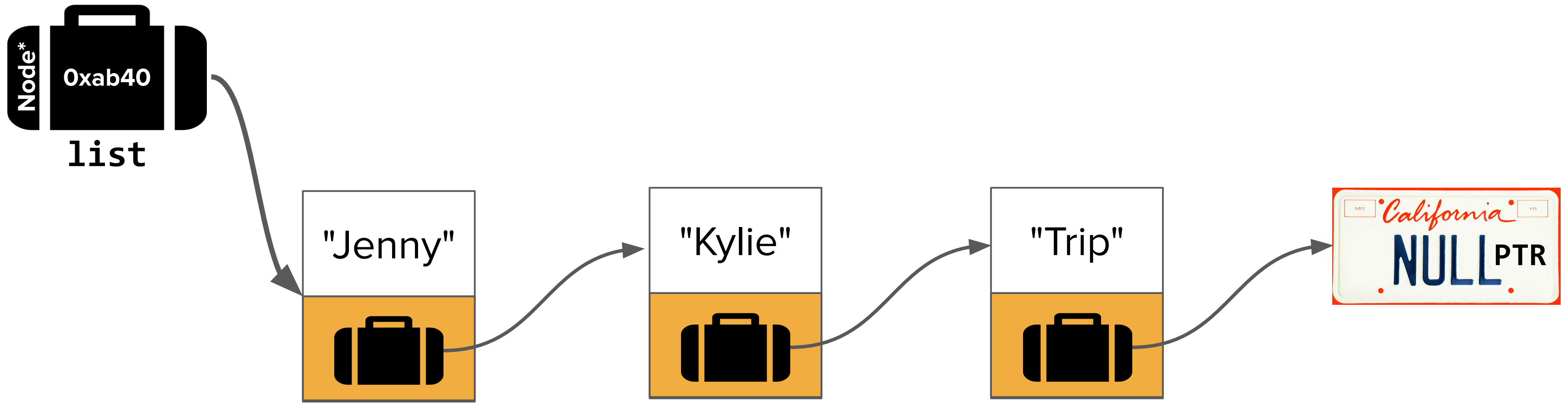
Insertion at the front (prepend)

Prepending an Element

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

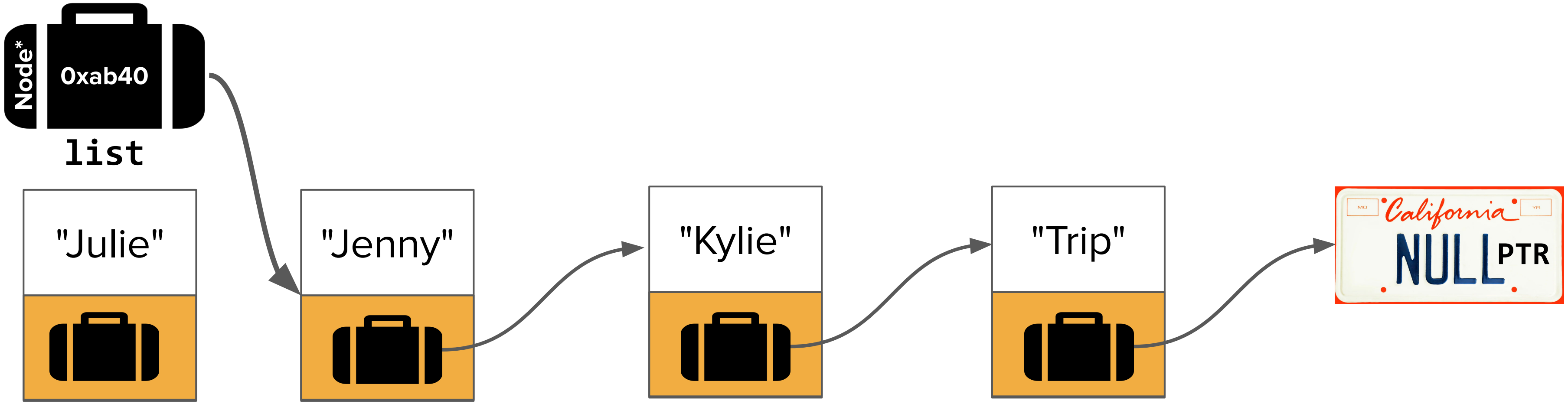
Prepending an Element

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



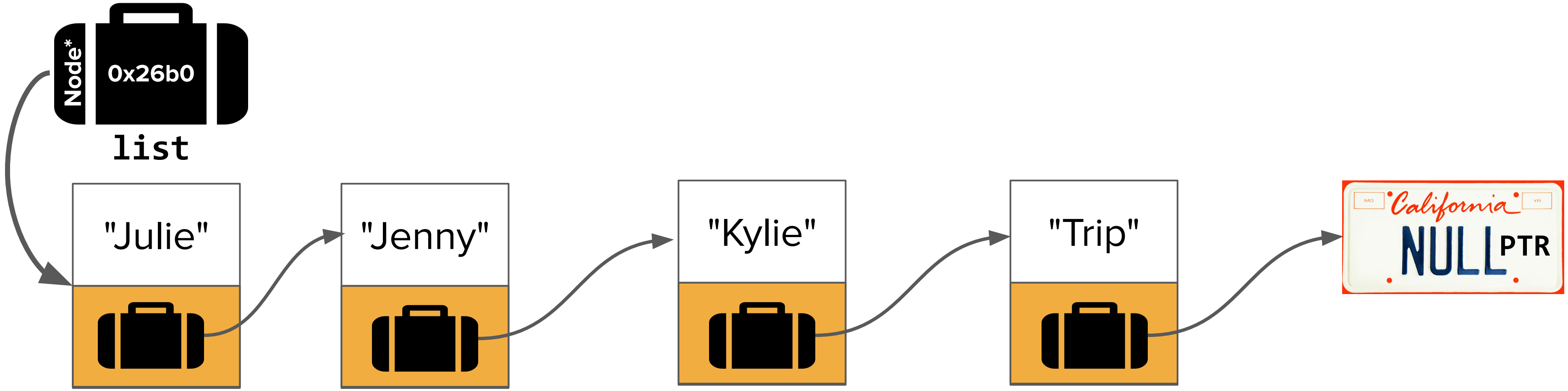
Prepending an Element

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



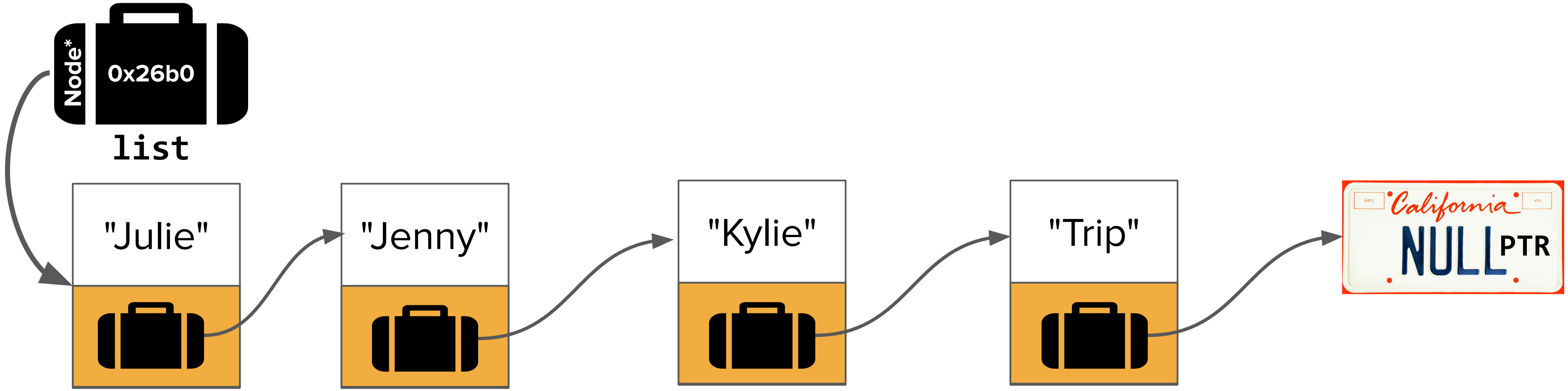
Prepending an Element

- 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 Monday, but now we're writing a standalone function to do this on an arbitrary list. Let's code it!



`prependTo()`

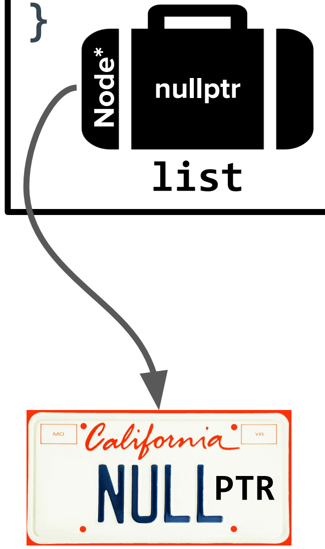
Let's code it!

What went wrong?

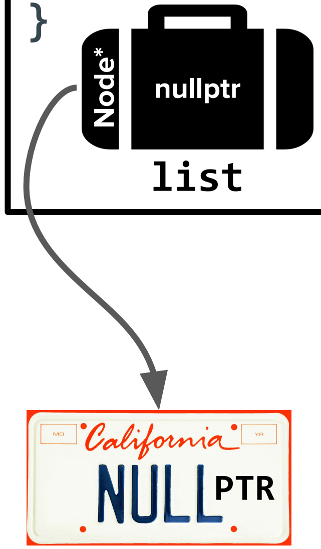
```
int main() {
    Node* list = nullptr;
    prependTo(list, "Trip");
    prependTo(list, "Kylie");
    prependTo(list, "Jenny");
    return 0;
}
```

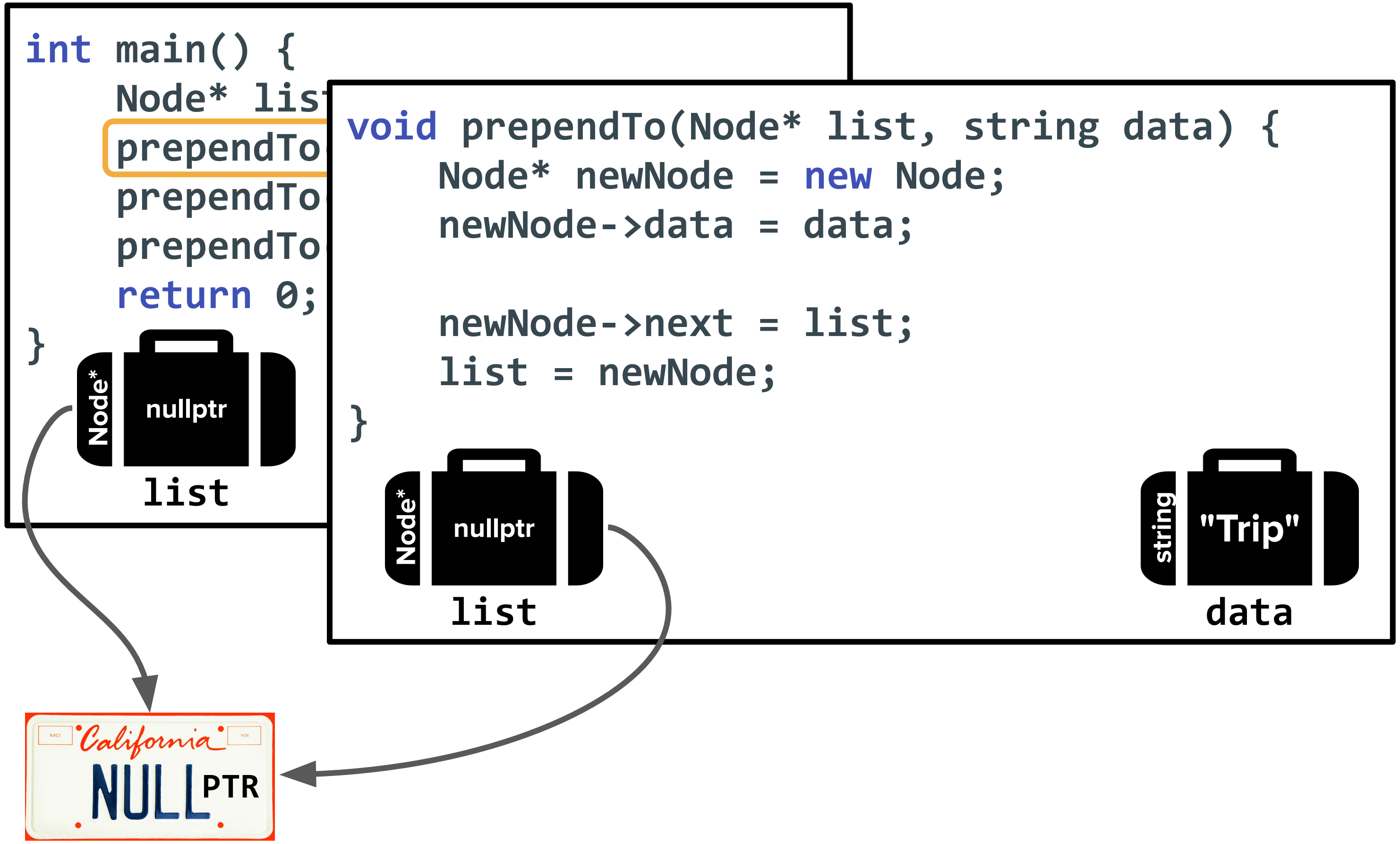
```
int main() {
    Node* list = nullptr;
    prependTo(list, "Trip");
    prependTo(list, "Kylie");
    prependTo(list, "Jenny");
    return 0;
}
```

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



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



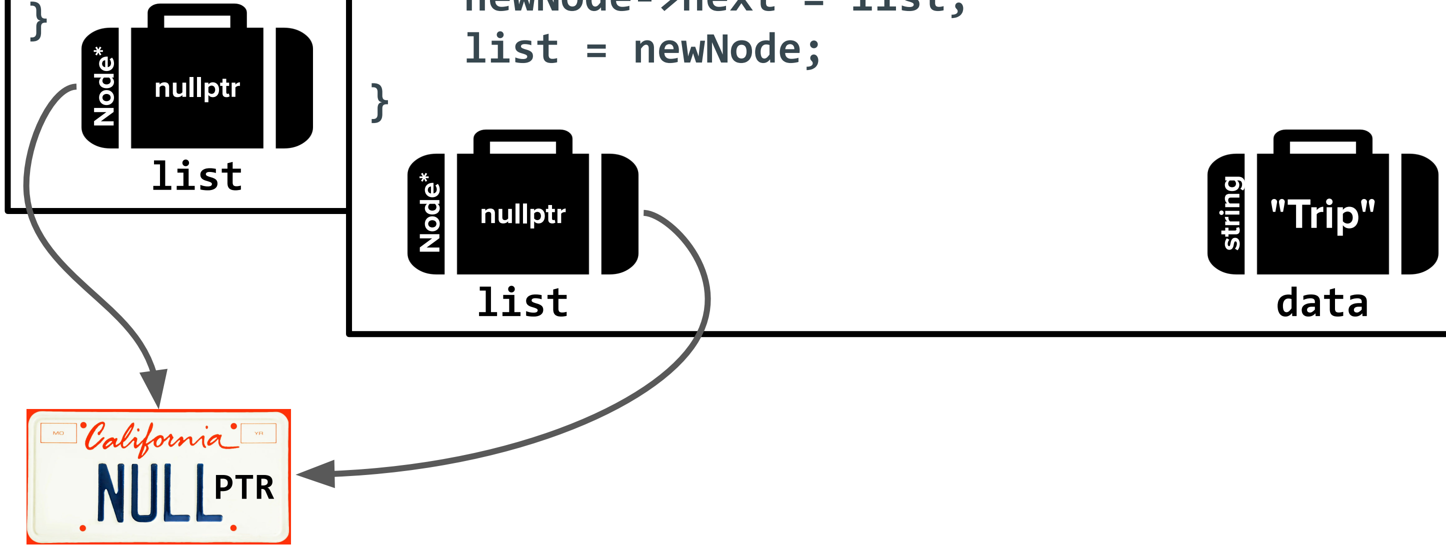


```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

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

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

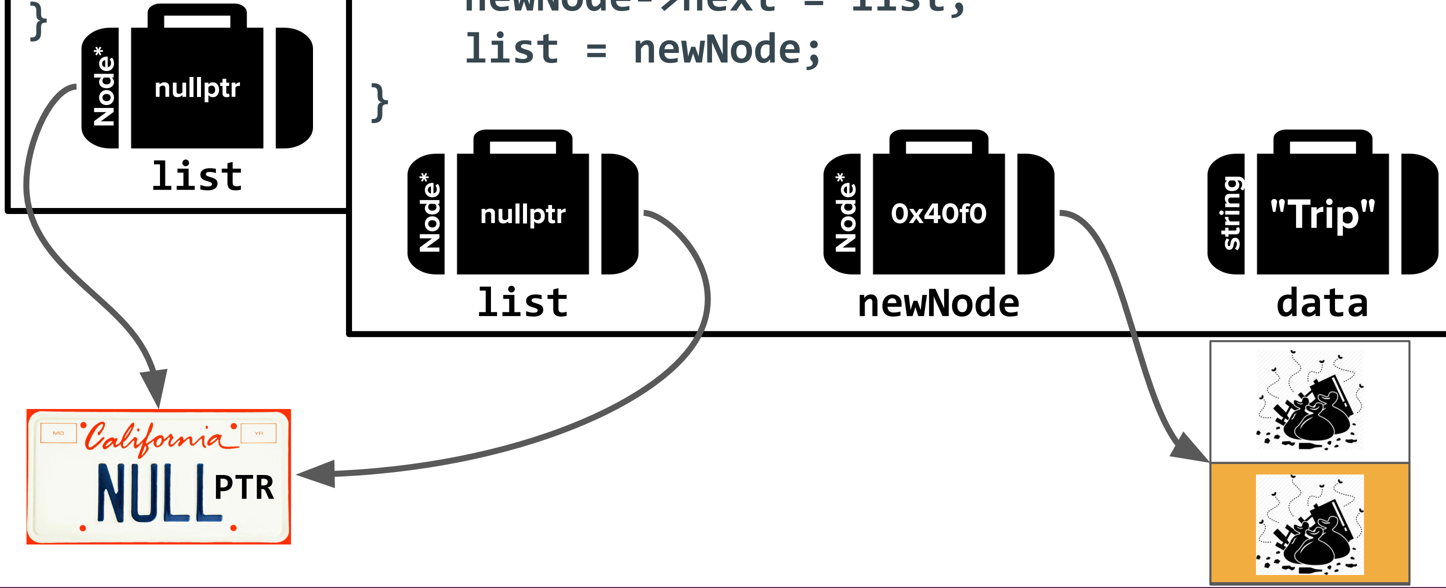
```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

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

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



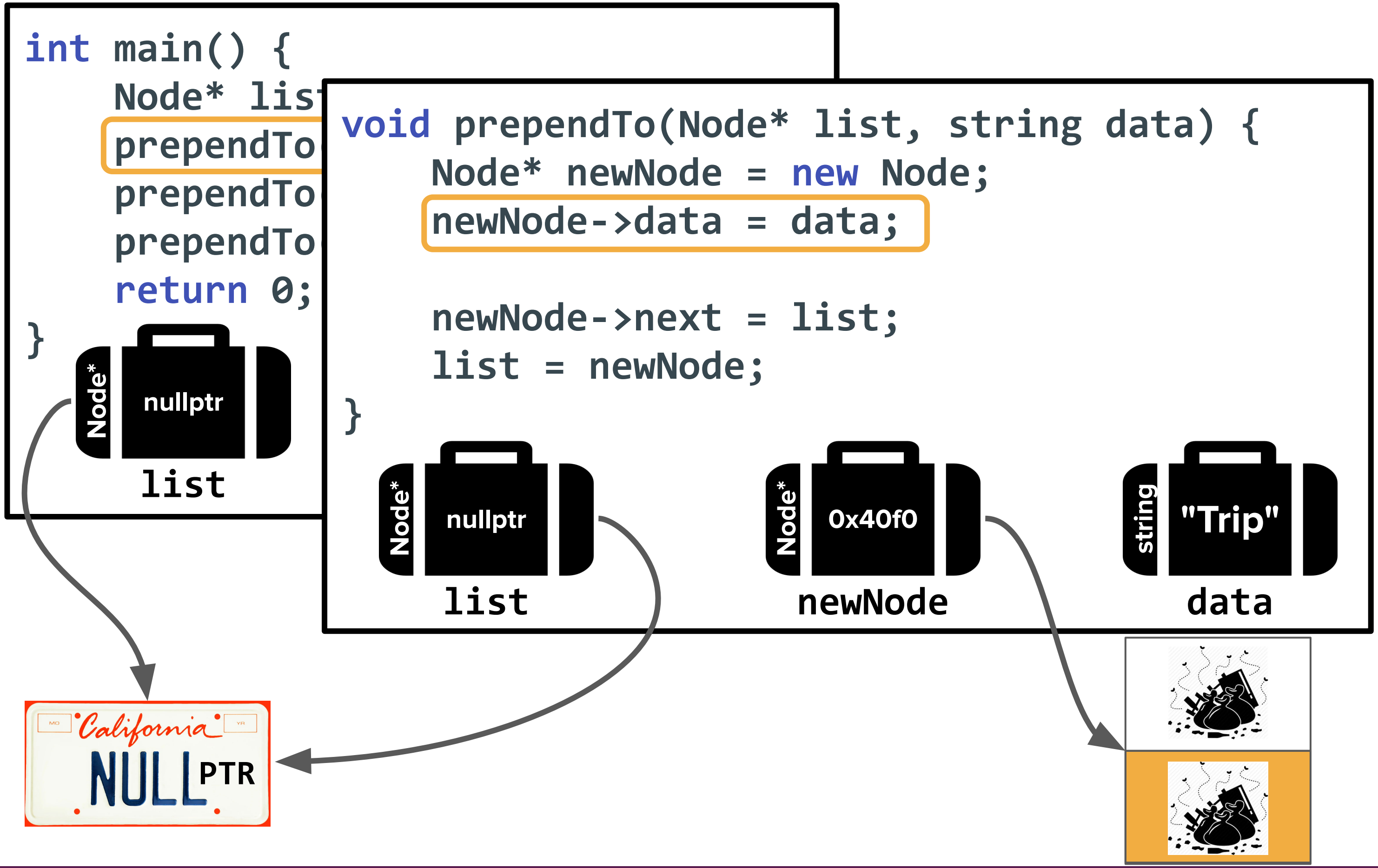
```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

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

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



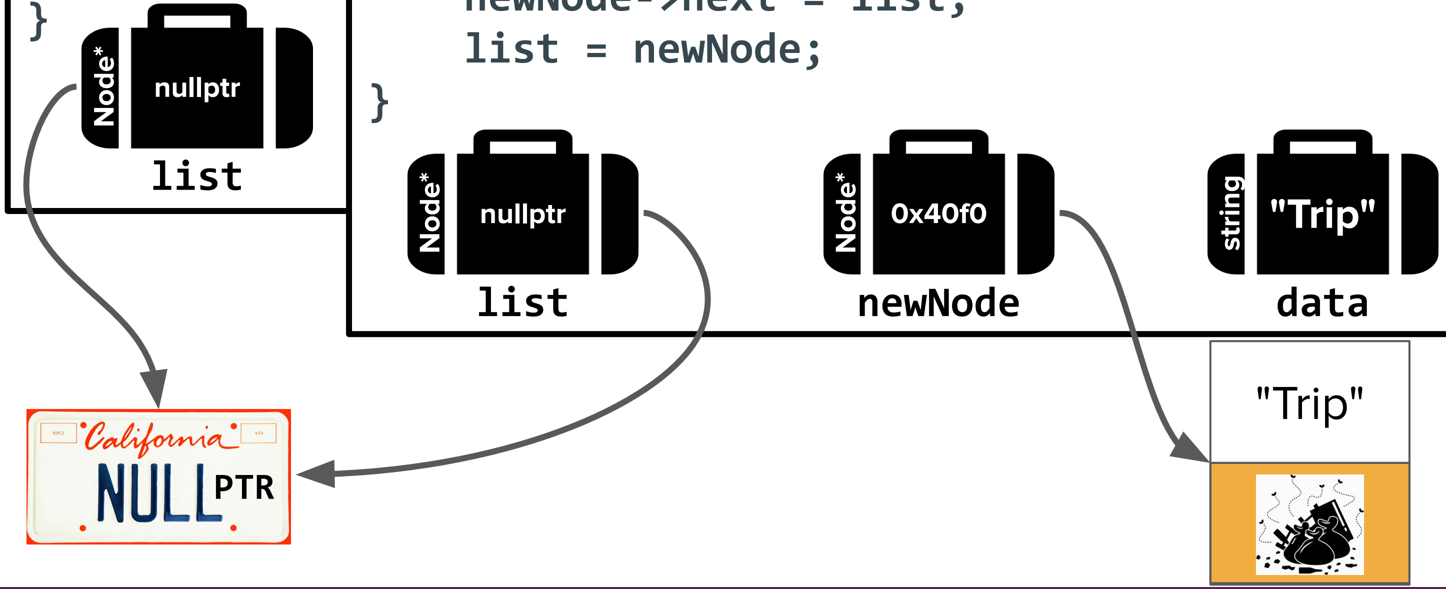
```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

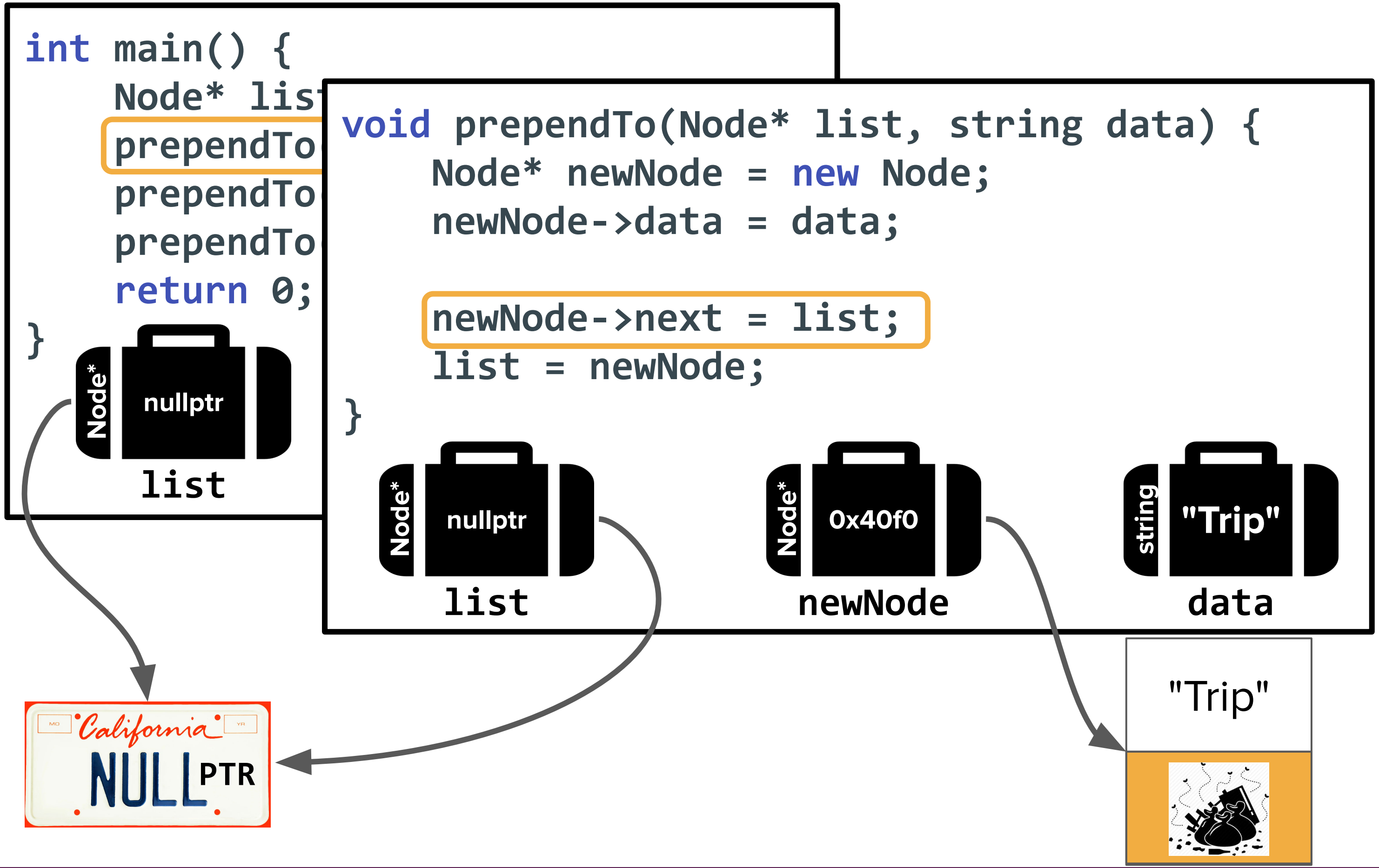
```
void prependTo(Node* list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

    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;
}
```



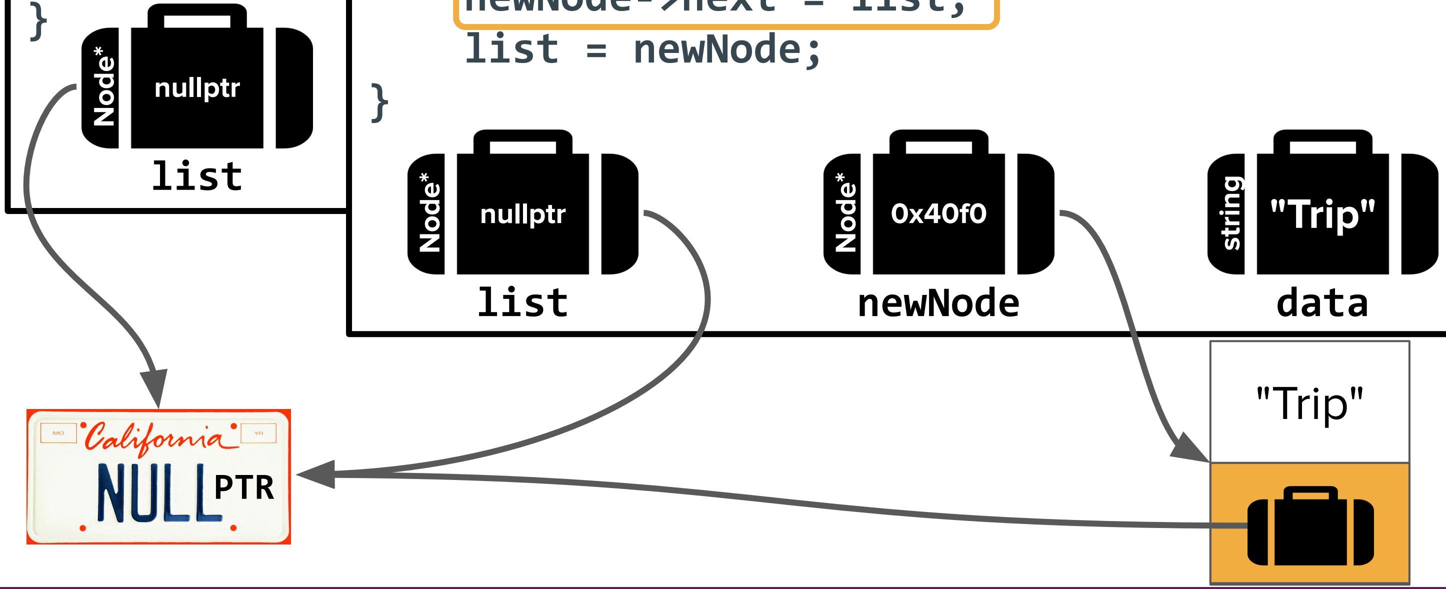


```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

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

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

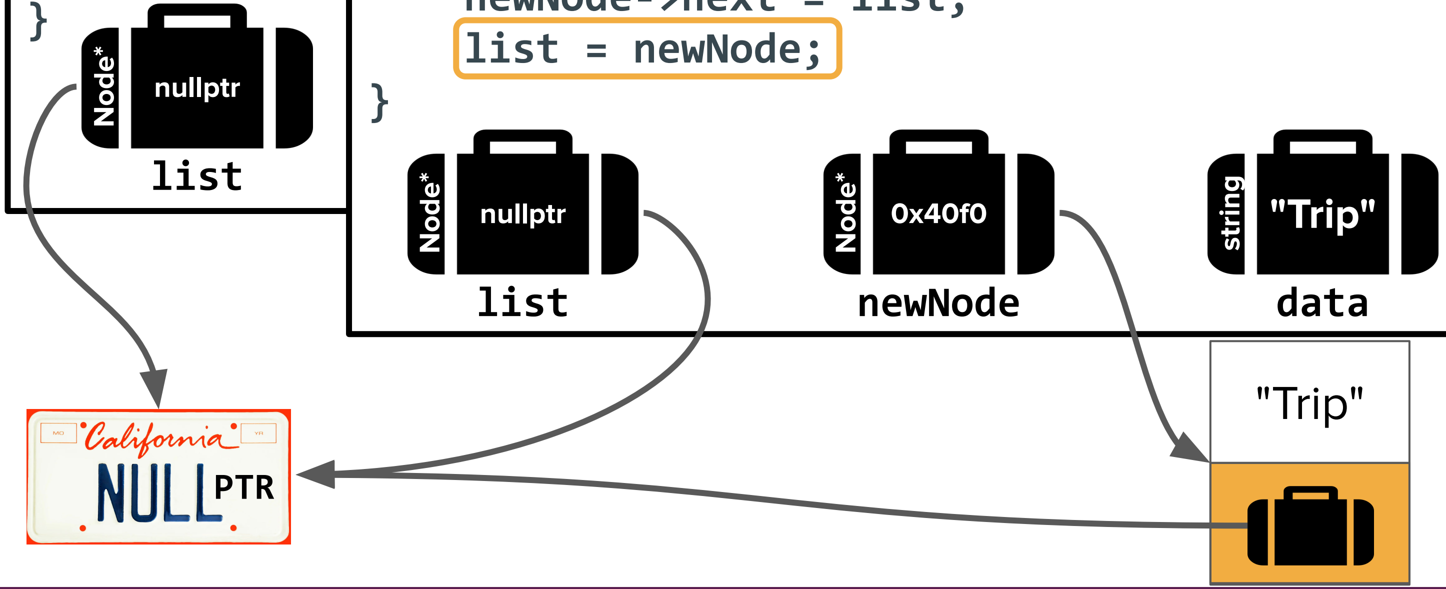
```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

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

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



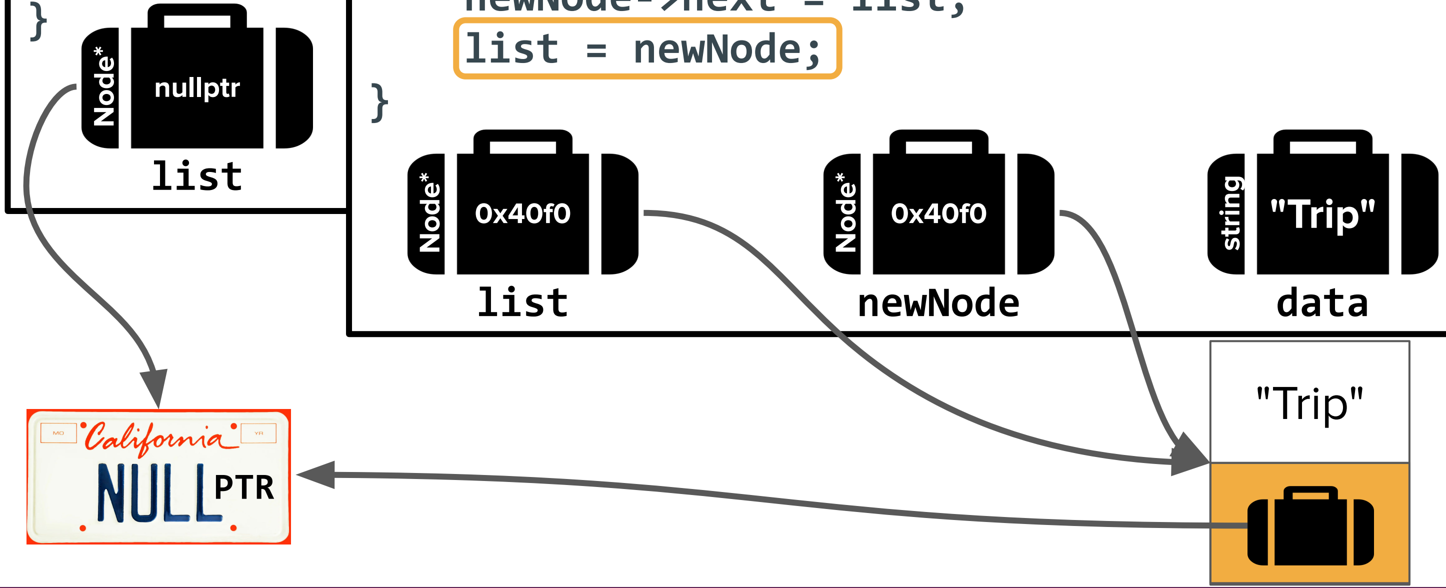
```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

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

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

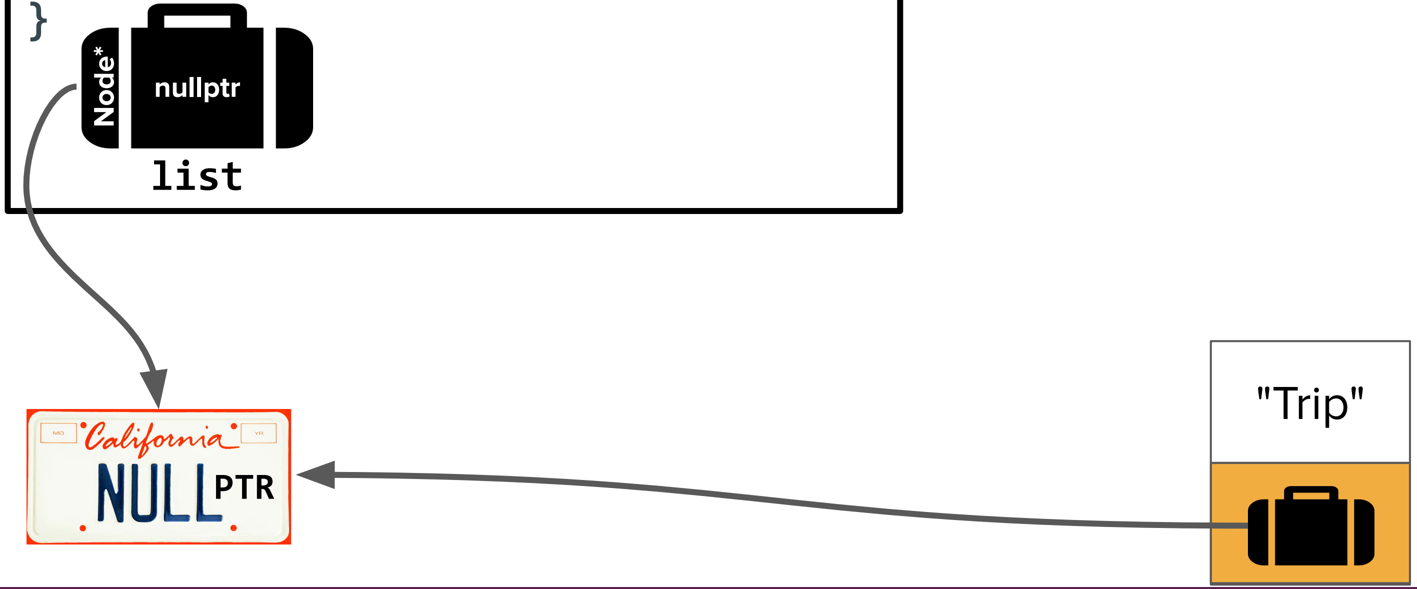


```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

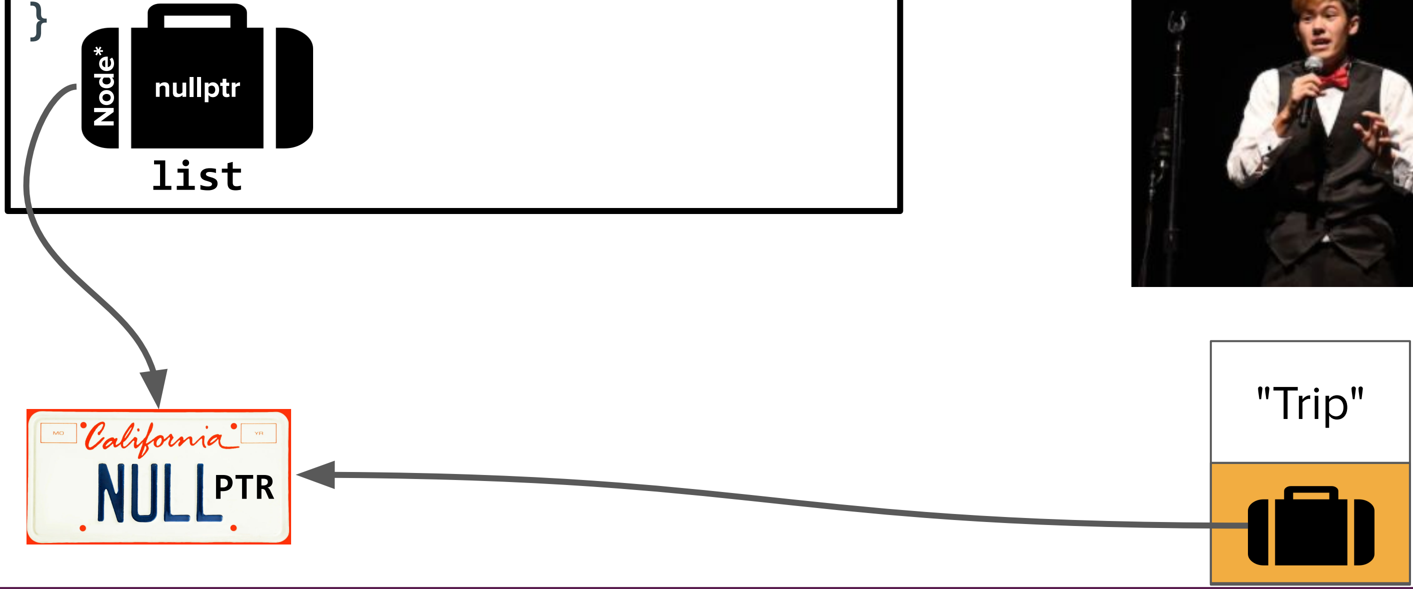
```
void prependTo(Node* list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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

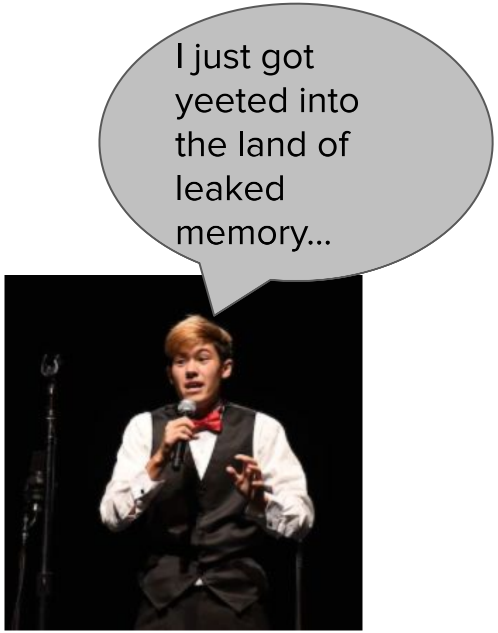


```
int main() {
    Node* list = nullptr;
    prependTo(list, "Trip");
    prependTo(list, "Kylie");
    prependTo(list, "Jenny");
    return 0;
}
```



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



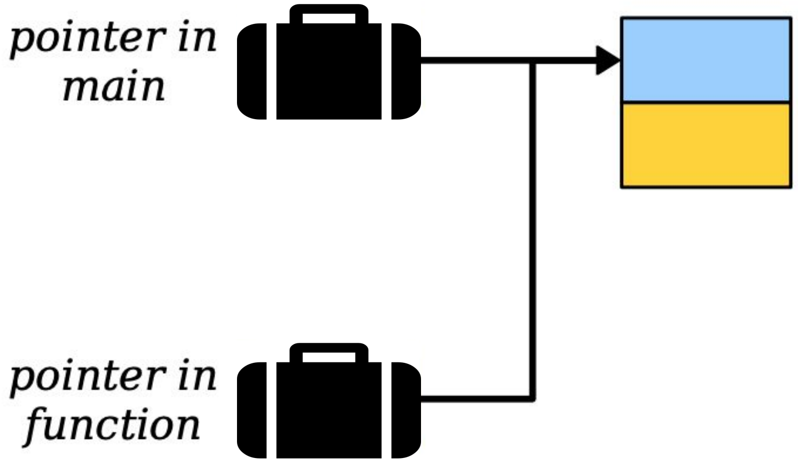


Pointers by Value

- 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:

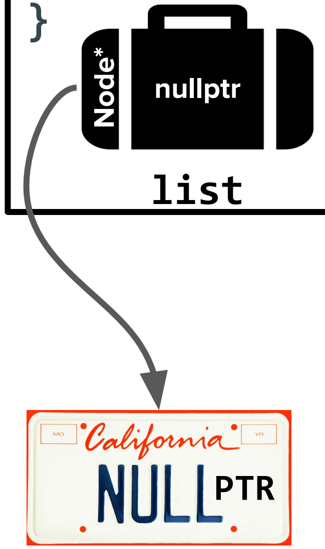
```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

    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!

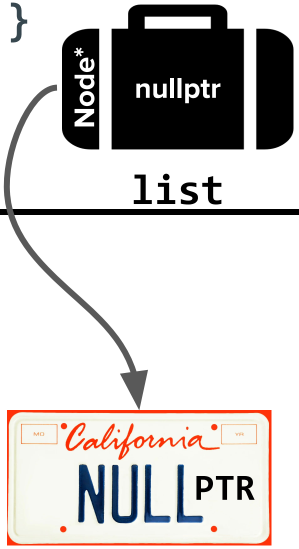
```cpp
int main() {
    Node* list = nullptr;
    prependTo(list, "Trip");
    prependTo(list, "Kylie");
    prependTo(list, "Jenny");
    return 0;
}
```

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

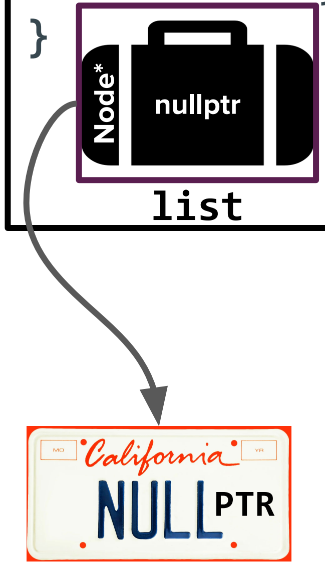
```
int main() {
    Node* list = nullptr;
    prependTo(list, "Trip");
    prependTo(list, "Kylie");
    prependTo(list, "Jenny");
    return 0;
}
```



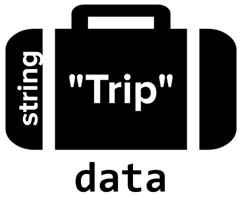
```
int main() {
    Node* list = nullptr;
    prependTo(list, "Trip");
    prependTo(list, "Kylie");
    prependTo(list, "Jenny");
    return 0;
}
```



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

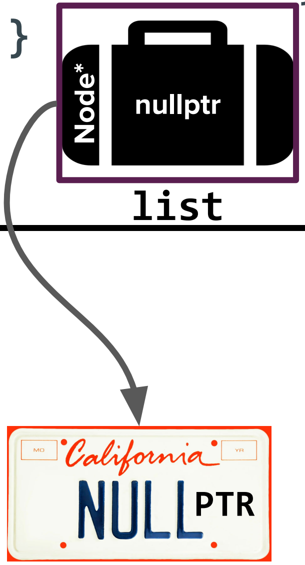


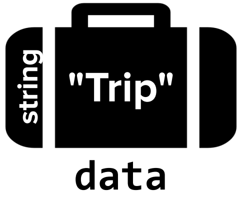
```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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



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

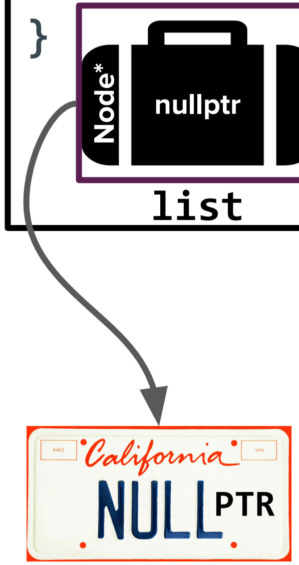
```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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

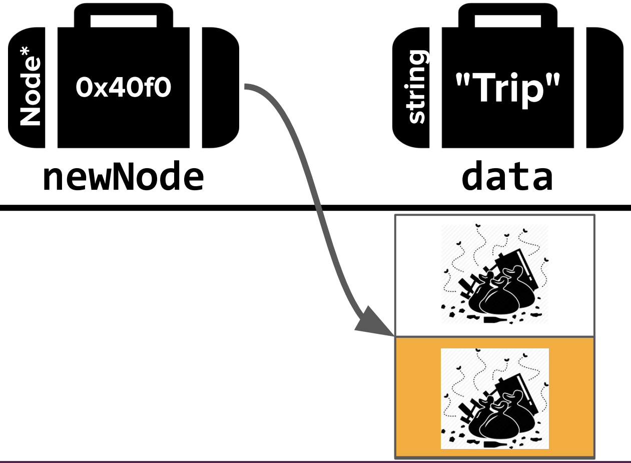




```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

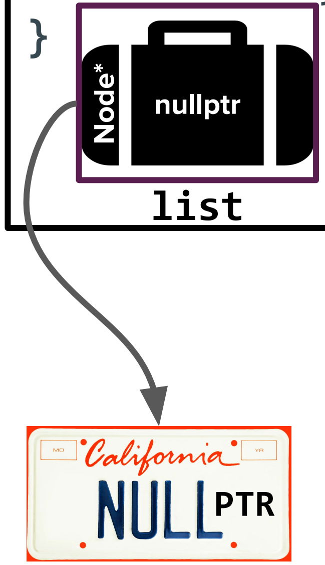
```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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

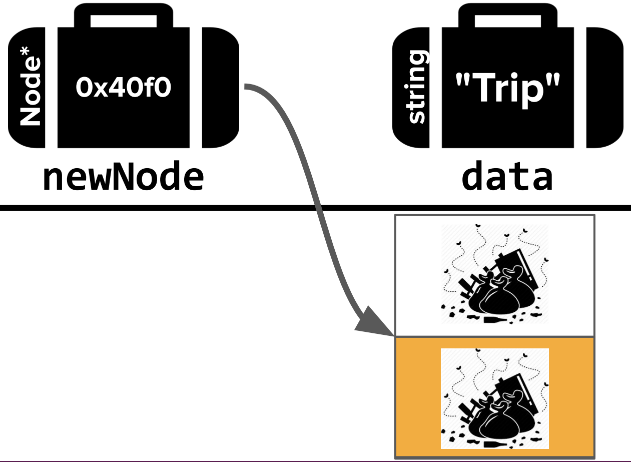




```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

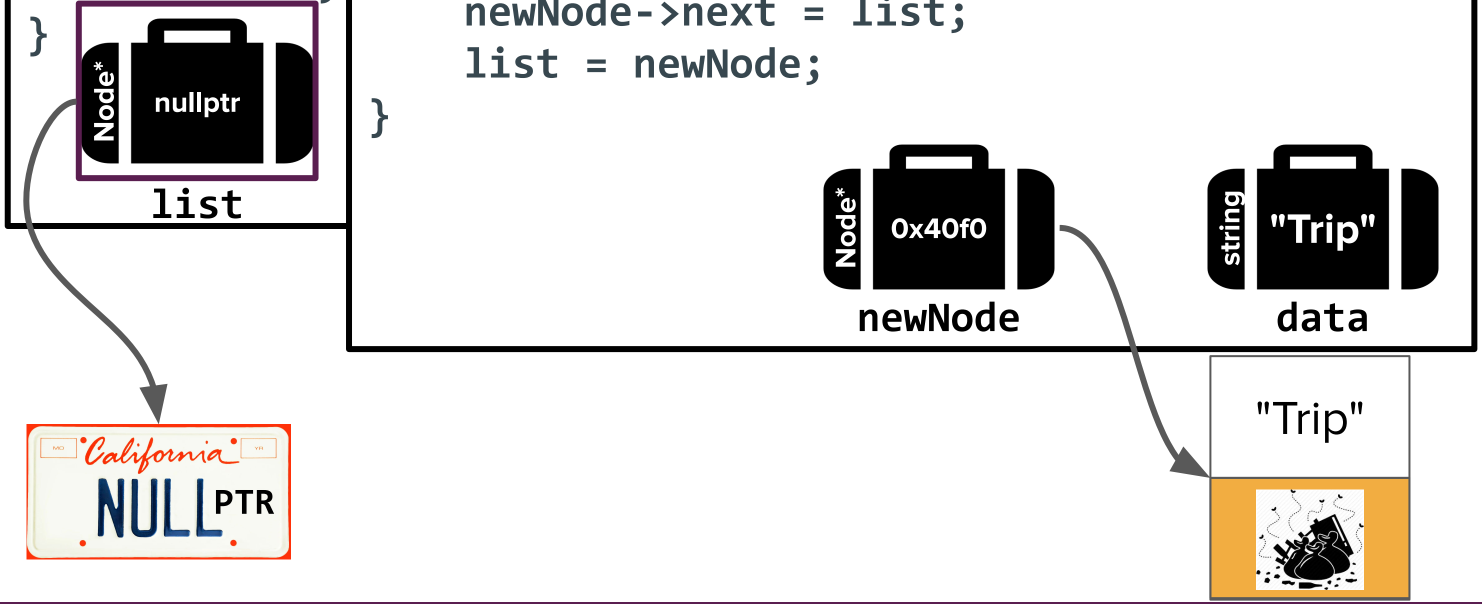
```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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





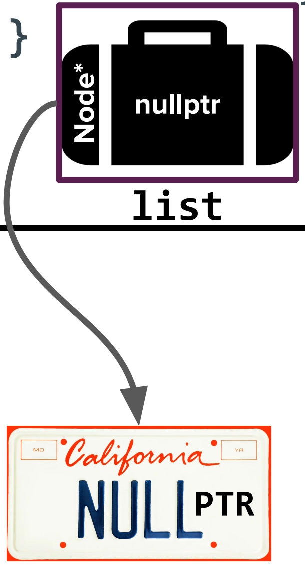
```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

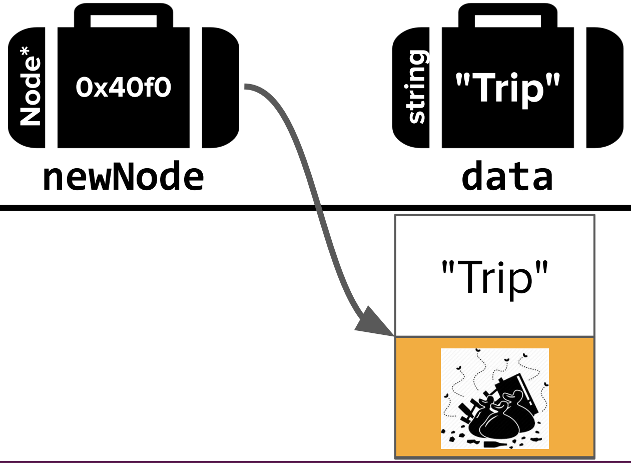
```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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



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

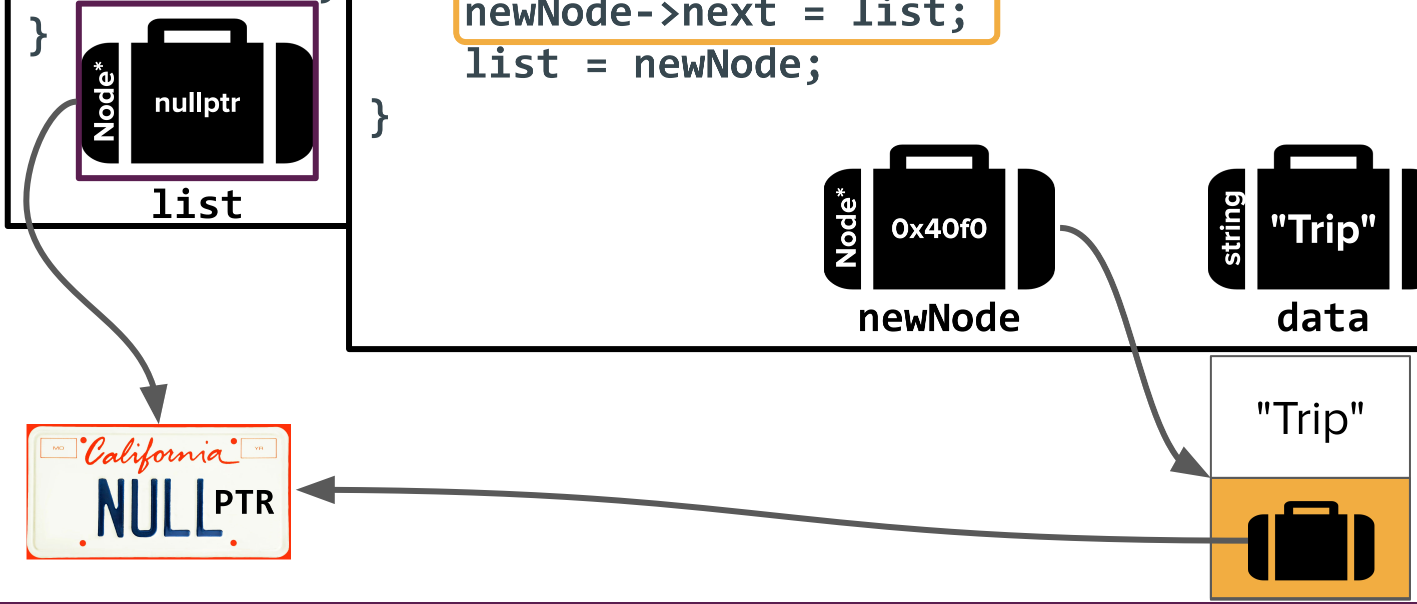
```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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

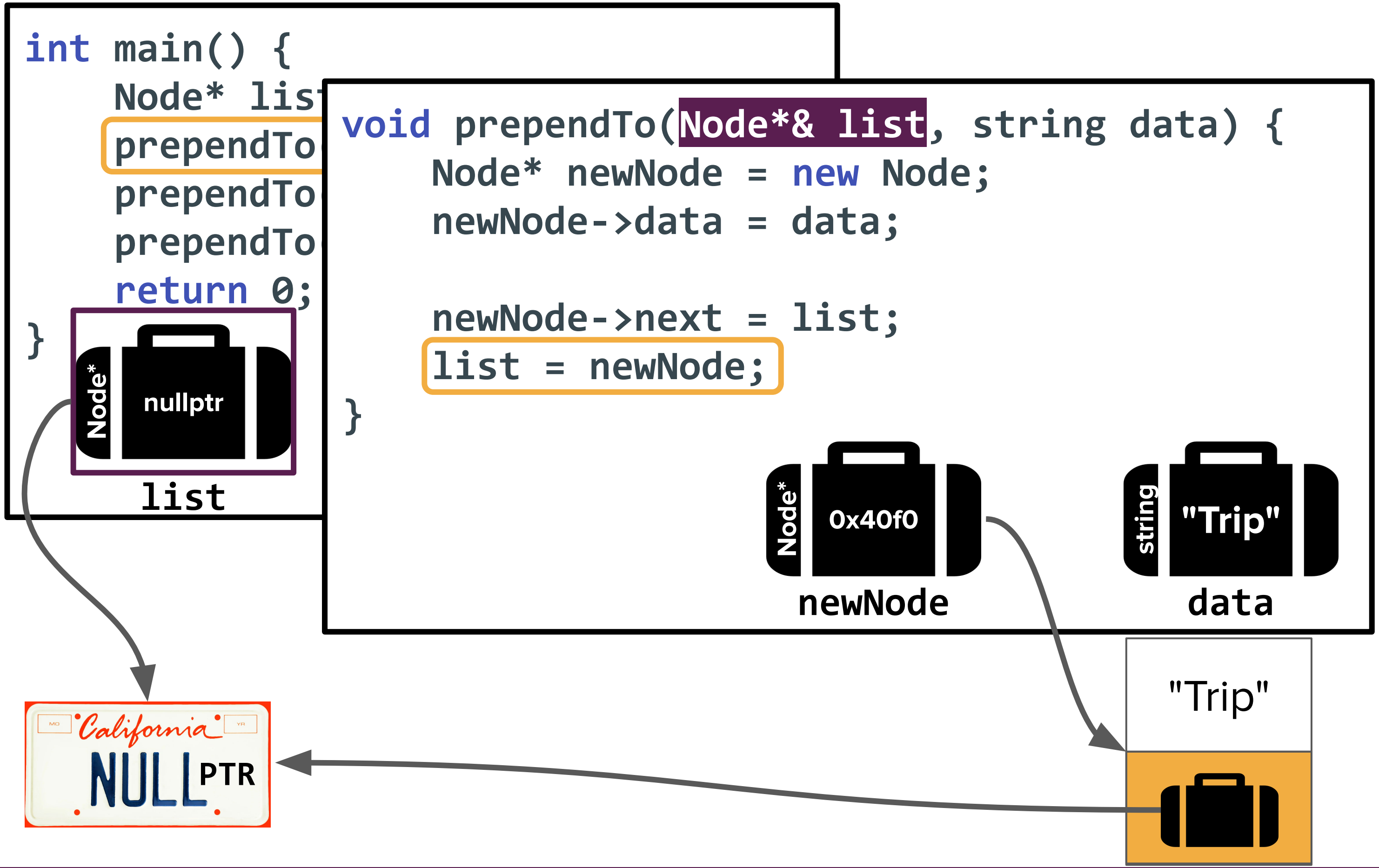




```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

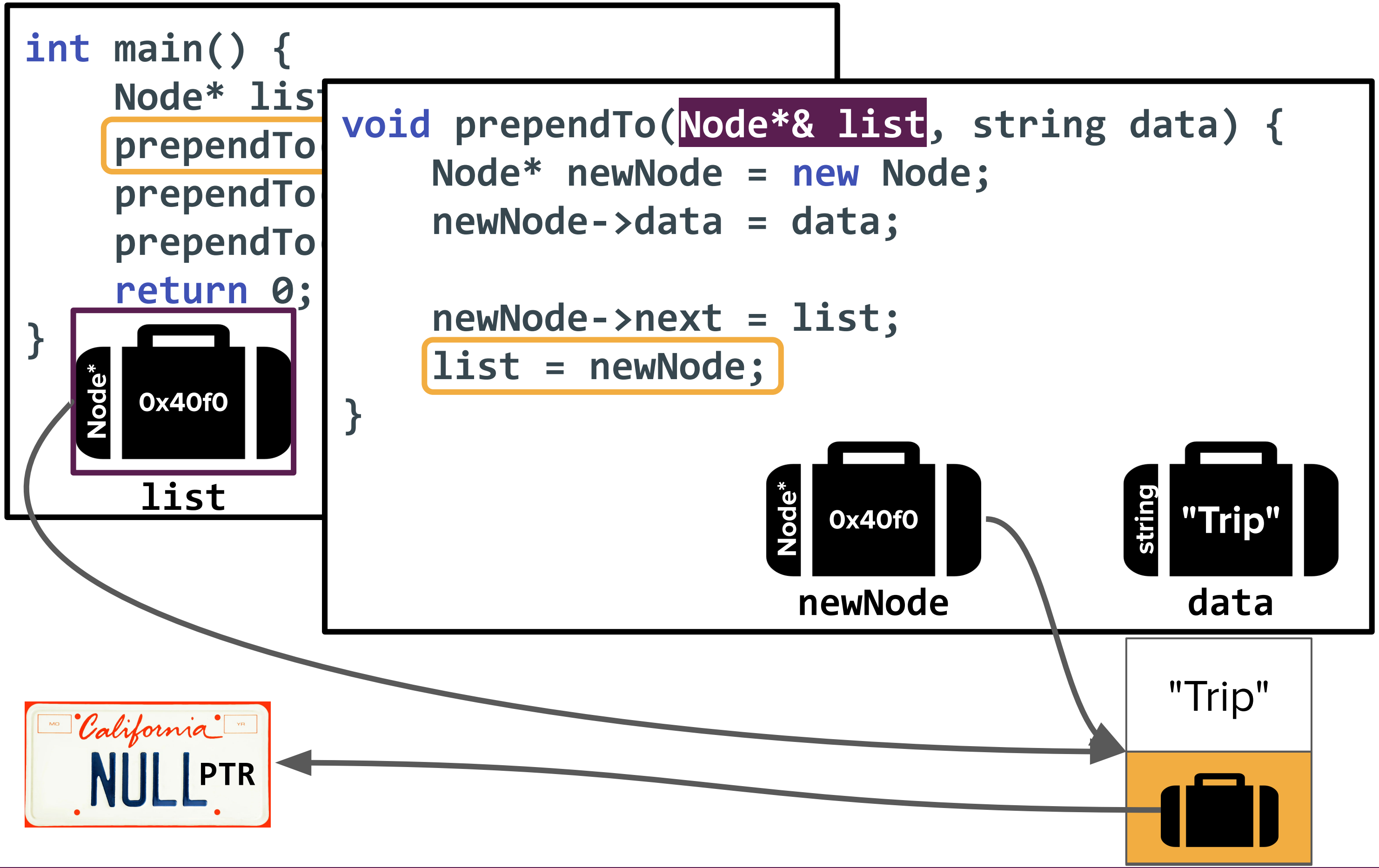
```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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





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

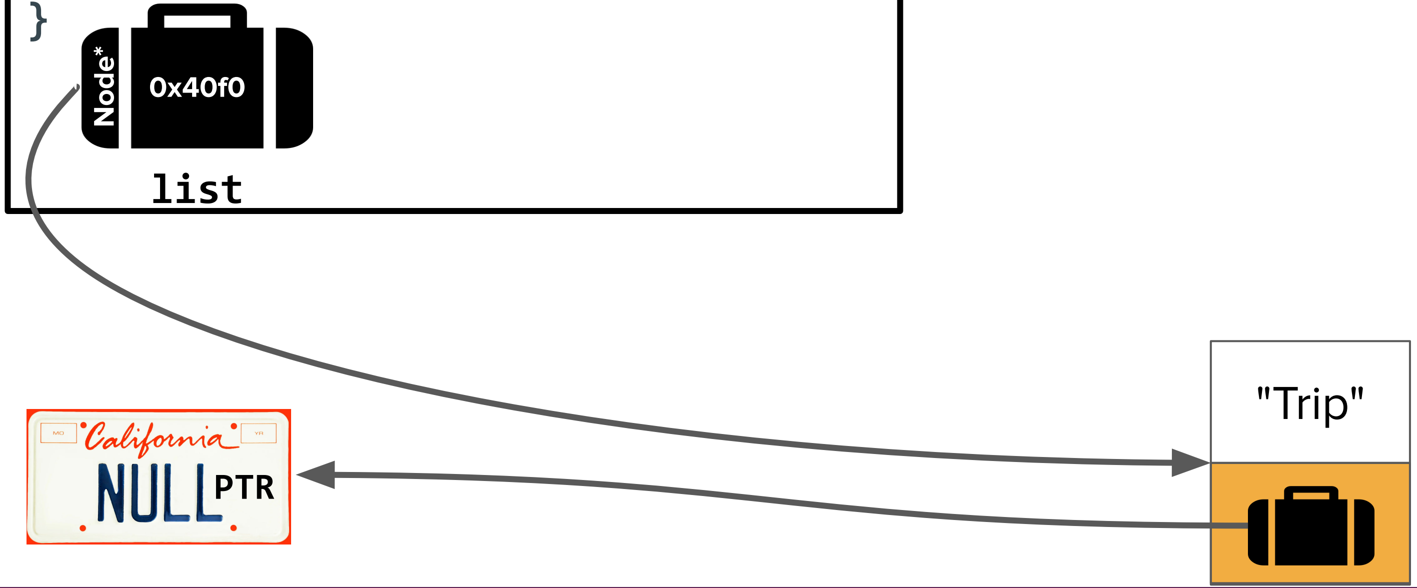
```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

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



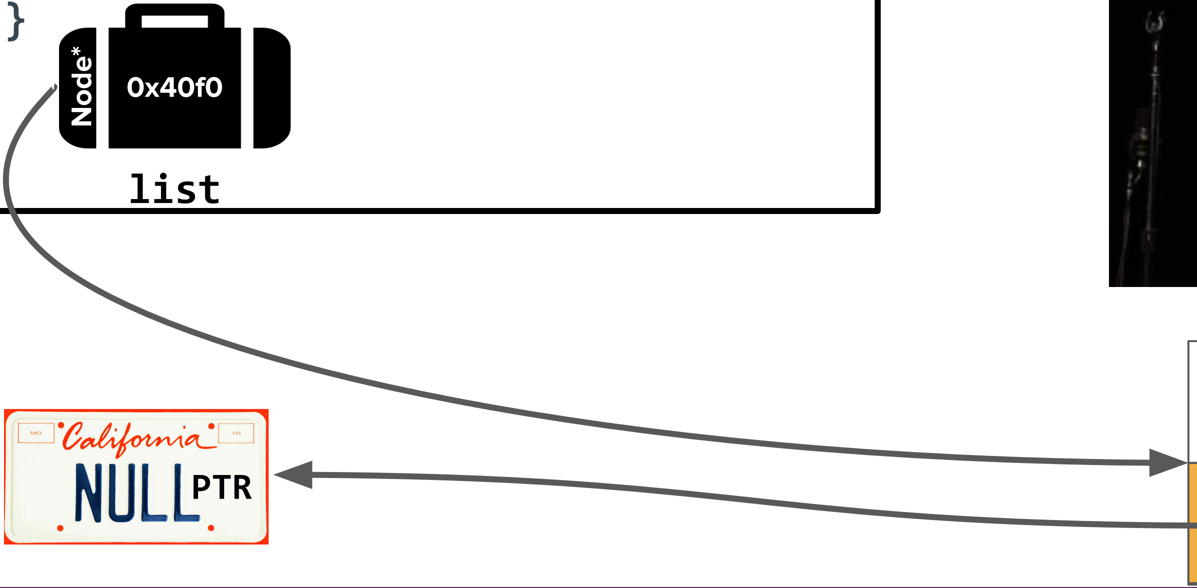
```
int main() {
    Node* lis
    prependTo
    prependTo
    prependTo
    return 0;
}
```

```
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, "Jenny");
    return 0;
}
```



```
int main() {
    Node* list = nullptr;
    prependTo(list, "Trip");
    prependTo(list, "Kylie");
    prependTo(list, "Jenny");
    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.
    - How do you change the contents? Dereferencing!
    - But you don’t have access to the original pointer if it’s passed by value.

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.
  - The utility function will now edit the original pointer, just like when we passed ADTs by reference!

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!

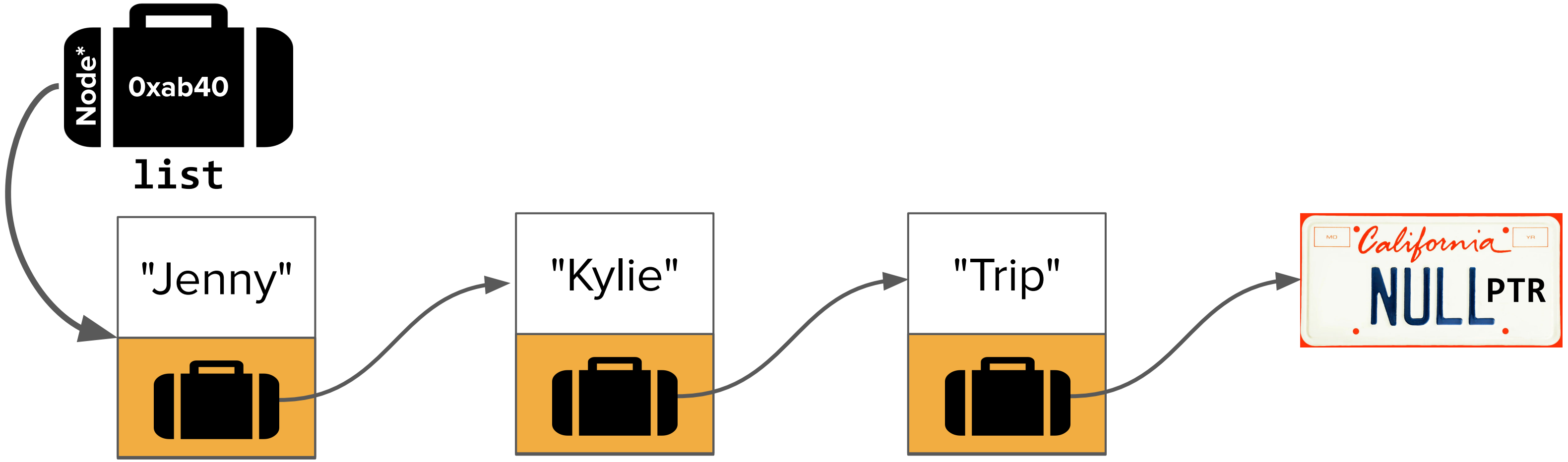
Insertion at the end (append)

Appending an Element

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

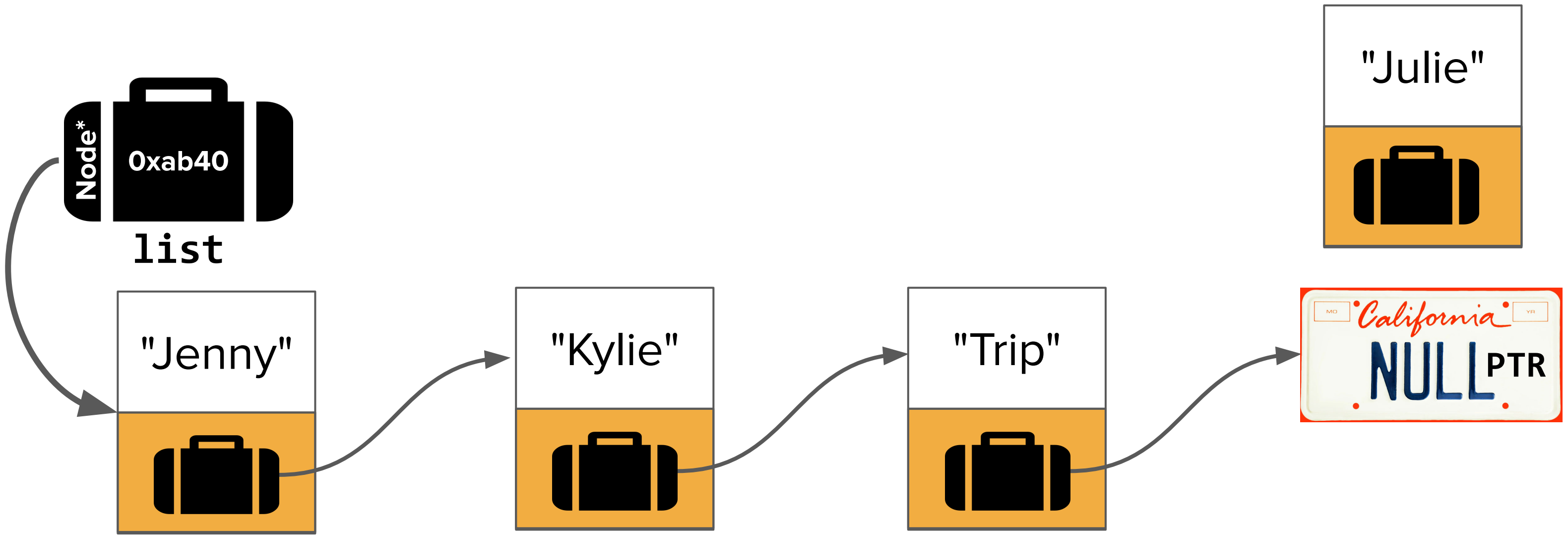
Appending an Element

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



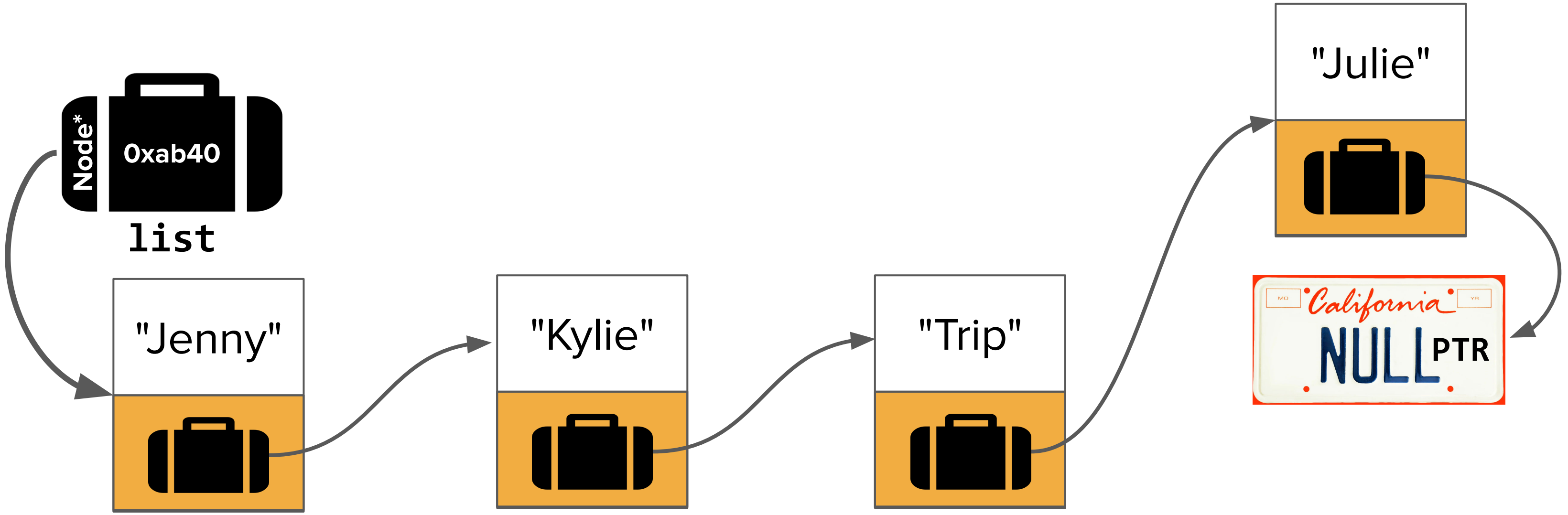
Appending an Element

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



Appending an Element

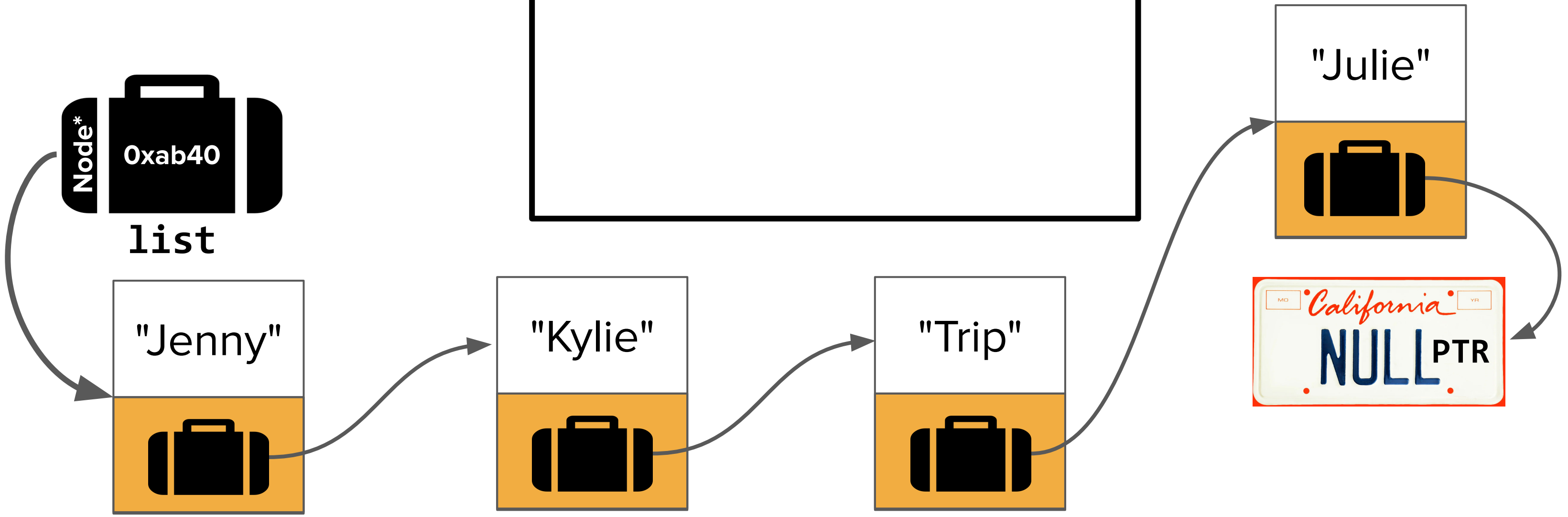
- Suppose we wanted to write a function to add an element to the end of a linked list.



Appending an Element

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

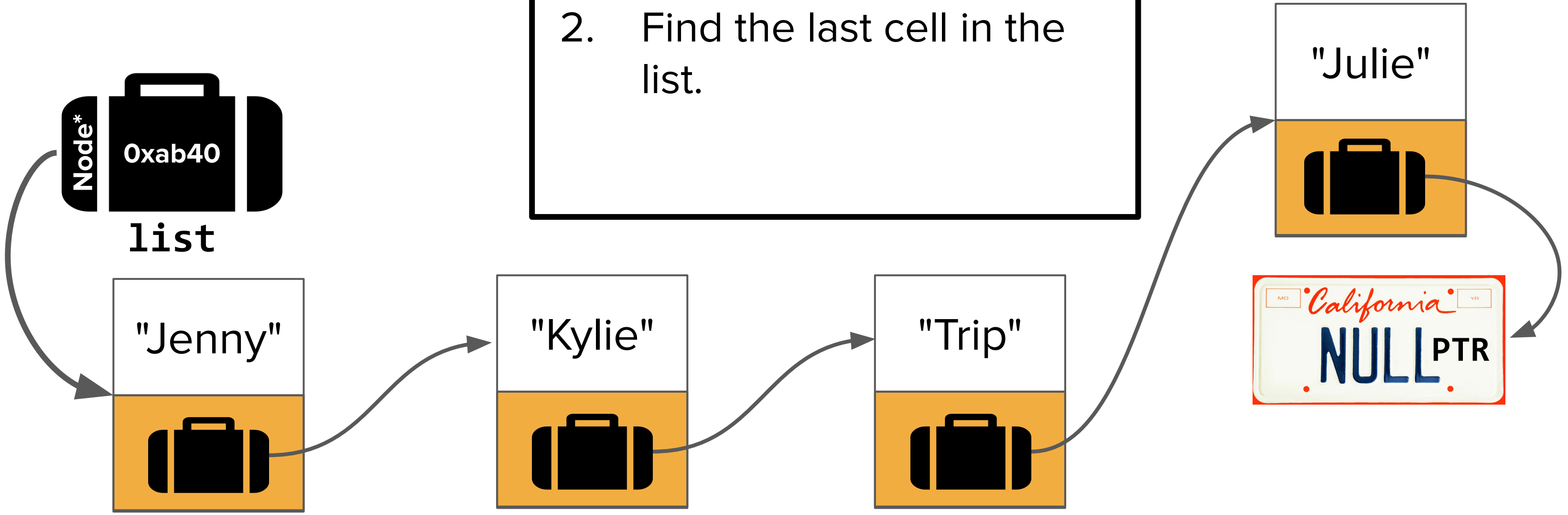
1. Create a cell whose **`next`** field is nullptr.



Appending an Element

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

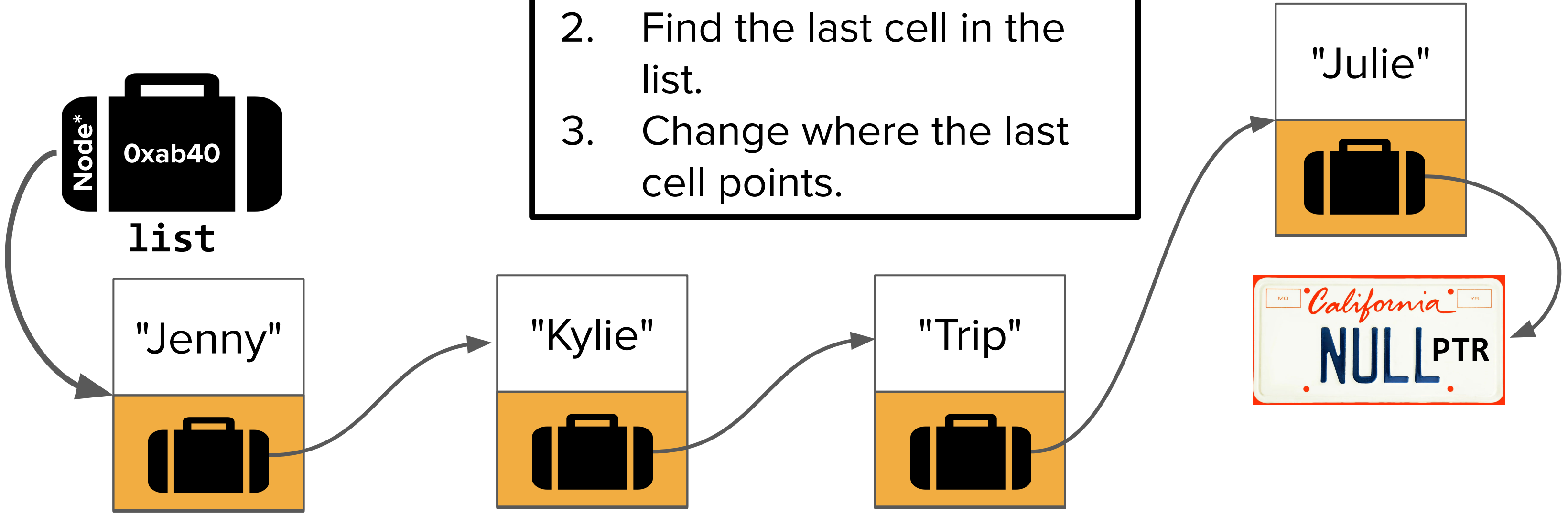
1. Create a cell whose **next** field is nullptr.
2. Find the last cell in the list.



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 after the break!

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)

Announcements

Announcements

- Assignment 4 is due today. Assignment 3 revisions are due Friday, July 29.

- Assignment 5 will be released by the end of the day.
  - YEAH Hours will be tomorrow on Wednesday, July 27 at 5pm in Hewlett 103.

- Lecture this Thursday will be open project work time!
  - Jenny and I will be having open OH in NVIDIA if you have questions.
  - We'll also set up areas in the lecture hall where you can discuss projects by topic so you can get feedback and ideas from your classmates.
  - Attendance is optional.

- The deadline to change your grading basis is this Friday, July 29 at 5pm.

Linked List Insertion

(continued)

Two ways to add (so far)

- Insertion at the front: **`prependTo()`**
    - Prepending is faster but results in a reversed order of items (things added earlier are at the back of the list)

- Insertion at the back: **`appendTo()`**
    - Appending requires traversing all items but maintains order (things added earlier are at the front of the list)

```
void nameOfAddFunction(Node*& list, string data) {
    ...
}
```

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?

A more efficient append

A more efficient **`appendTo()`**

- Earlier, we saw an O(n) **`appendTo()`** that added to the back of a linked list. We can do better!
- What if we know we're going to add many things in some maintained order?
- Specifically, we'll use the example of adding items from a vector into linked list.

Attempt #1

```
Node* createListWithAppend(Vector<string> values) {
    if (values.isEmpty()) {
        return nullptr;
    }
    Node* head = new Node(values[0], nullptr);

    for (int i = 1; i < values.size(); i++) {
        appendTo(head, values[i]);
    }
    return head;
}
```

Attempt #1: **What's the runtime? (poll)**

```
Node* createListWithAppend(Vector<string> values) {
    if (values.isEmpty()) {
        return nullptr;
    }
    Node* head = new Node(values[0], nullptr);

    for (int i = 1; i < values.size(); i++) {
        appendTo(head, values[i]);
    }
    return head;
}
```

A. $O(N)$
B. $O(N^2)$
C. $O(N^3)$
D. $O(\log N)$

https://pollev.com/cs106bpolls

What's the runtime?

- O(N)
- O(N^2)
- O(N^3)
- O(logN)

Attempt #1: **What's the runtime? (poll)**

```
Node* createListWithAppend(Vector<string> values) {
    if (values.isEmpty()) {
        return nullptr;
    }
    Node* head = new Node(values[0], nullptr);

    for (int i = 1; i < values.size(); i++) {
        appendTo(head, values[i]);
    }
    return head;
}
```

A. O(N)
[✓ B. O(N^2)]
C. O(N^3)
D. O(log N)

Attempt #2:

`createListWithTailPtr()`

Let's code it!

How does it work?

```
int main() {
    Vector<string> values = {"Jenny", "Kylie", "Trip"};
    Node* list = createListWithTailPtr(values);

    /* Do other list-y things here, like printing/freeing the list. */
    return 0;
}
```

```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

**{"Jenny",
"Kylie",
"Trip"}**

```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

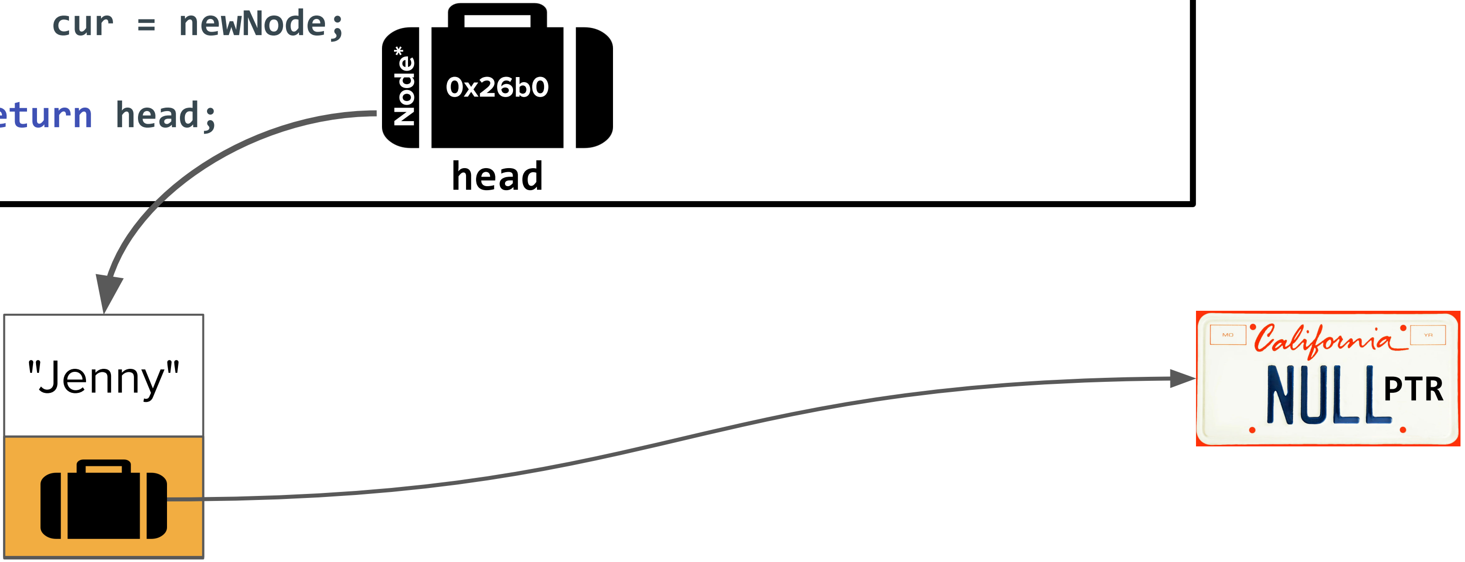
{"Jenny",
"Kylie",
"Trip"}

```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

{"Jenny",
"Kylie",
"Trip"}

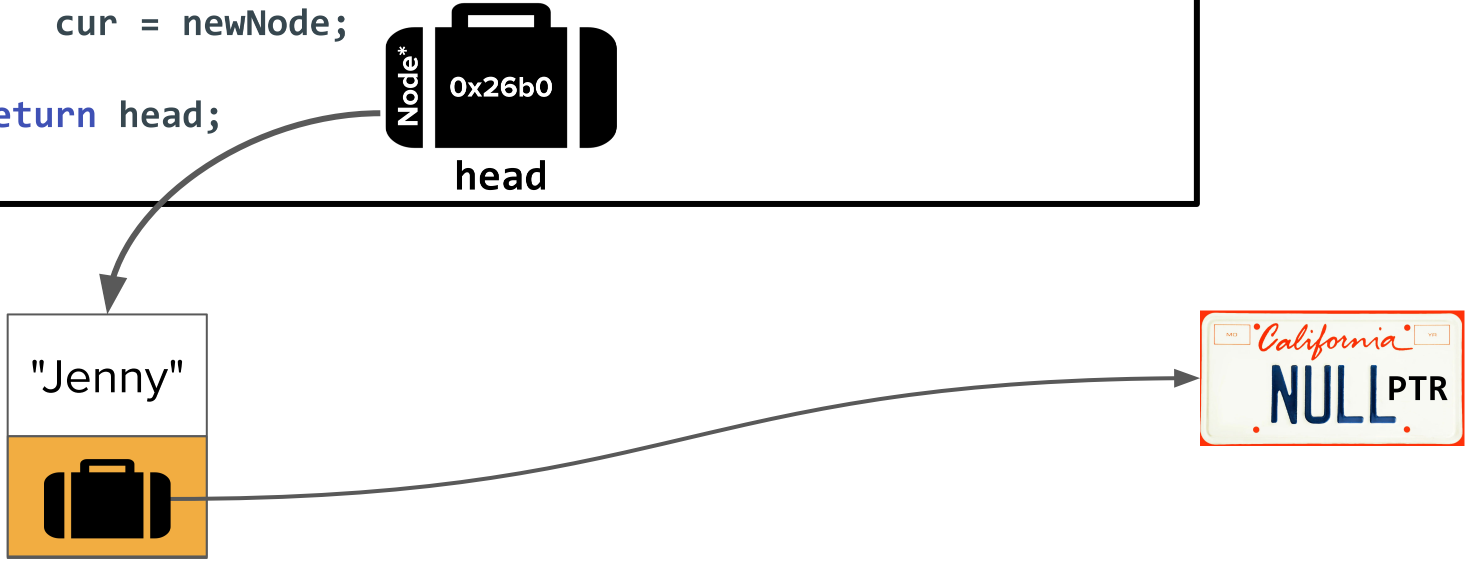
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

{"Jenny",
"Kylie",
"Trip"}

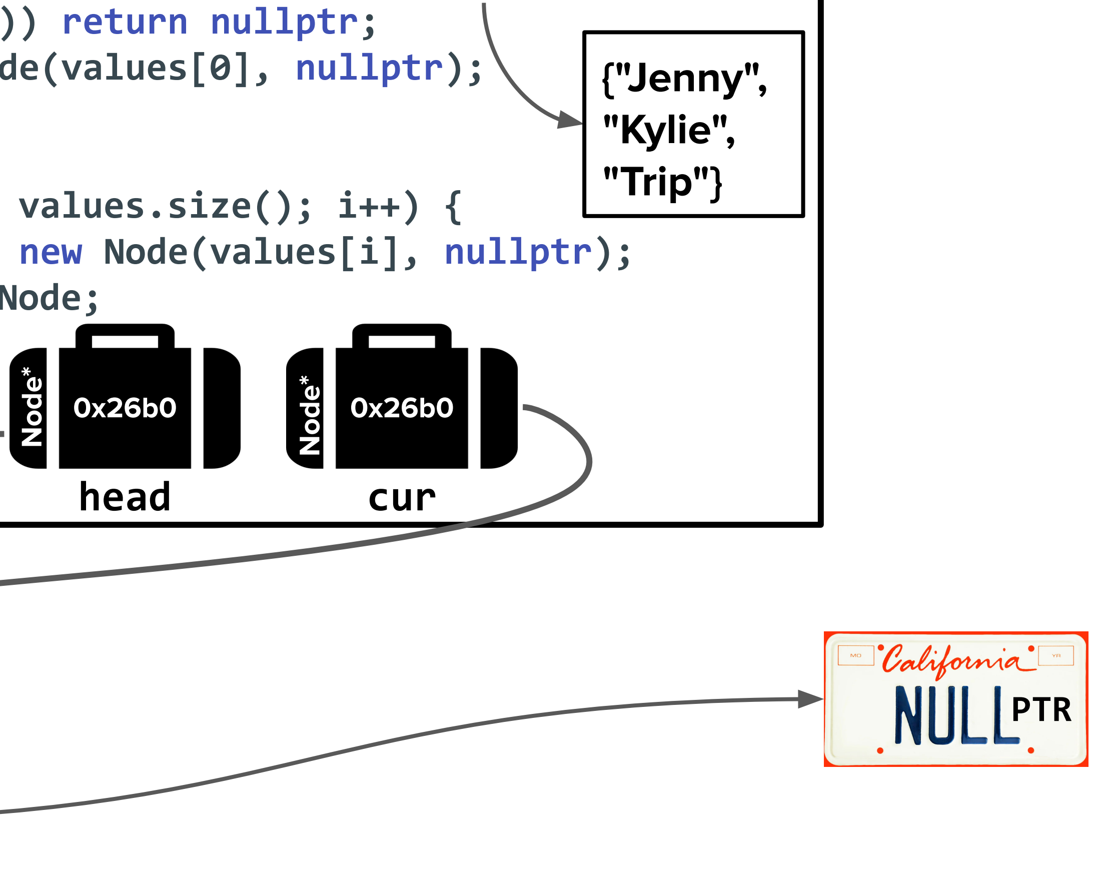


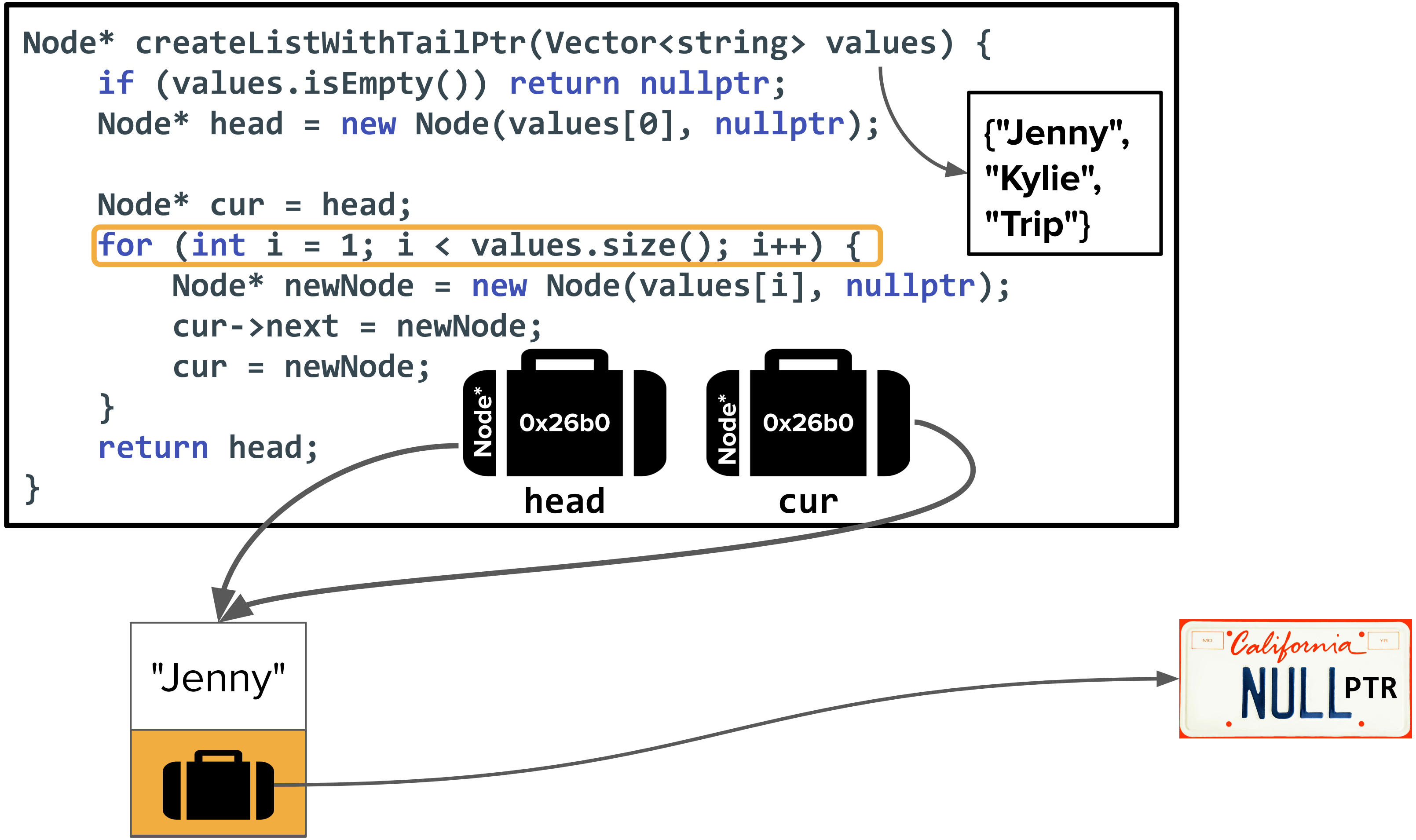
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

{"Jenny",
"Kylie",
"Trip"}



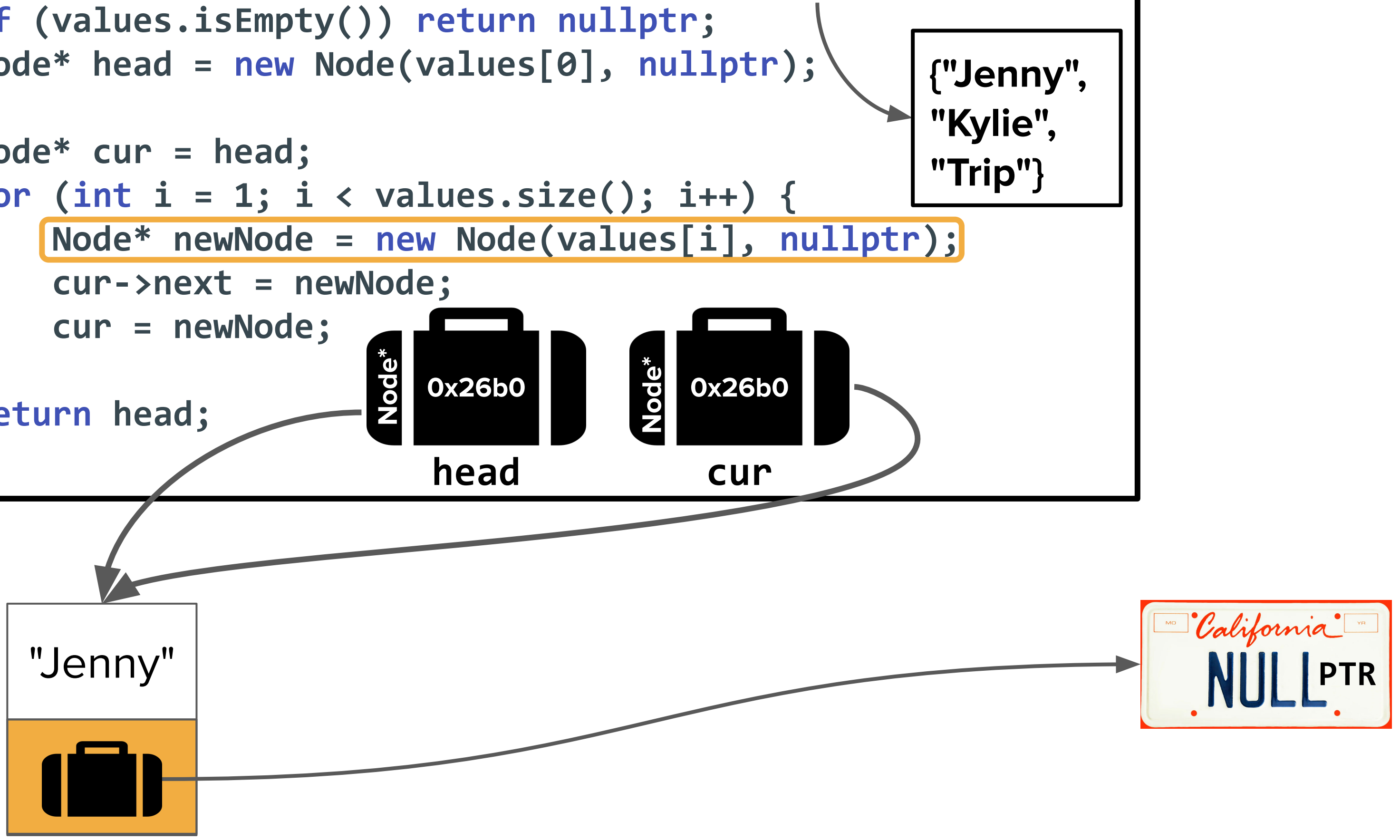
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



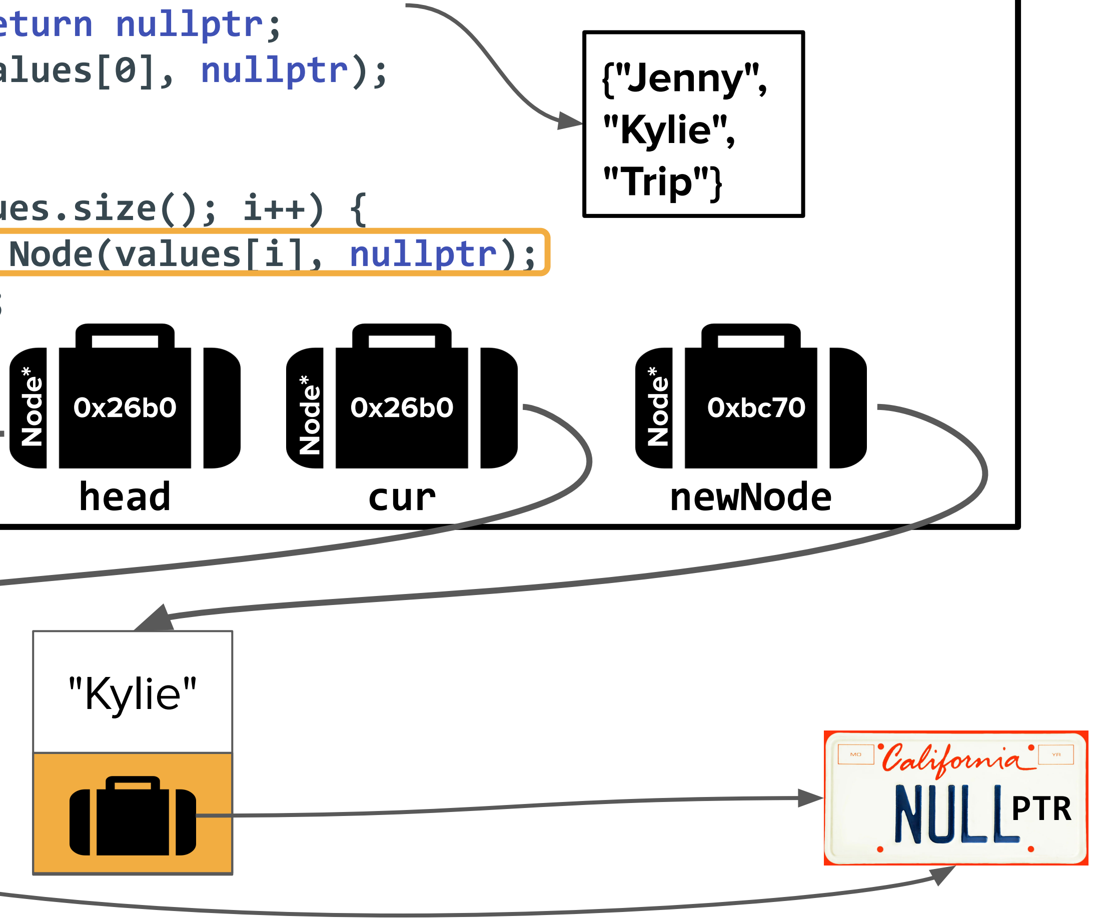


```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

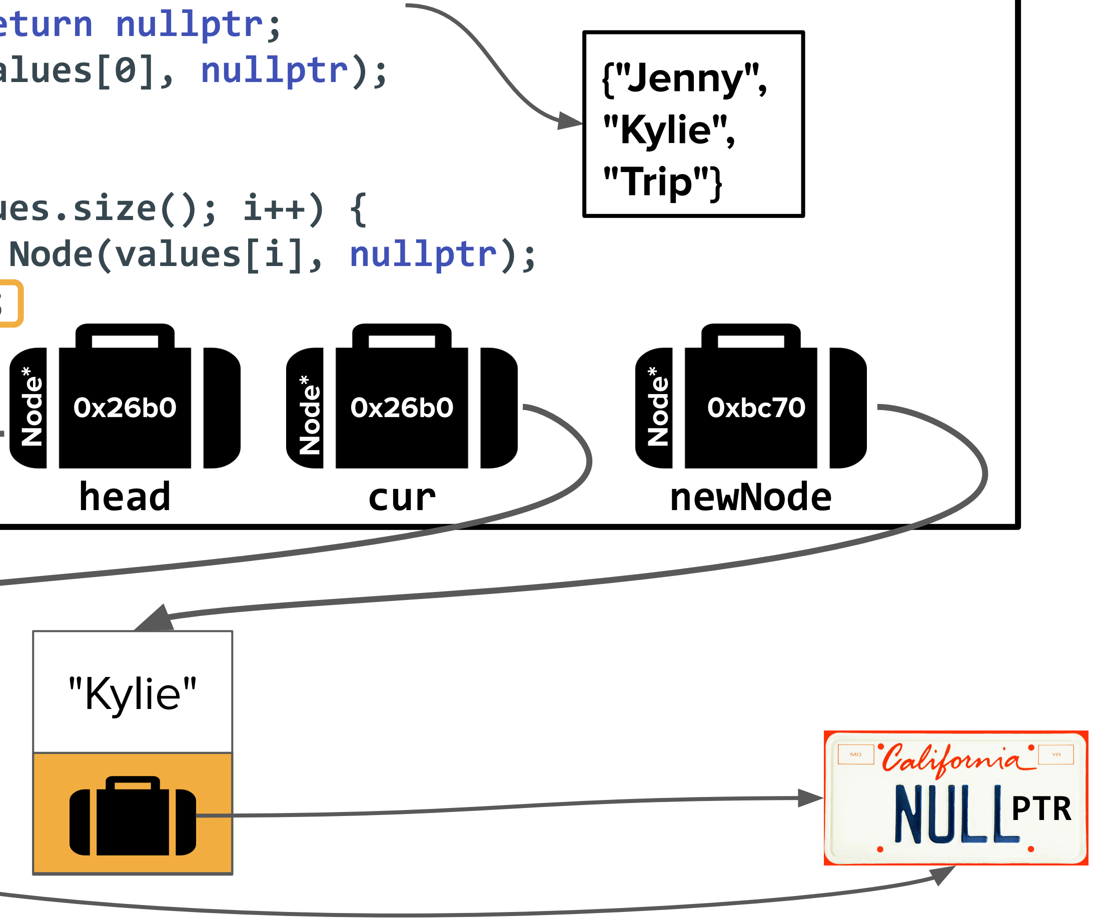
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



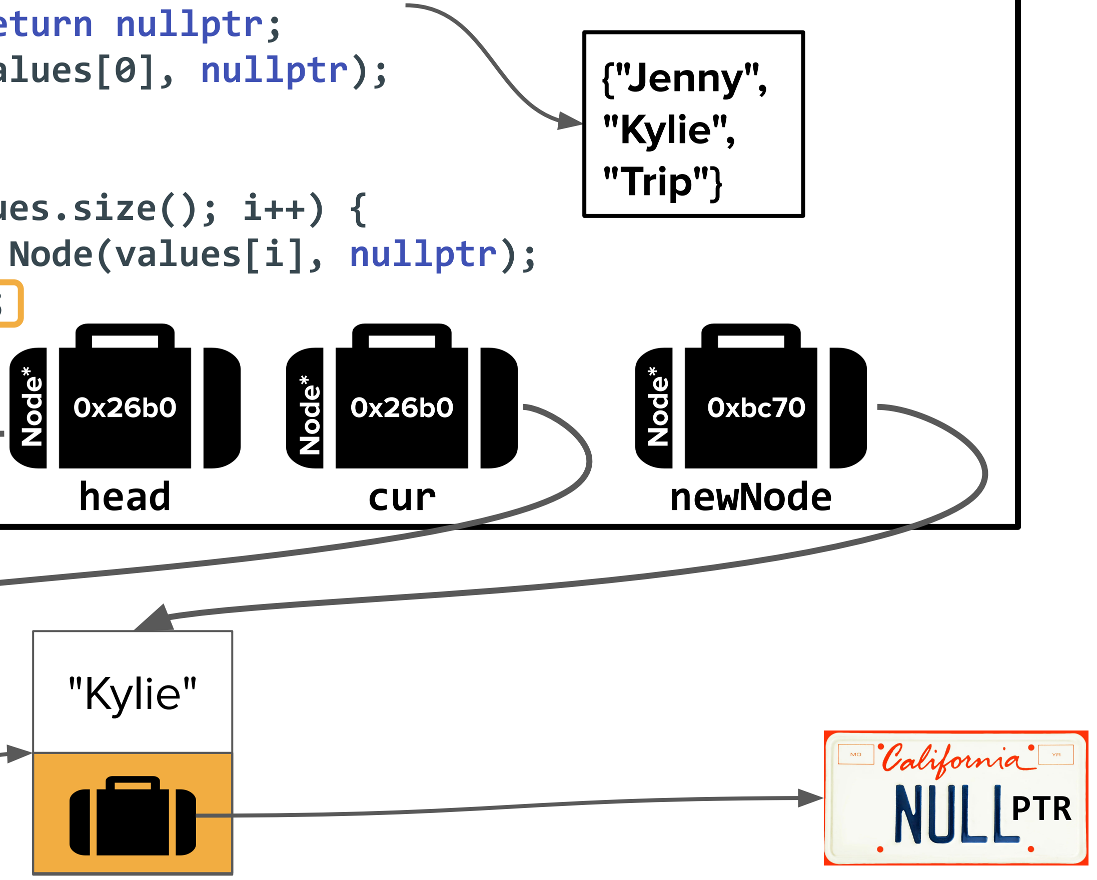
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



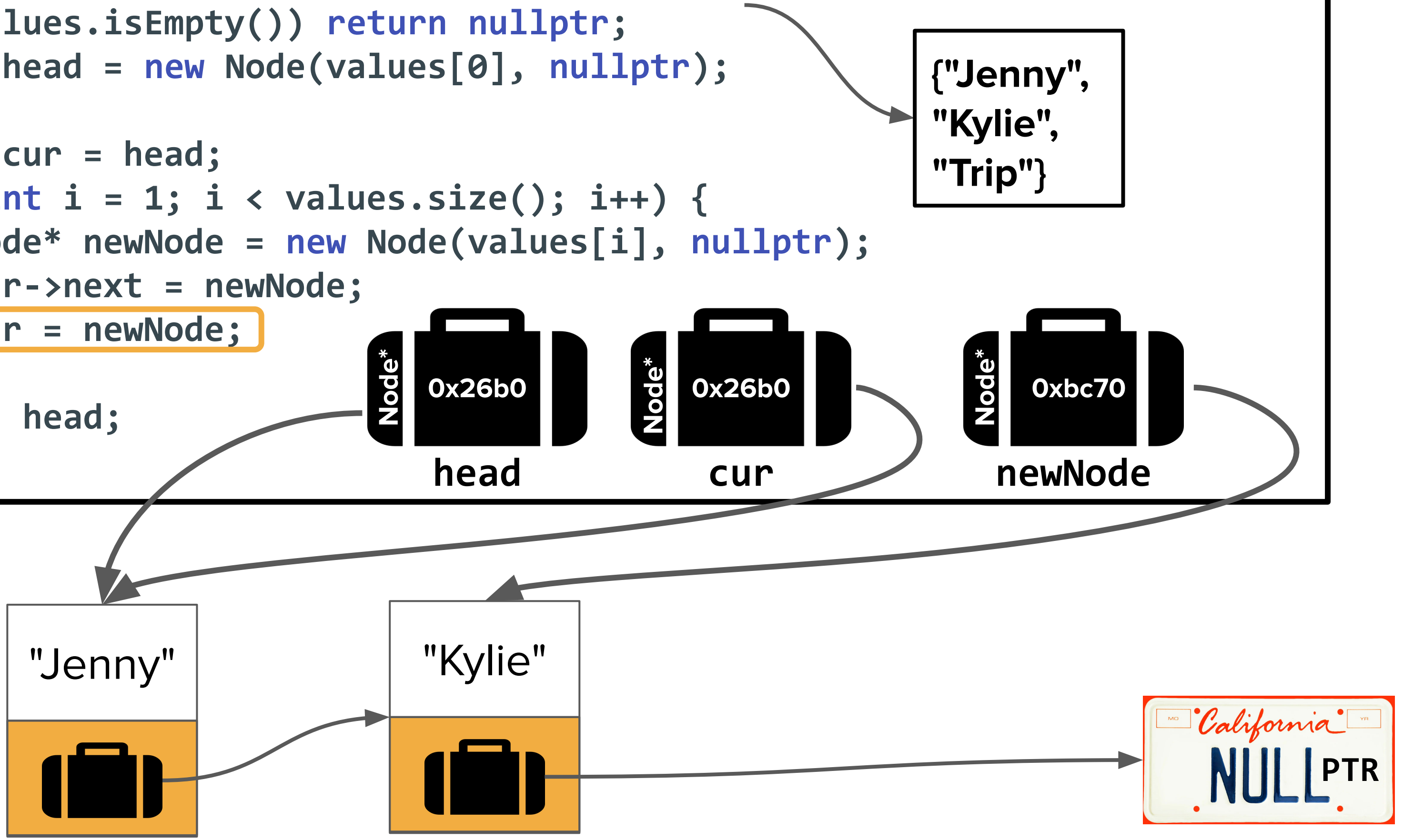
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



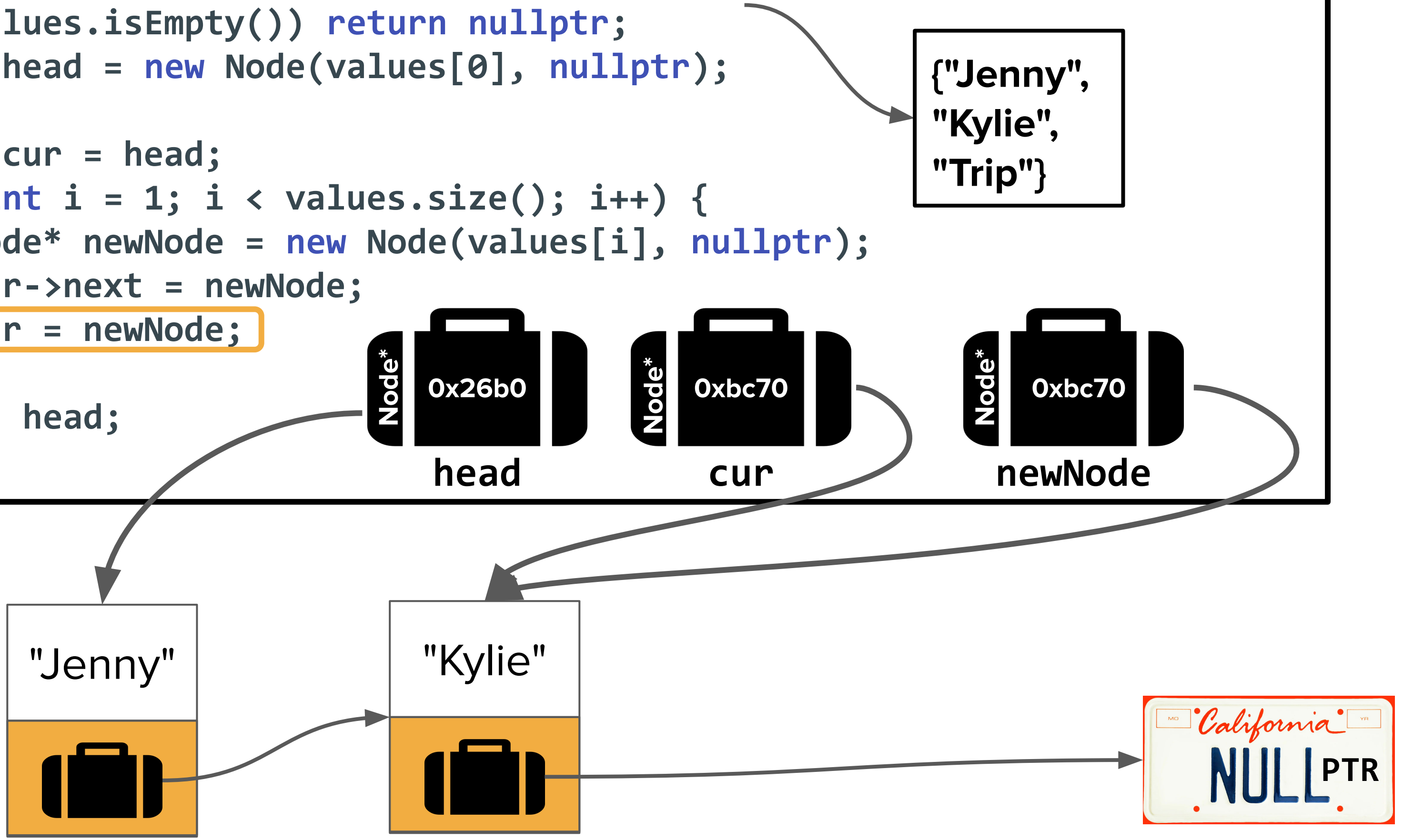
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



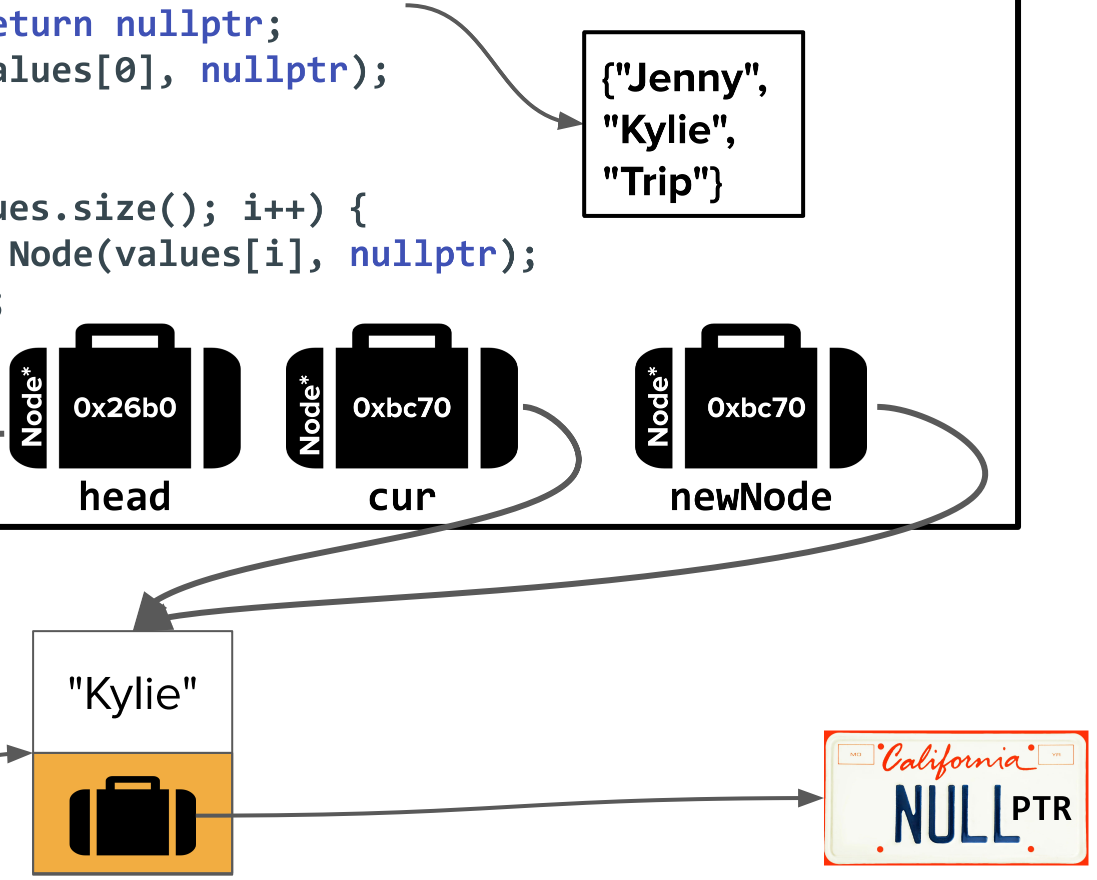
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



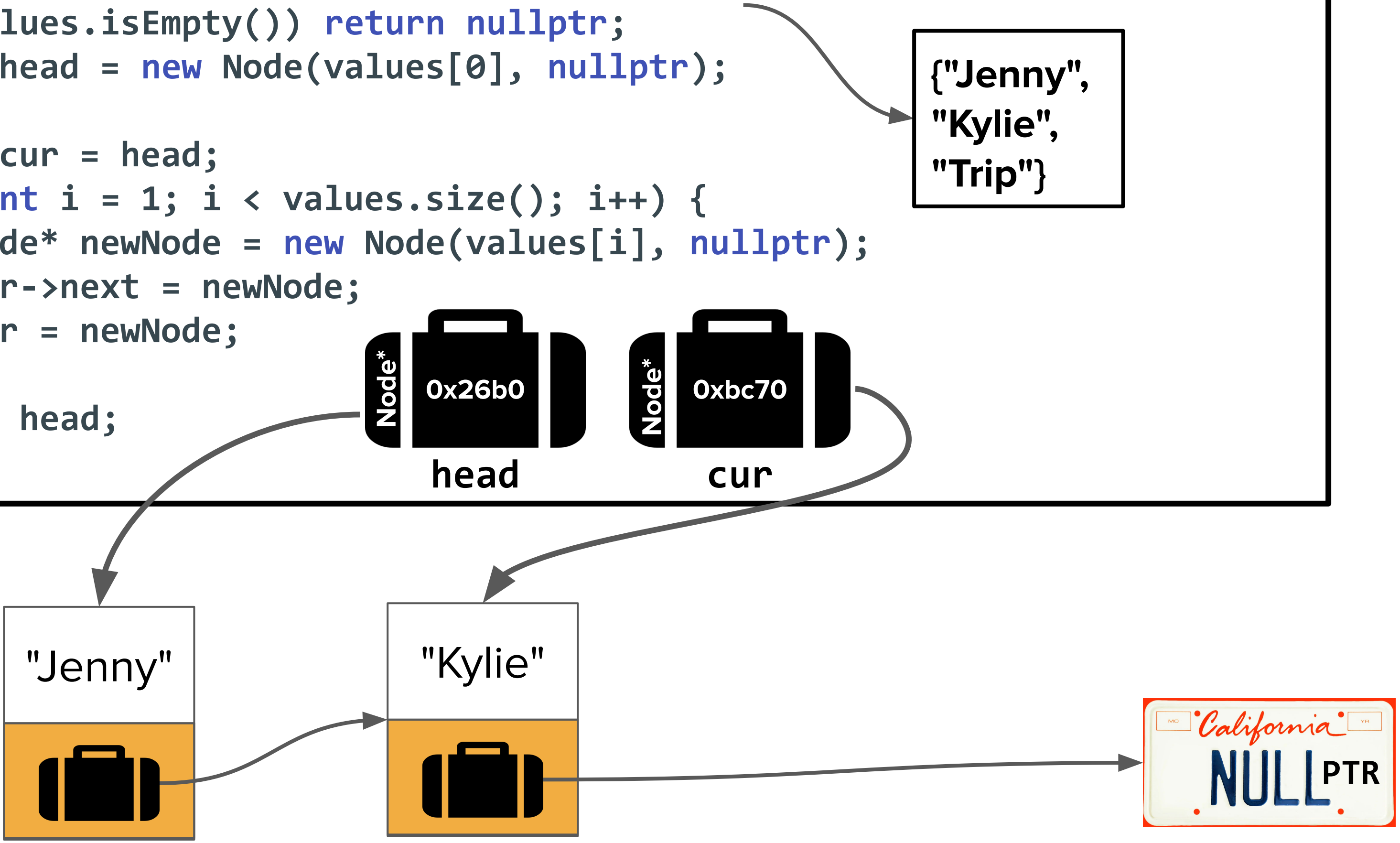
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



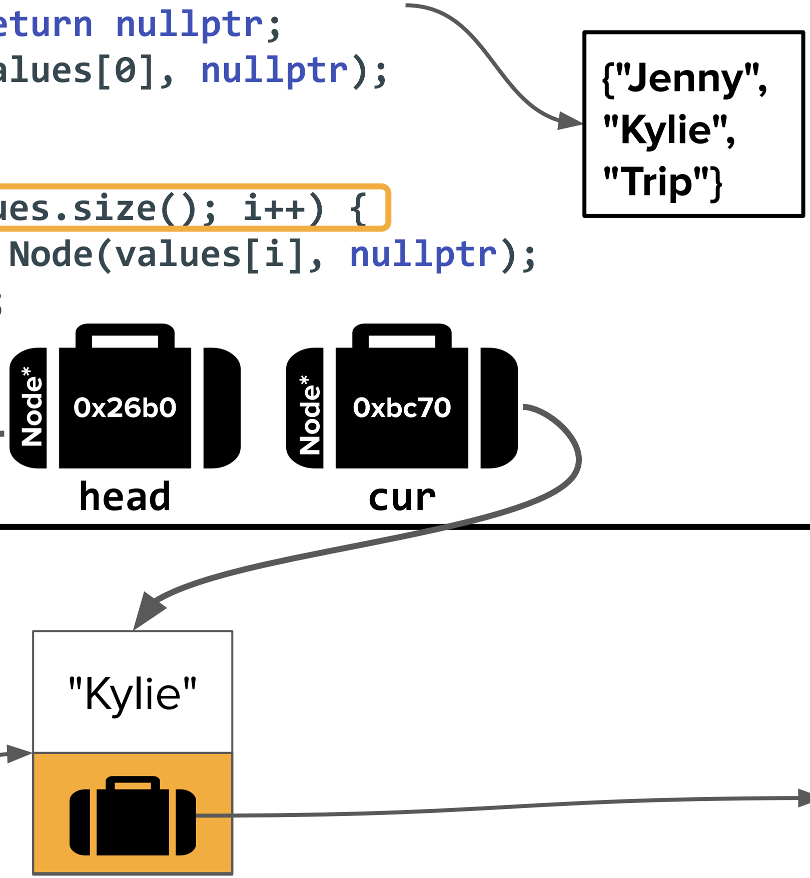
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



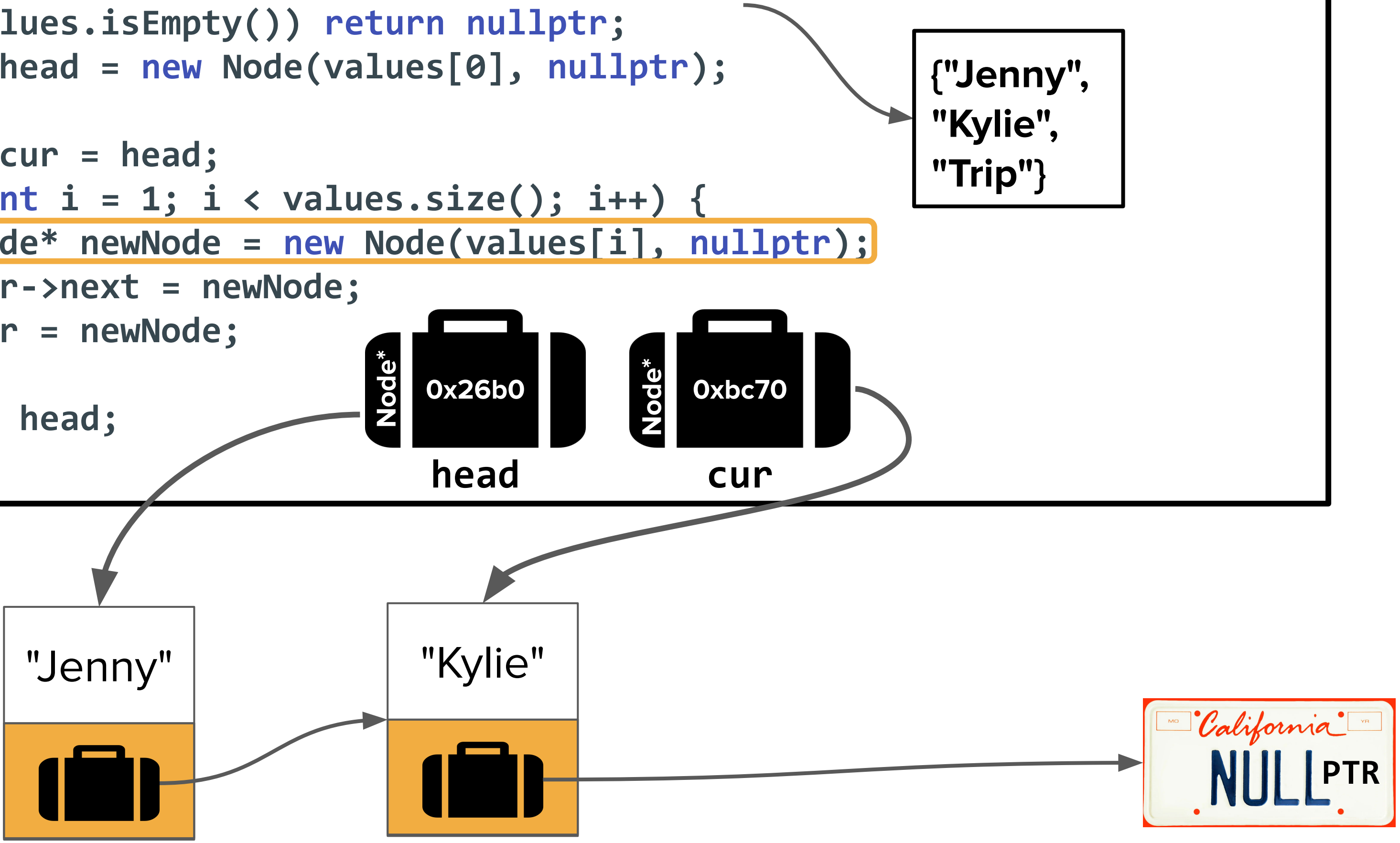
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



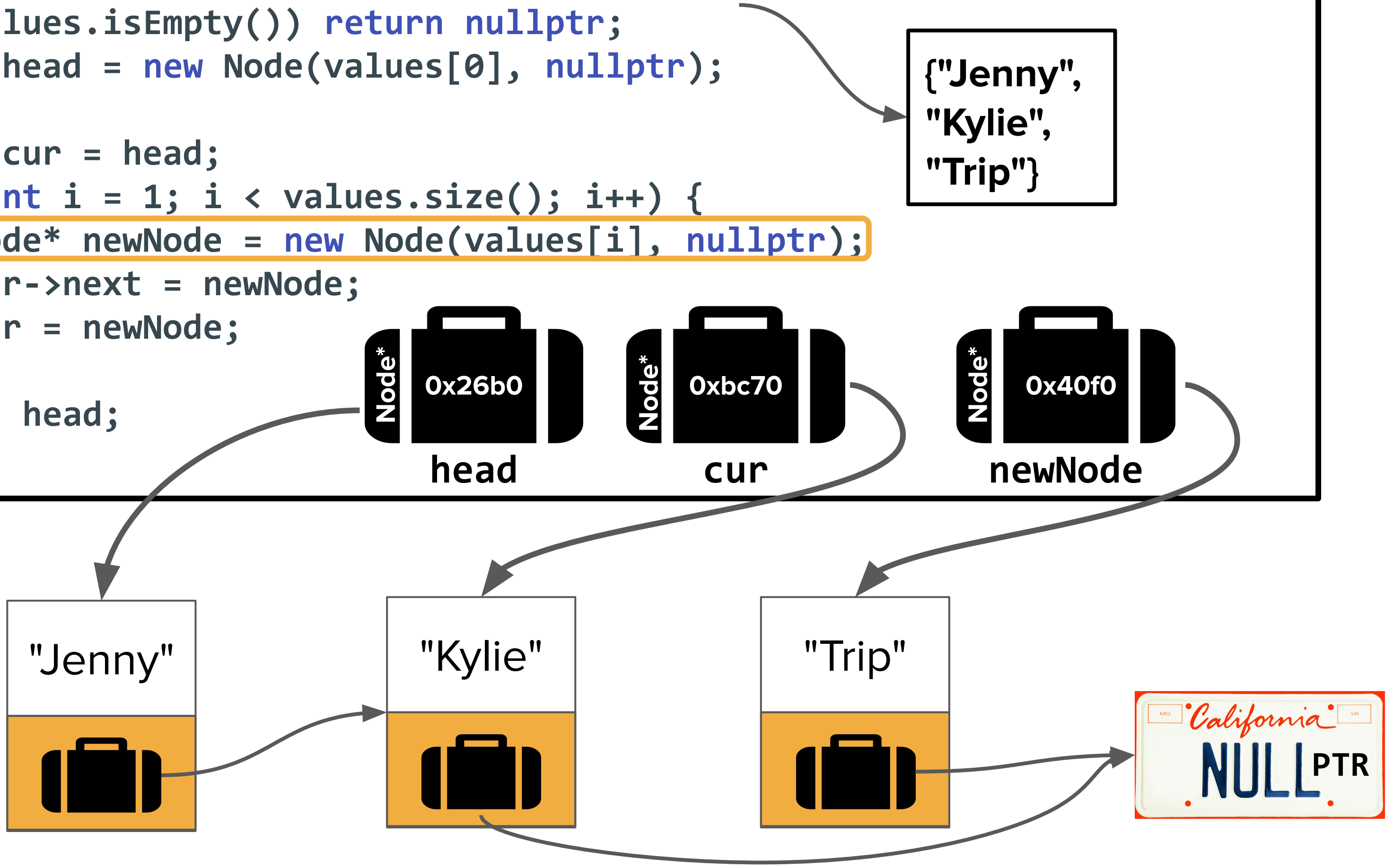
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



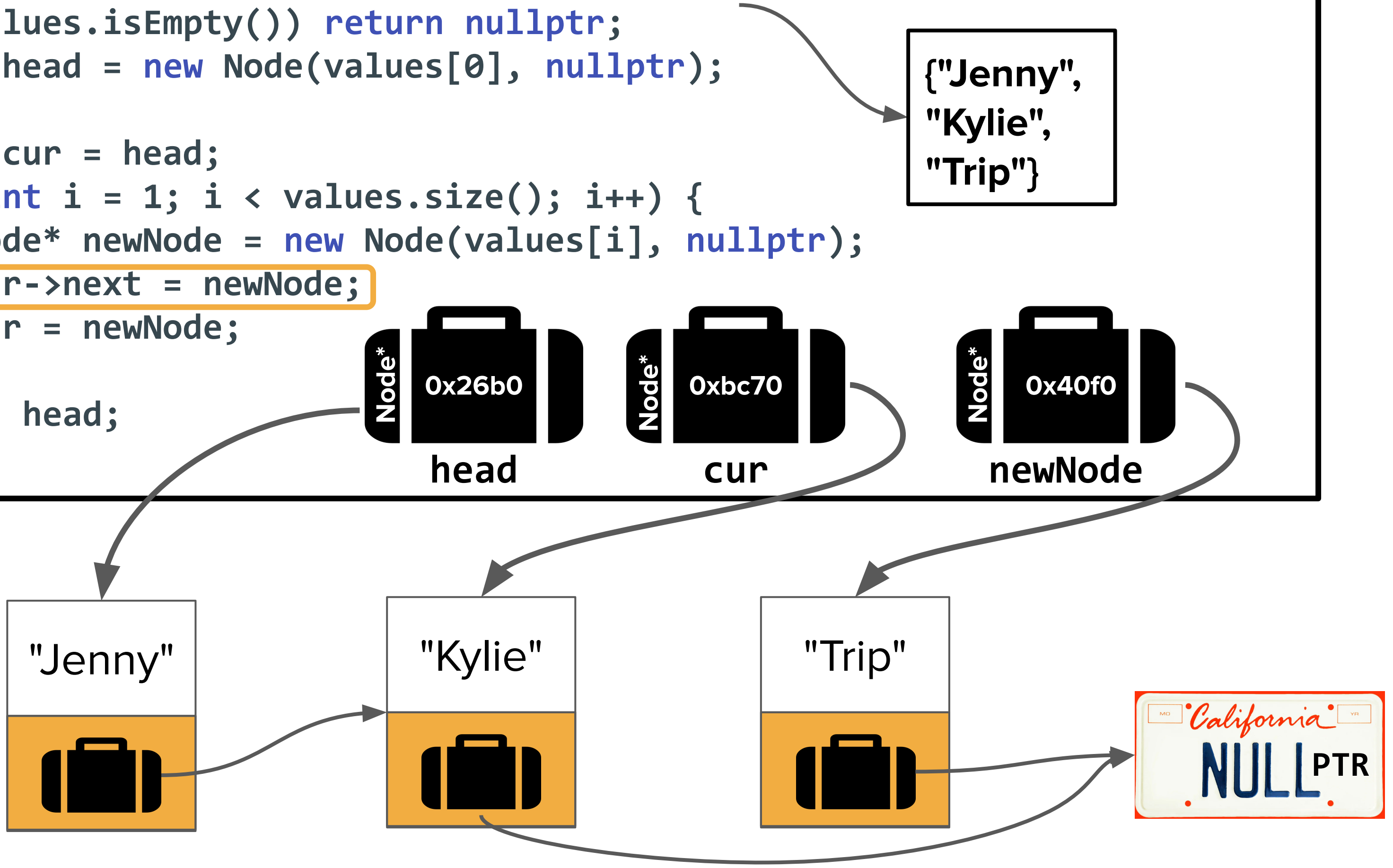
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



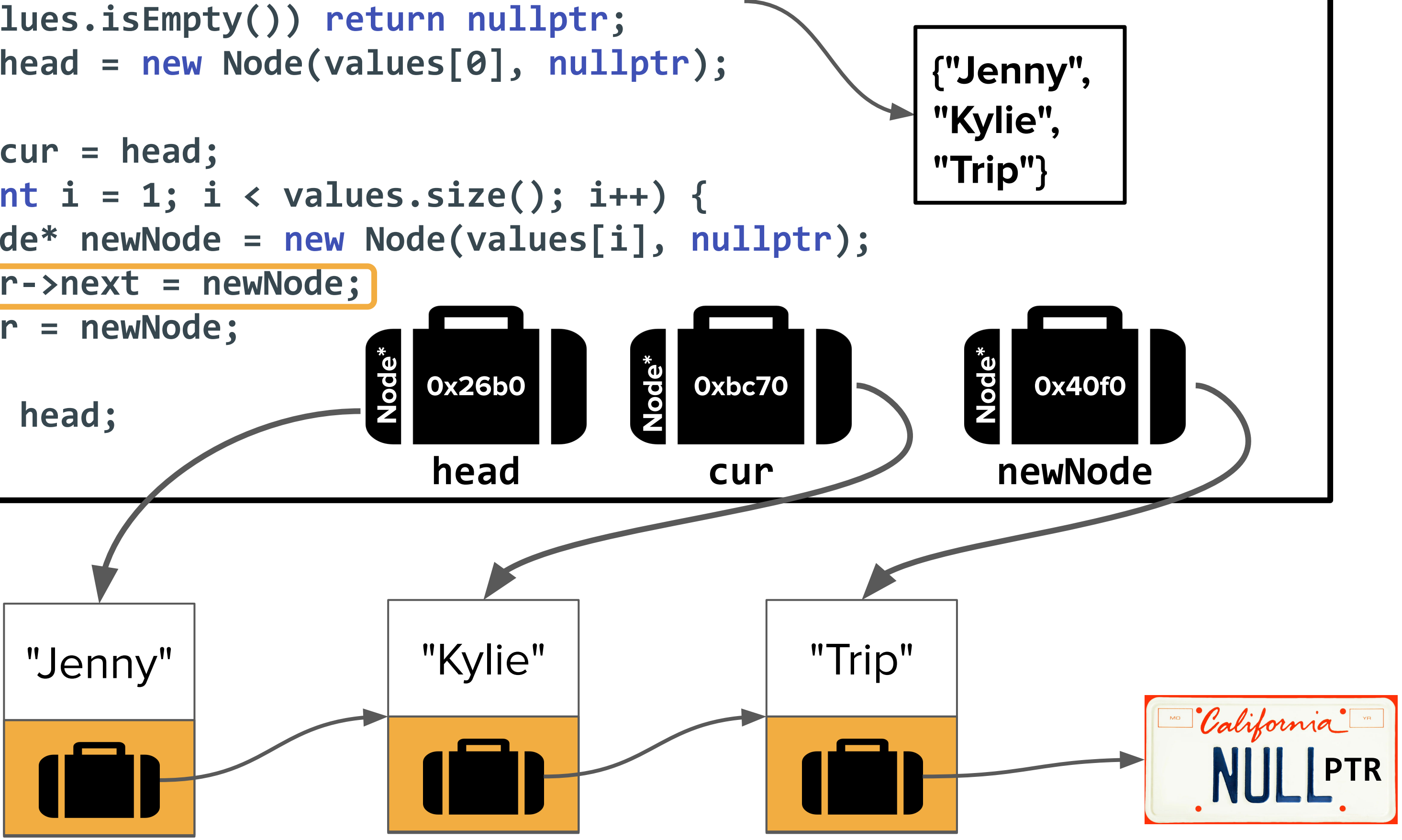
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



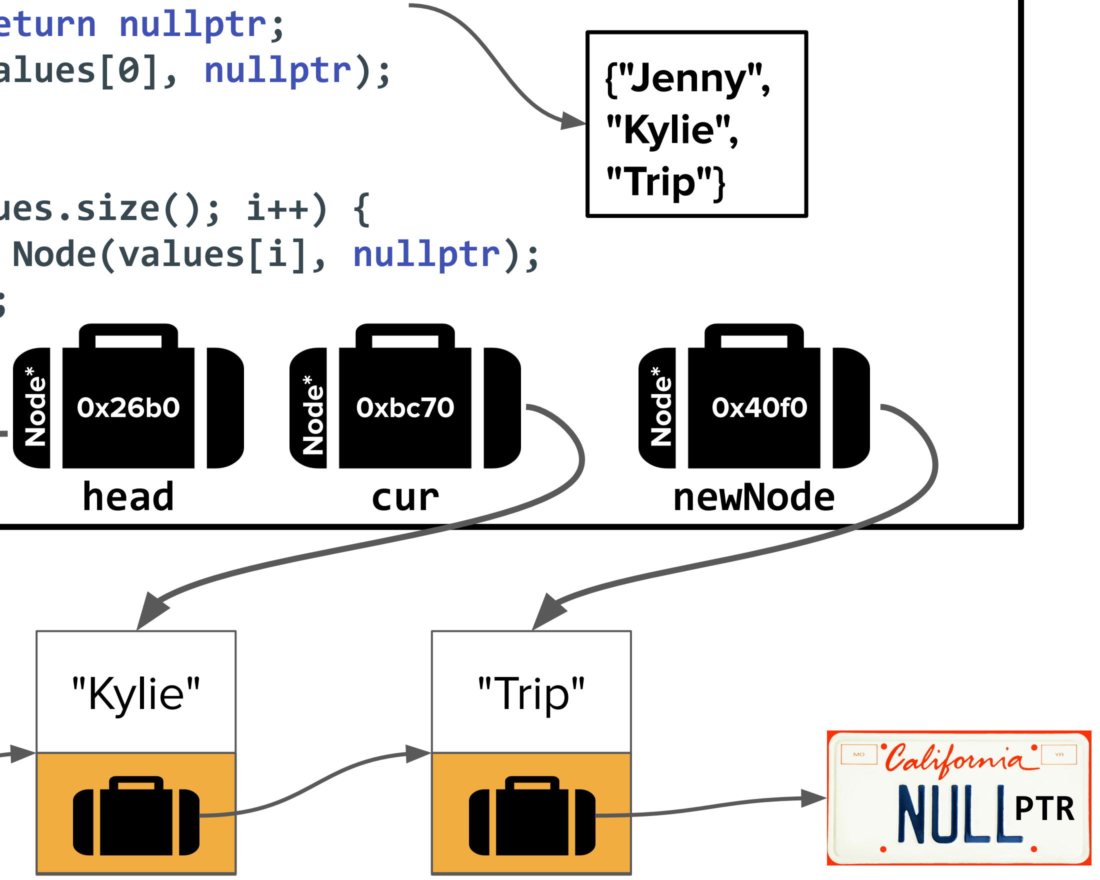
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



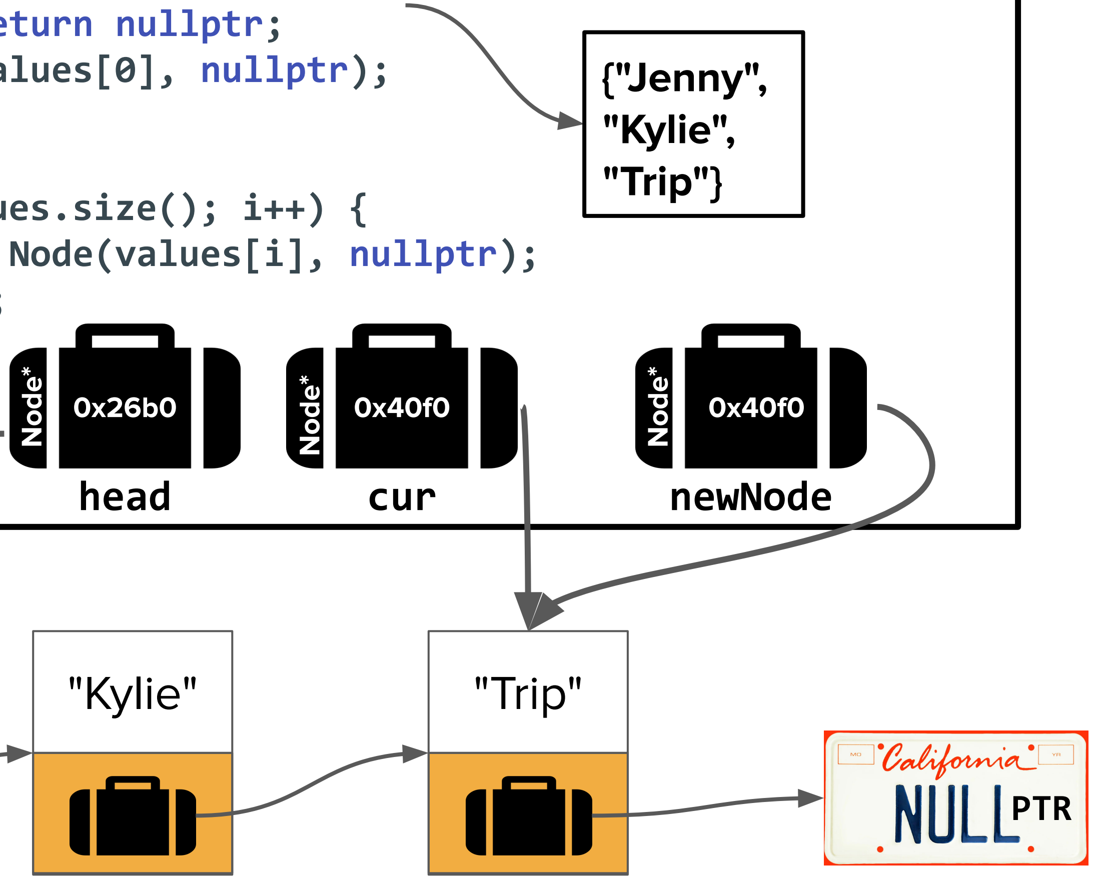
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



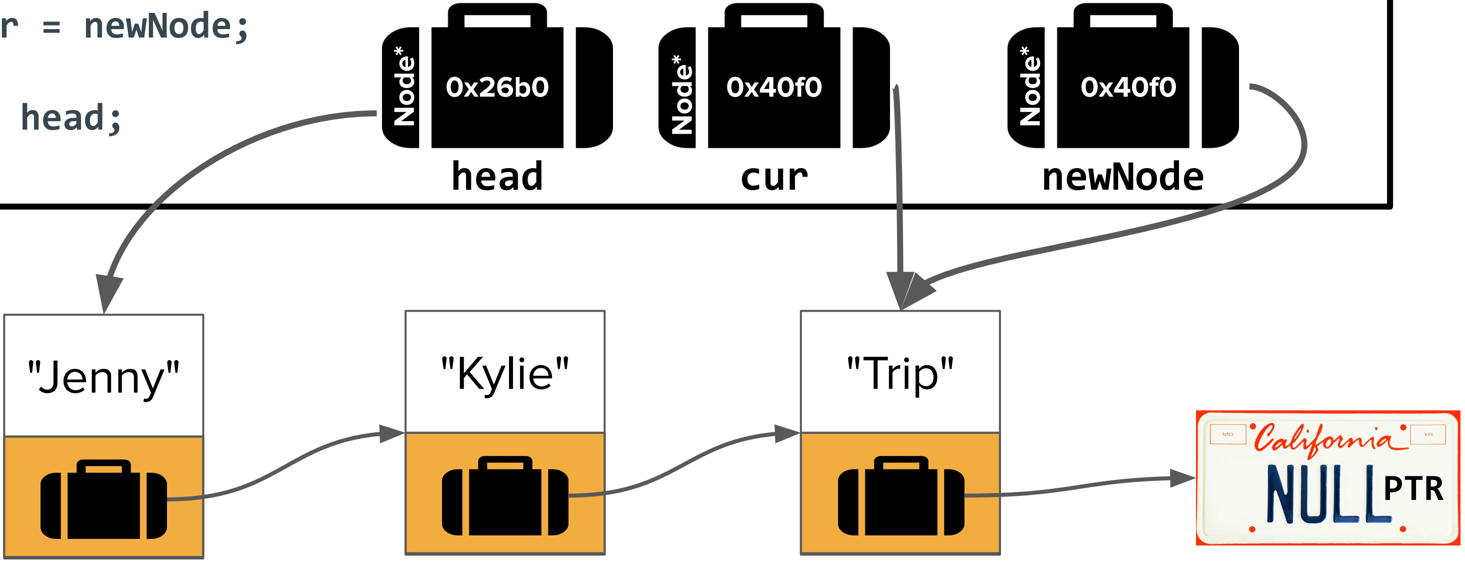
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



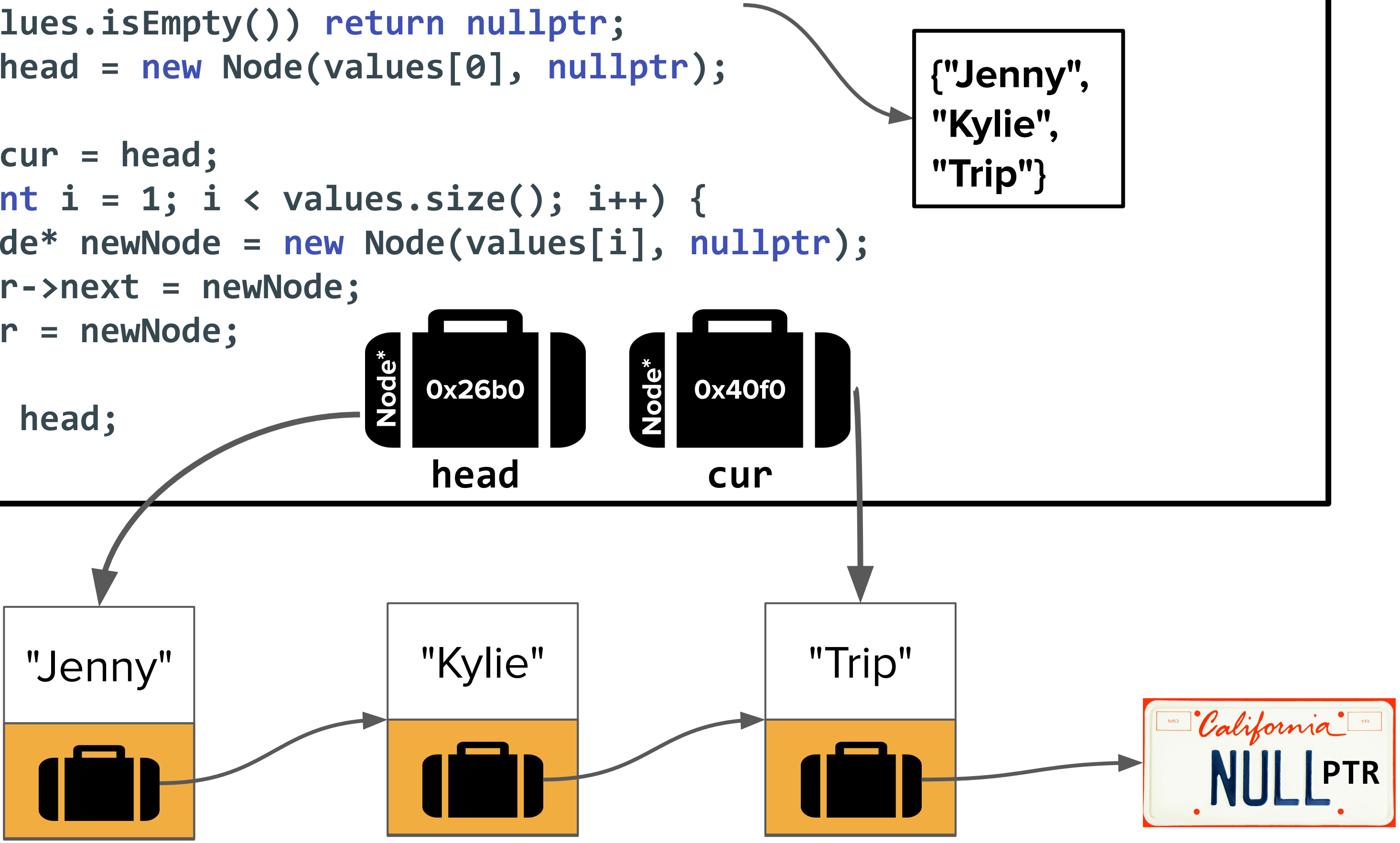
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



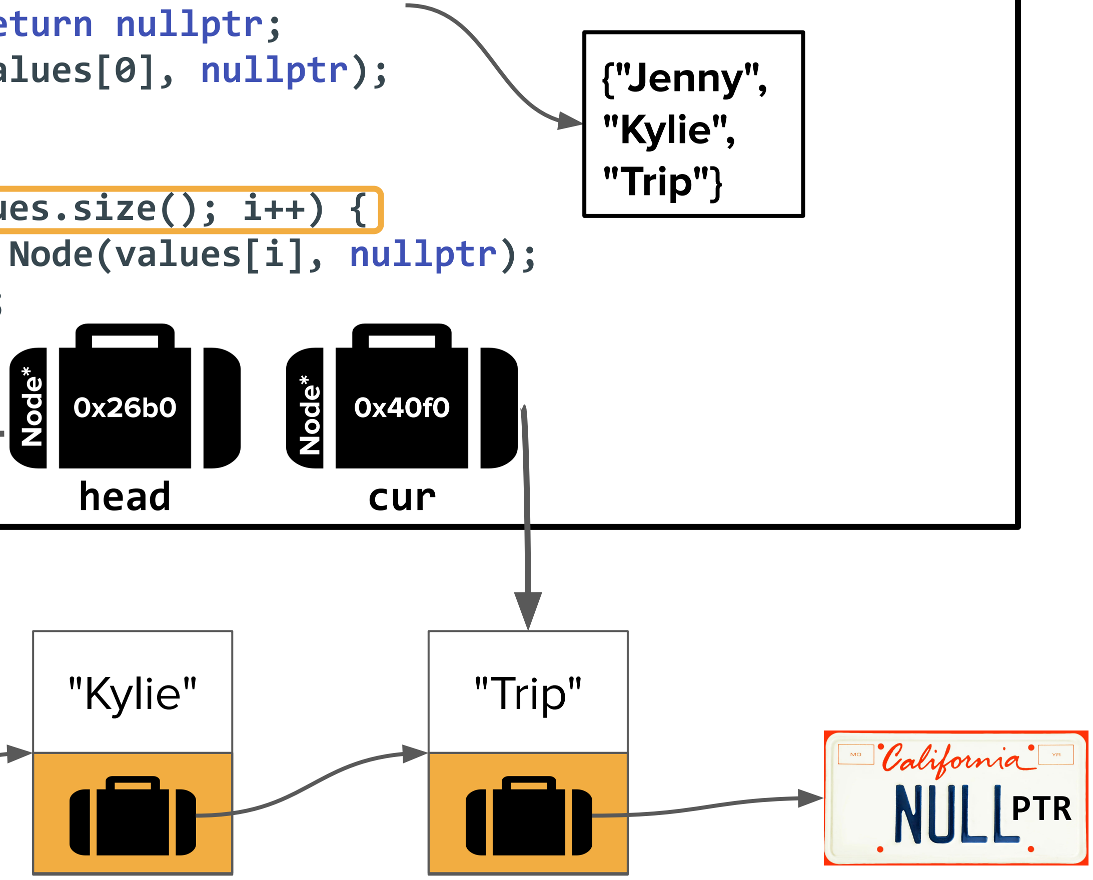
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



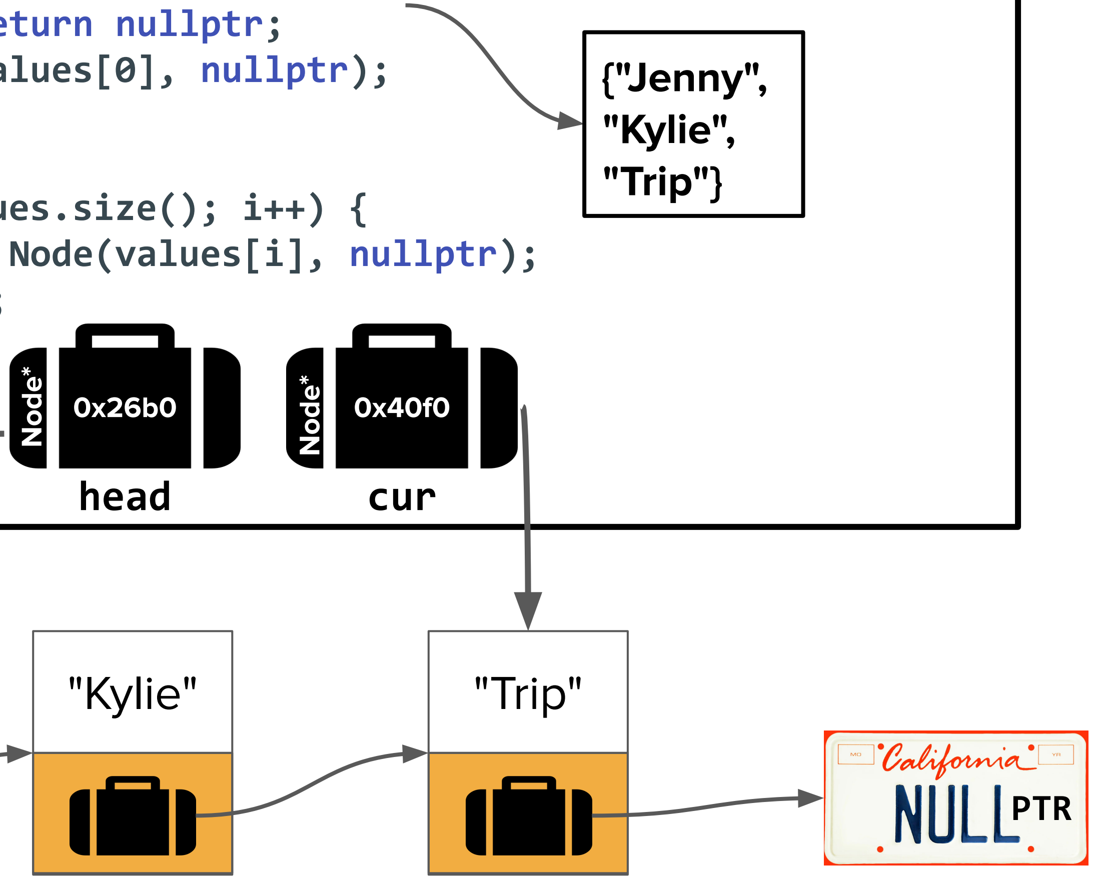
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



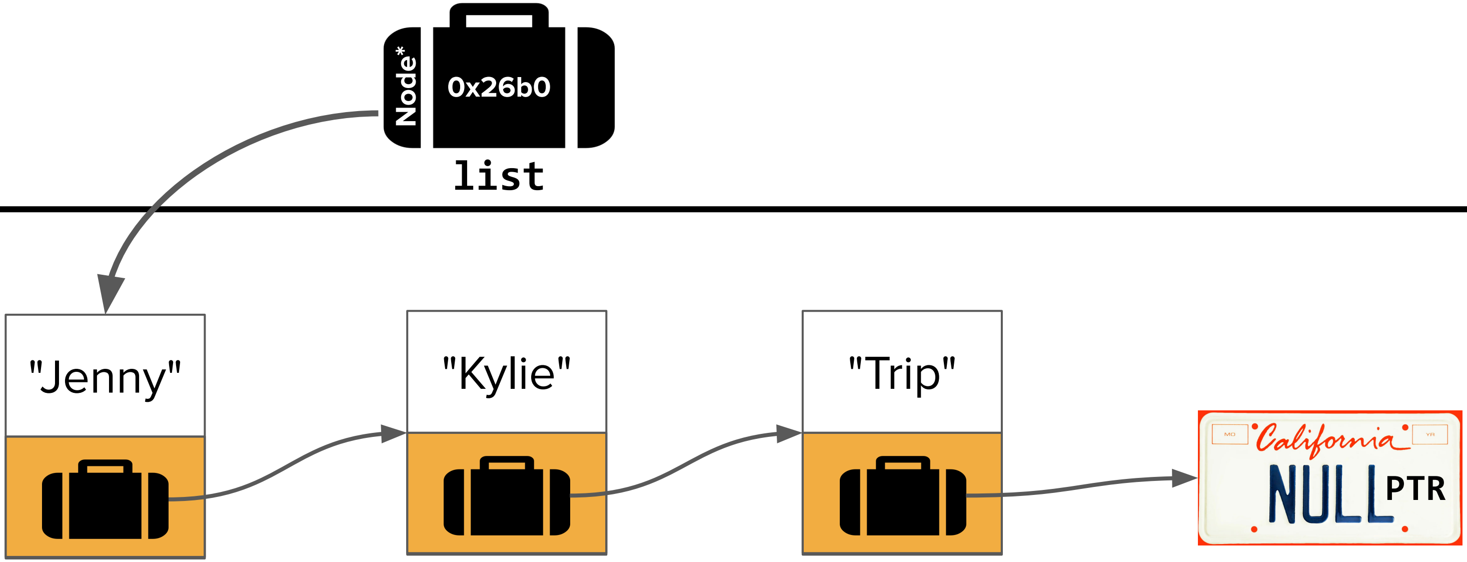
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



```
int main() {
    Vector<string> values = {"Jenny", "Kylie", "Trip"};
    Node* list = createListWithTailPtr(values);

    /* Do other list-y things here, like printing/freeing the list. */
    return 0;
}
```



We just built a linked list with the **desired order of elements maintained** in **`O(n)`** time. Awesome!

Manipulating the middle of a list

Insertion/deletion in the middle of a list

- Why might we want this?
  - To maintain a particular sorted order of the list
  - To find and remove a particular element in the list

Insertion/deletion in the middle of a list

- Why might we want this?
  - To maintain a particular sorted order of the list
  - To find and remove a particular element in the list

- We're going to write two functions:
  - `alphabeticalAdd(Node*& list, string data)`
  - `remove(Node*& list, string dataToRemove)`

Insertion/deletion in the middle of a list

- Why might we want this?
  - To maintain a particular sorted order of the list
  - To find and remove a particular element in the list

- We're going to write two functions:
  - `alphabeticalAdd(Node*& list, string data)`
  - `remove(Node*& list, string dataToRemove)`

Note that we'll need to pass our list by reference!

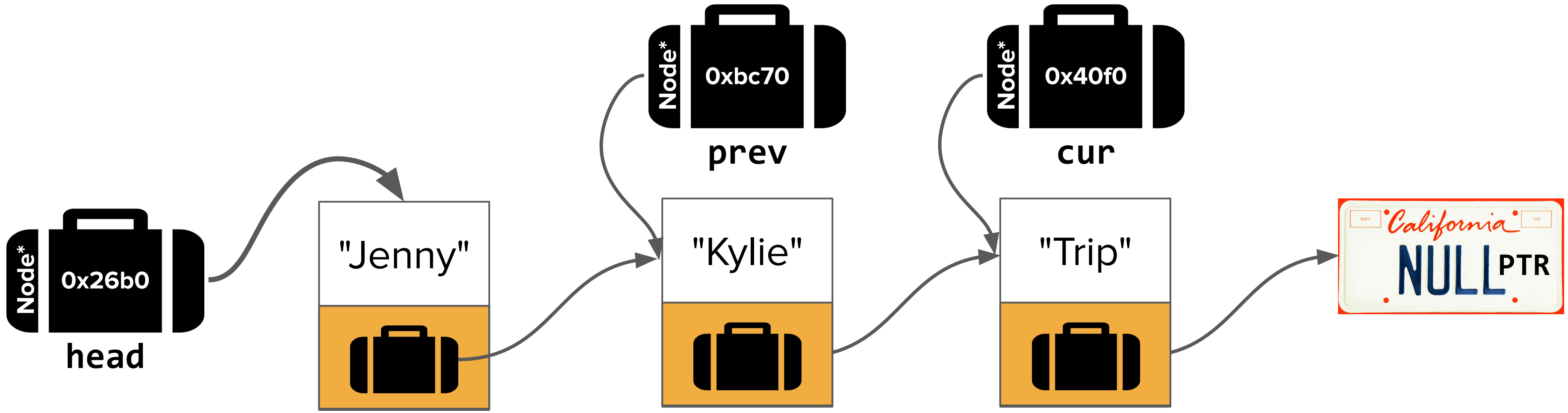
`alphabeticalAdd()` – Let’s code it!

Linked List Deletion

`remove()` –
Check it out at home!

Takeaways for advanced linked list manipulation

- While traversing to where you want to add/remove a node, you'll often want to keep track of both a current pointer and a previous pointer.
    - This makes rewiring easier between the two!
    - This also means you have to check that neither is nullptr before dereferencing.



Linked list summary

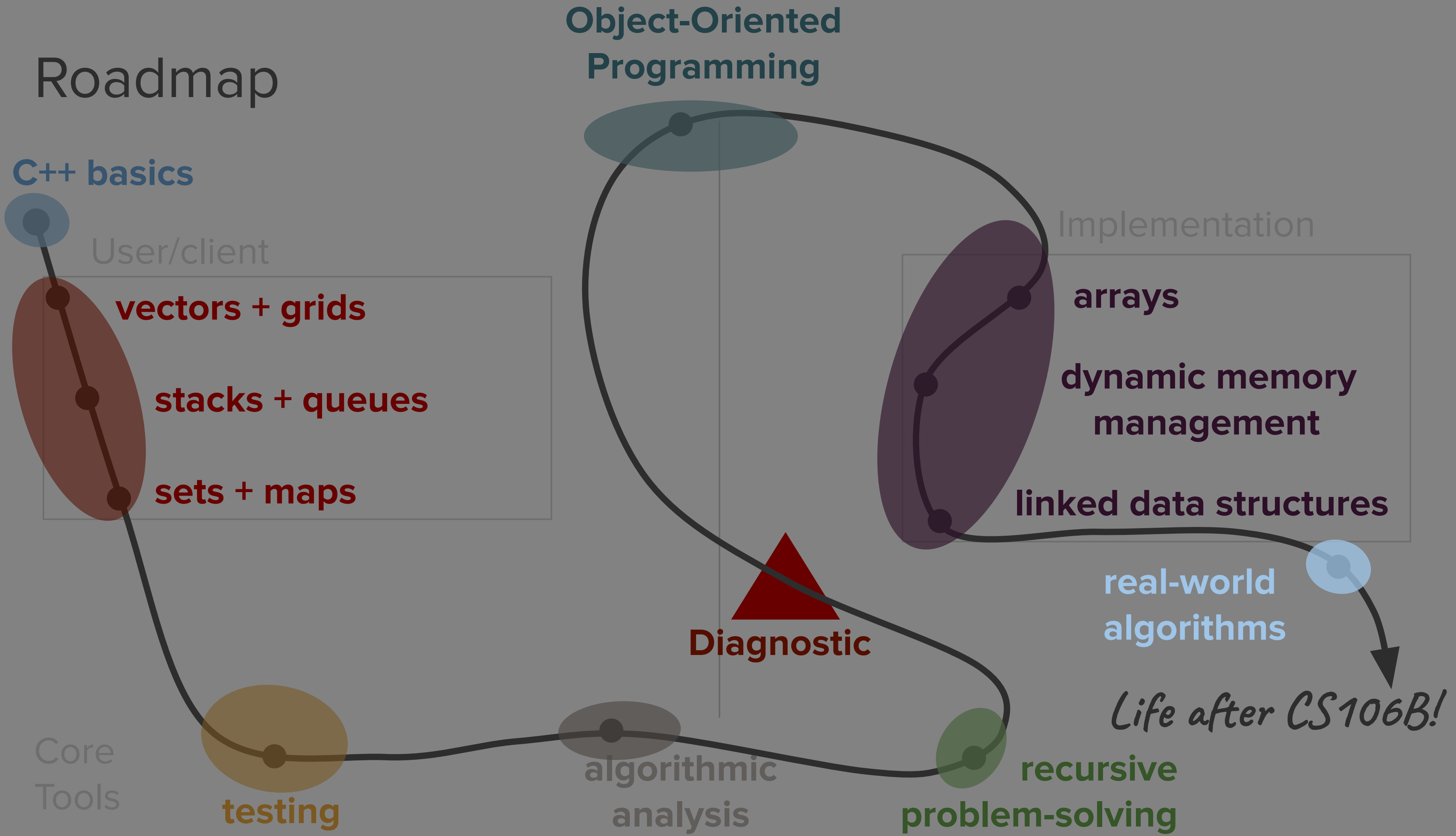
- You’ve now learned lots of ways to manipulate linked lists!
  - Traversal
  - Rewiring
  - Insertion (front/back/middle)
  - Deletion (front/back/middle)

- You’ve seen linked lists in classes and outside classes, and pointers passed by value and passed by reference.

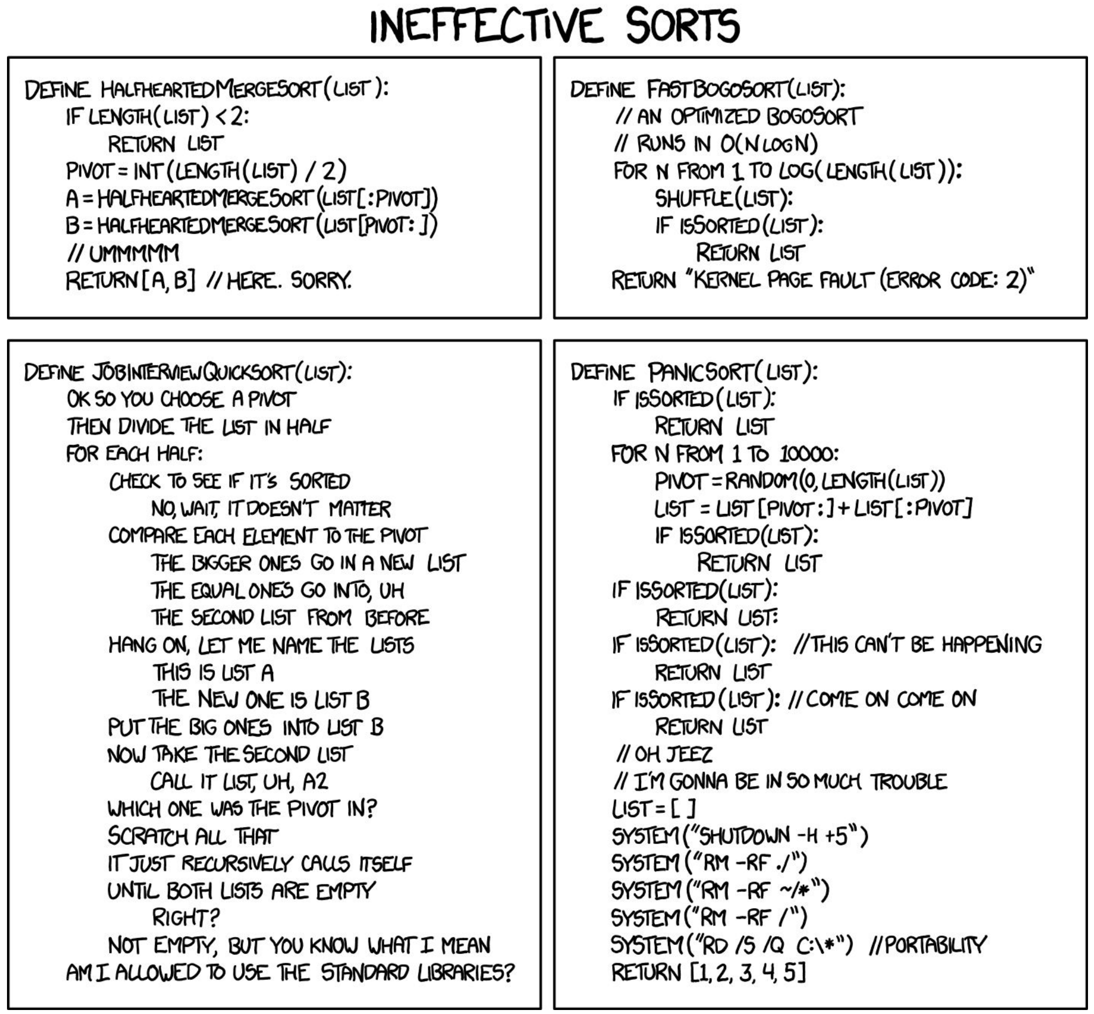
- Assignment 5 will really test your understanding of linked lists.
  - Draw lots of pictures!
  - Test small parts of your code at a time to make sure individual operations are working correctly.

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

Sorting Algorithms!



INEFFECTIVE SORTS

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOGN)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
            NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
            THE BIGGER ONES GO IN A NEW LIST
            THE EQUAL ONES GO INTO, UH
            THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
            THIS IS LIST A
            THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
            CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
            RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST): //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): // COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*") //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```