CS 106X, Lecture 15
Dynamic Memory and Linked Lists

reading:

Programming Abstractions in C++, Chapters 11-12

Plan For Today

- Implementing a Linked List
  - Pointers
  - Dynamic memory
  - Classes
  - Testing

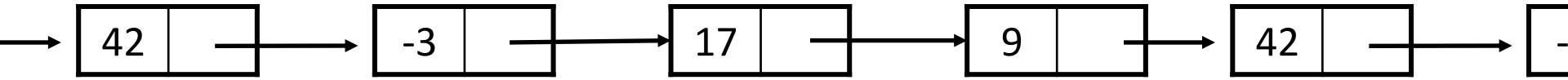
Learning Goals

- Understand why pointers and dynamic memory are necessary to implement a Linked List.

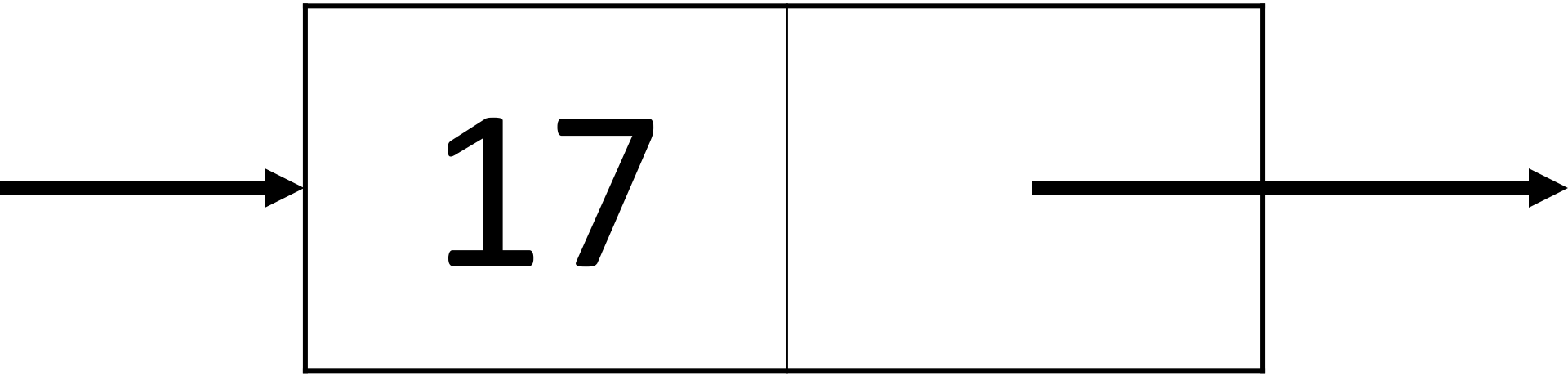
Plan For Today

- Implementing a Linked List
  - Pointers
  - Dynamic memory
  - Classes
  - Testing

Linked Lists 🔗



Nodes



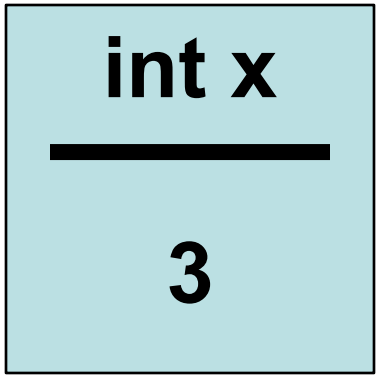

```
struct ListNode {
    int data;
    ListNode *next;
};
```

Pointers

A pointer is a variable type that stores a memory address.

Addresses

42 Wallaby Way




```
int x = 3;
int *xAddress = &x;
```

The `&` operator is the **address of** operator. It gets the address of a variable in memory.

Addresses

```
int x = 3;
int *xAddress = &x;
```

`xAddress` is a **pointer** to **`x`**. It is a variable that "points to" another variable, meaning that it stores the address of another variable.

Addresses

```
int x = 3;
int *xAddress = &x;
```

`x` is the **pointee** of **`xAddress`**. It is being pointed to by **`xAddress`**.

Dereferencing

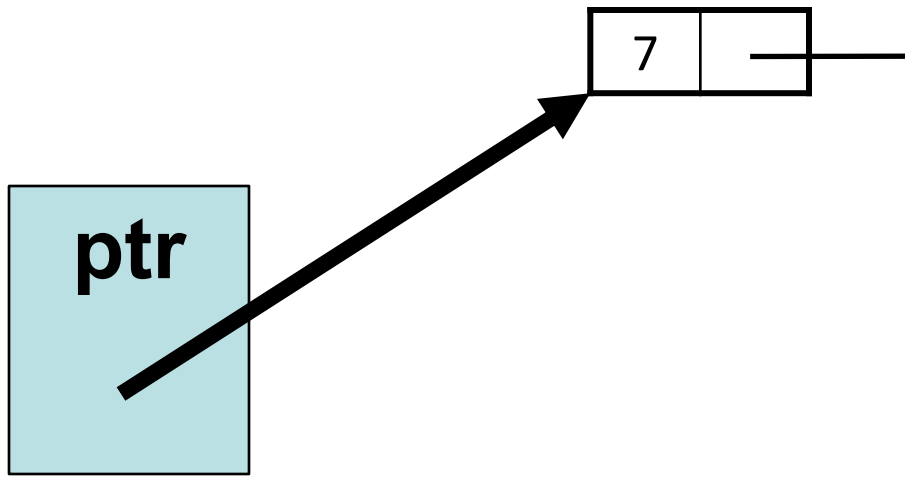
```
int x = 3;
int *xAddress = &x;

*xAddress = 5;
```

The * operator is the **dereference** operator. It tells C++ to *go to the variable* at the address stored in that pointer.

Dereference Classes/Structs

```
ListNode n = ...
ListNode *ptr = &n;

ptr->data = 7;
```

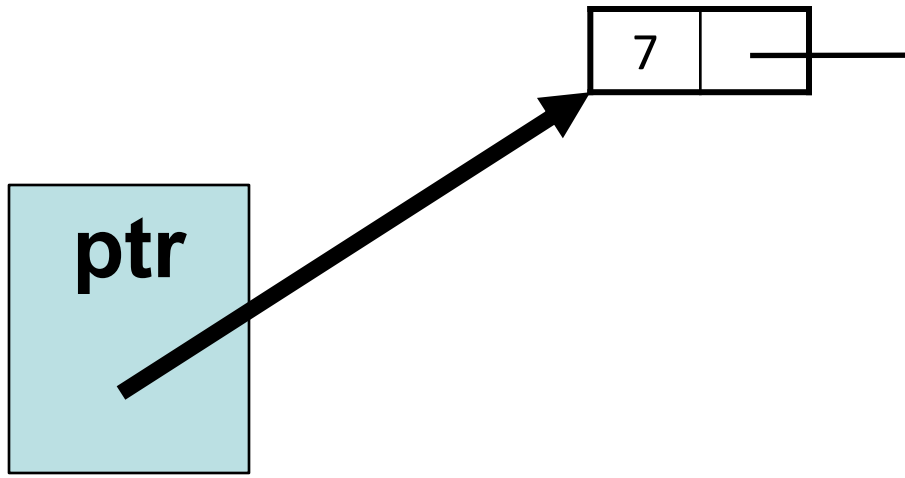
The -> operator is shorthand for dereferencing a pointer and then accessing a member.



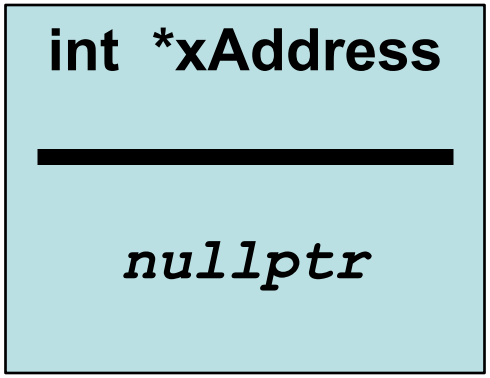
Dereference Classes/Structs

```
ListNode n = ...
ListNode *ptr = &n;

(*ptr).data = 7;
```

The -> operator is shorthand for dereferencing a pointer and then accessing a member.



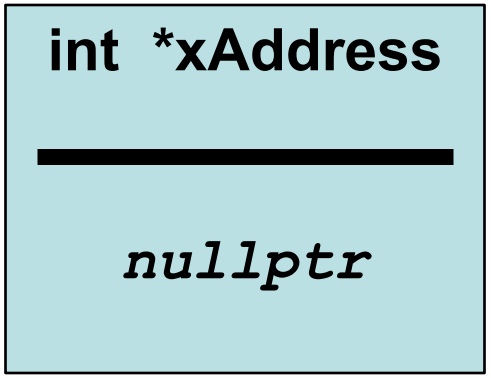
nullptr




```
int *xAddress = nullptr;
```

`nullptr` is a special value that represents "no address".

Dereferencing nullptr




```
int *xAddress = nullptr;
cout << *xAddress << endl;
```

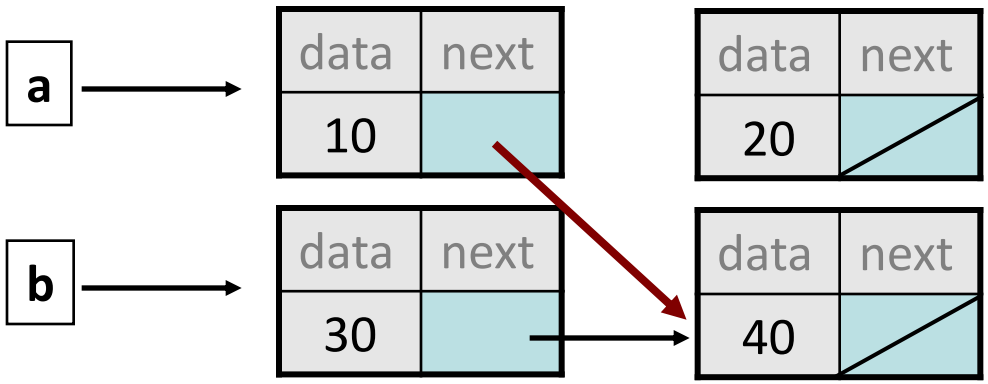
Console

```
...

***
*** STANFORD C++ LIBRARY
*** A segmentation fault occurred during program execution.
*** This typically happens when you try to dereference a pointer
*** that is NULL or invalid.
***
*** Stack trace (line numbers are approximate):
*** 0x10ff14086      main()
```

Garbage Pointers


```
int *xAddress; // initially garbage ❌
cout << xAddress << endl; // ???
cout << *xAddress << endl; // likely crash!


// always initialize pointers!
// (even just to nullptr)
int *xAddress = nullptr;  // ✅
```

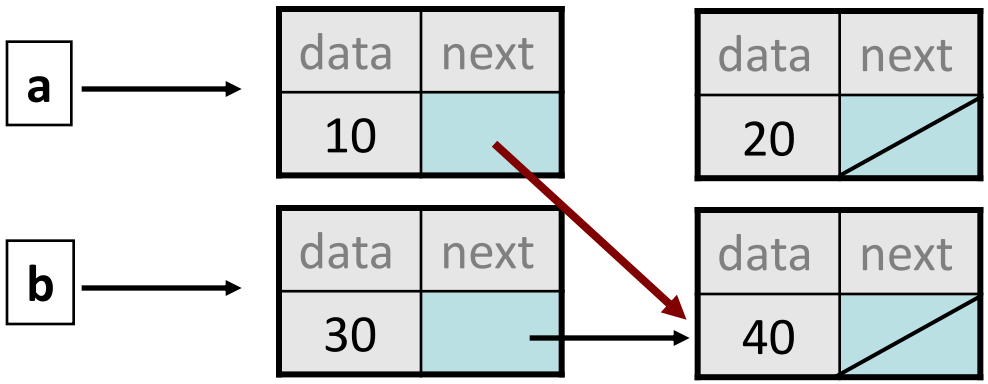
Reassigning Pointers




```
a->next = b->next;
```

Setting two pointers equal to each other means they both *point to the same place*.

Reassigning Pointers




```
ListNode secondNode = {40, nullptr};
```

~~`a->next = secondNode;`~~

Tip: the types on the left- and right-hand sides must always match!

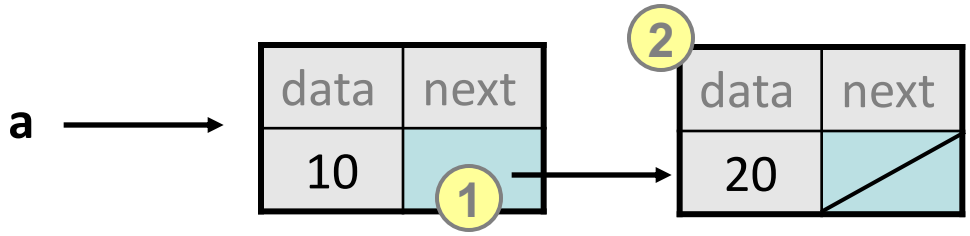
Pointer to struct/obj

variable `=` ***value;***

a *variable* (left side of =) is an arrow *(the base of an arrow)*

a *value* (right side of =) is an object *(a box; what an arrow points at)*

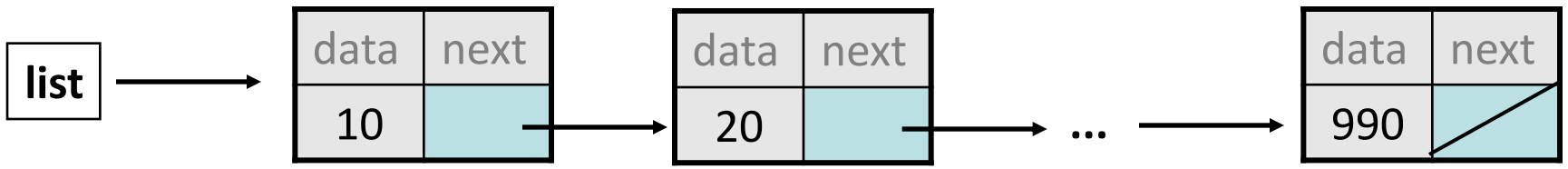
- For the list at right:



`a->next =` ***p;***
means to adjust (1) to point where ***p*** points

p `= a->next;`
means to make ***p*** point where `a->next` points, which is at (2)

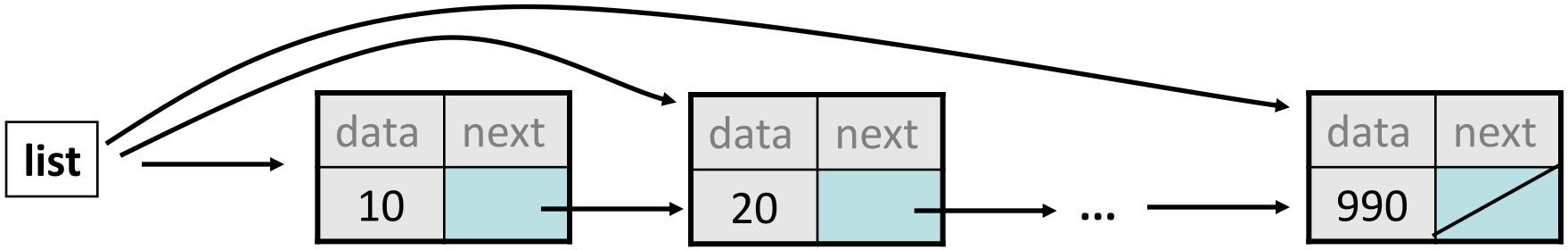
Traversing a Linked List



How do we print out the entire list, regardless of its length?

Traversing a list?

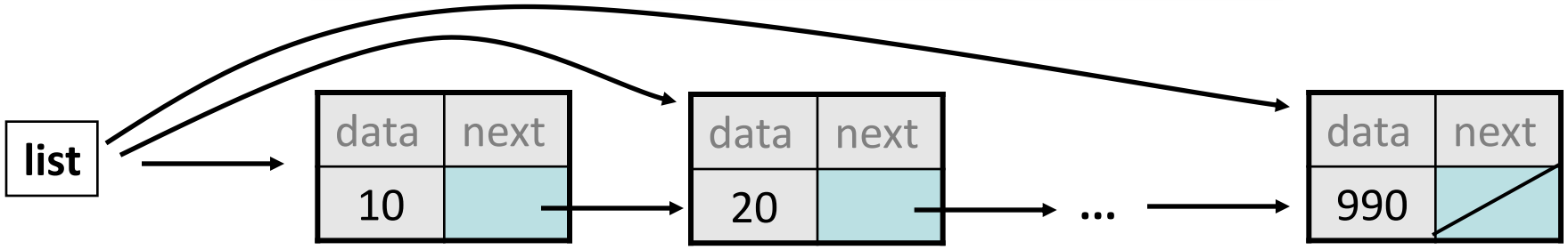
```
while (list != nullptr) {
    cout << list->data << endl;
    list = list->next;    // move to next node
}
```



Traversing a list?

```
while (list != nullptr) {
    cout << list->data << endl;
    list = list->next;      // move to next node
}
```

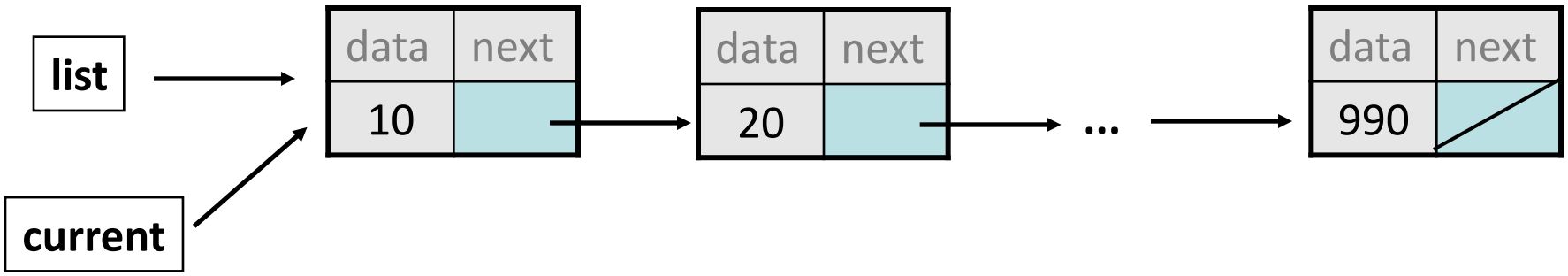
This modifies our only reference to the head of the list!



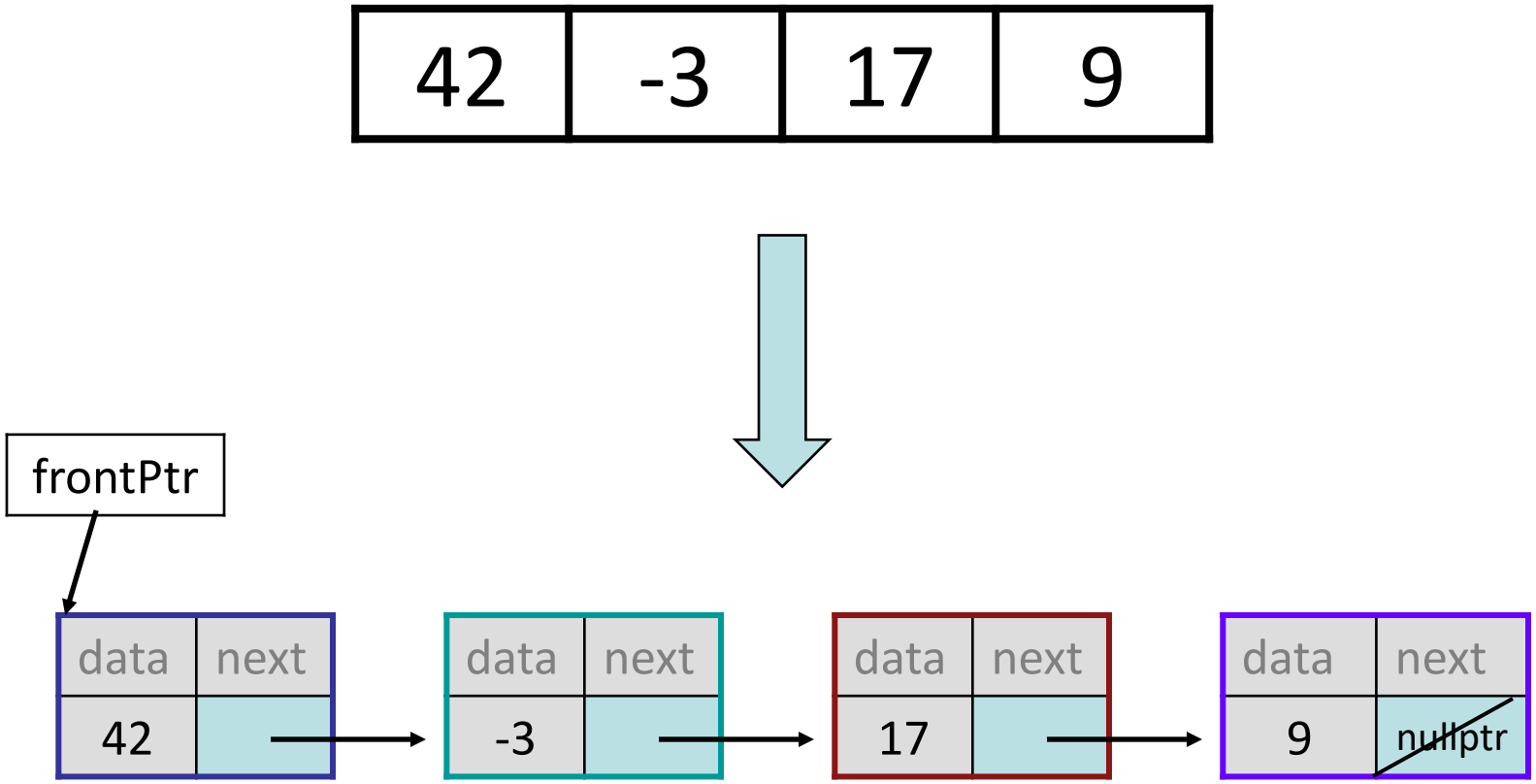
Traversing a list (12.2)

Instead, let's make another node pointer, and modify that:

```
ListNode* current = list;
while (current != nullptr) {
    cout << current->data << endl;
    current = current->next;  // move to next node
}
```



Creating a List



Creating a List

```
ListNode *vectorToLinkedList(const Vector<int>& v) {
    if (v.size() == 0) return nullptr;
    ListNode head = {v[0], nullptr};
    ListNode *currPtr = &head;
    for (int i = 1; i < v.size(); i++) {
        ListNode node = {v[i], nullptr};
        currPtr->next = &node;
        currPtr = &node;
    }
    return &head;
}
```

Creating a List

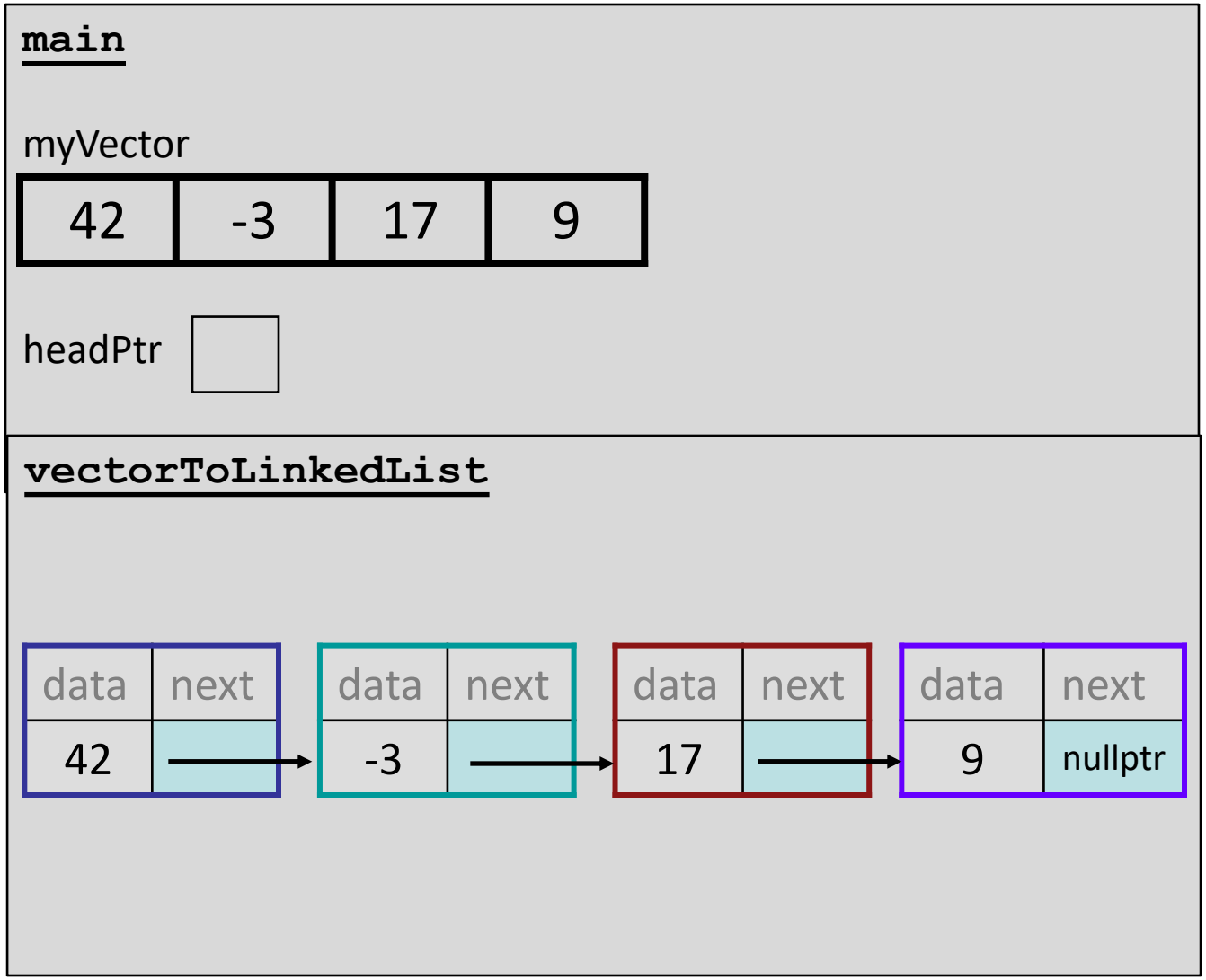
```
ListNode *vectorToLinkedList(const Vector<int>& v) {
    if (v.size() == 0) return nullptr;
    ListNode head = {v[0], nullptr};
    ListNode *currPtr = &head;
    for (int i = 1; i < v.size(); i++) {
        ListNode node = {v[i], nullptr};
        currPtr->next = &node;
        currPtr = &node;
    }
    return &head;
}
```

Problem: local variables go away when a function finishes. These Nodes will thus no longer exist, and the addresses will be for garbage memory!

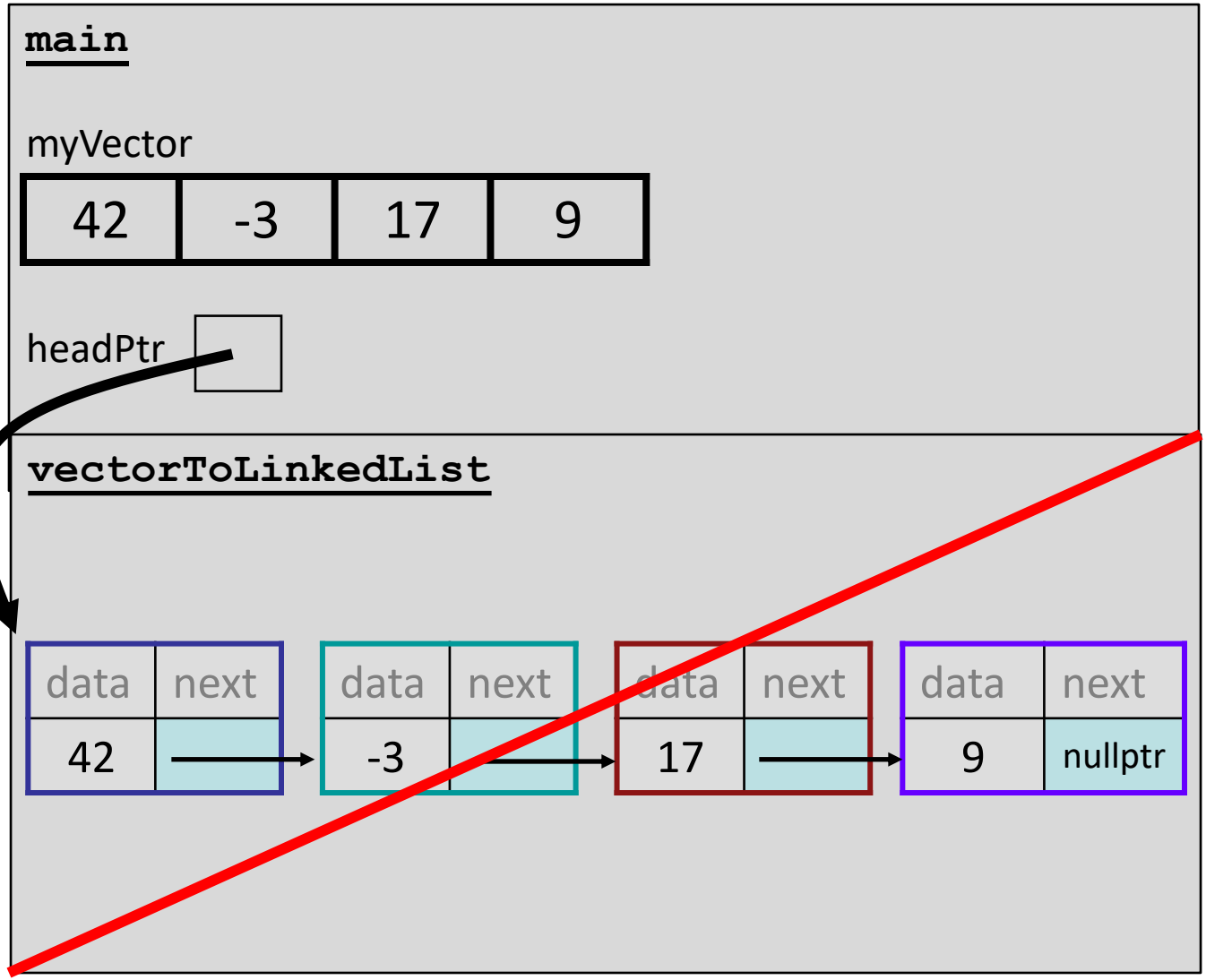
Creating a List

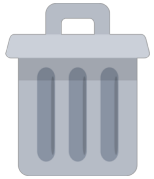
```
int main() {
    Vector<int> v = {42, -3, 17, 9};
    ListNode *headPtr = vectorToLinkedList(v);
    if (headPtr) {
        cout << headPtr->data << endl;
    }
}
```

Creating a List



Creating a List





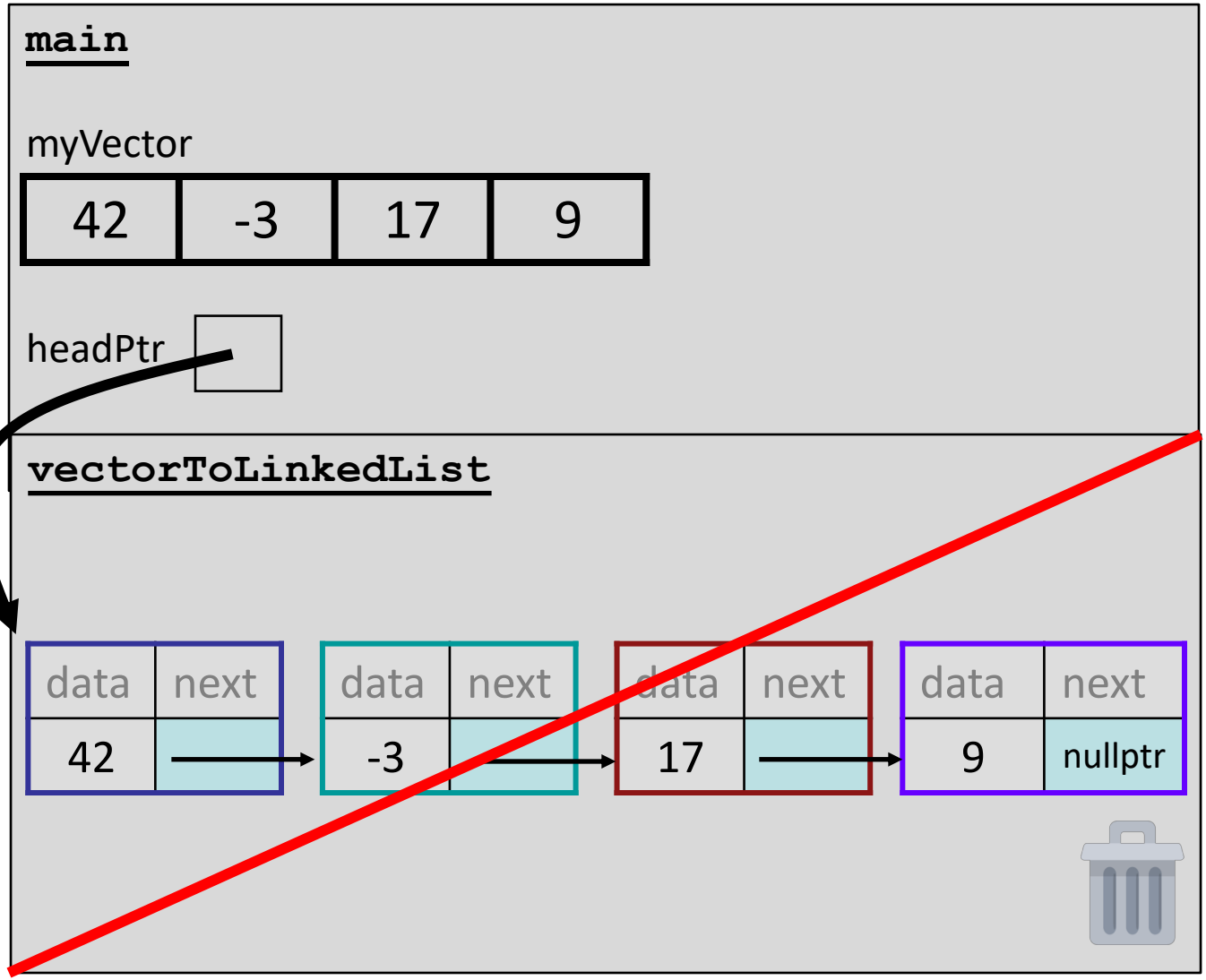
Creating a List

We need a way to have memory that doesn't get cleaned up when a function exits.

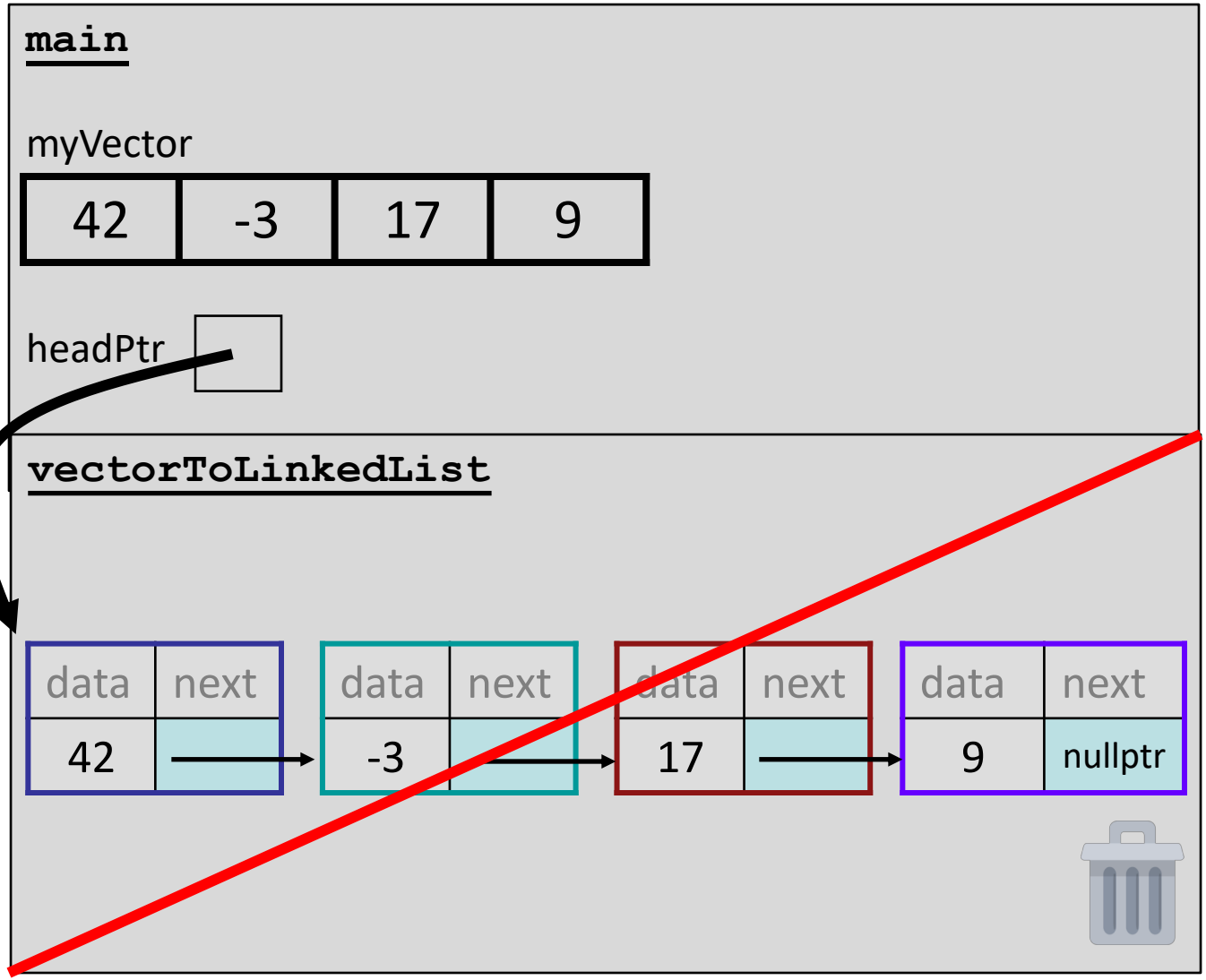
Plan For Today and Friday

- Implementing a Linked List
  - Pointers
  - Dynamic memory
  - Classes
  - Testing

A New Kind of Memory

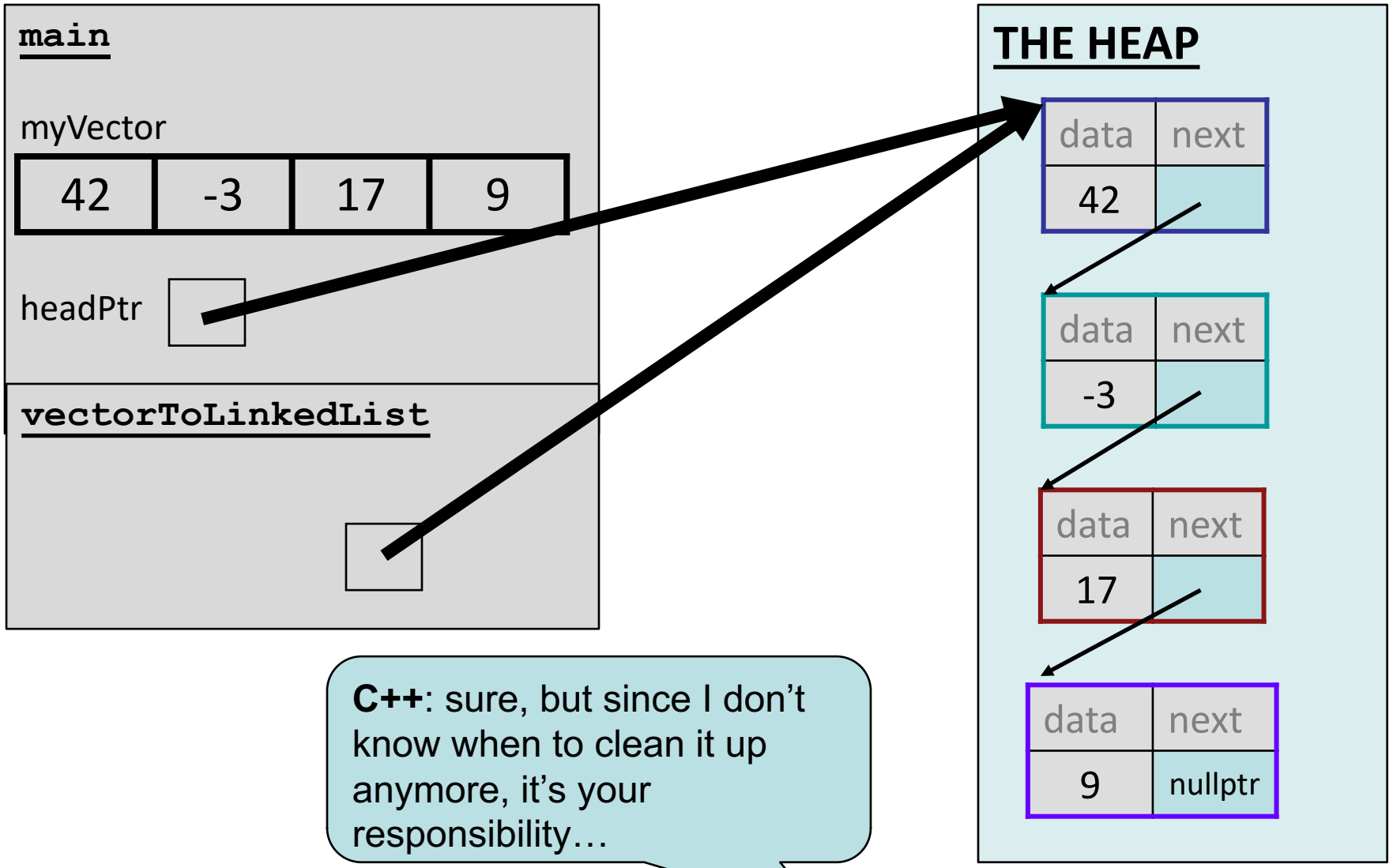


A New Kind of Memory



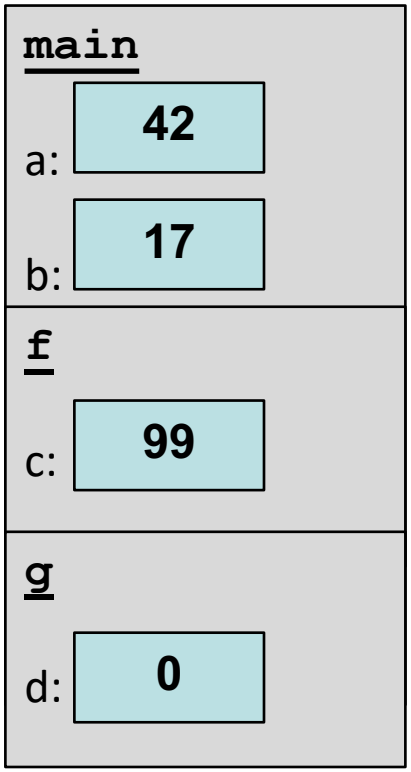
Us: hey C++, is there a way to make these variables in memory that isn't automatically cleaned up?

A New Kind of Memory



The Stack

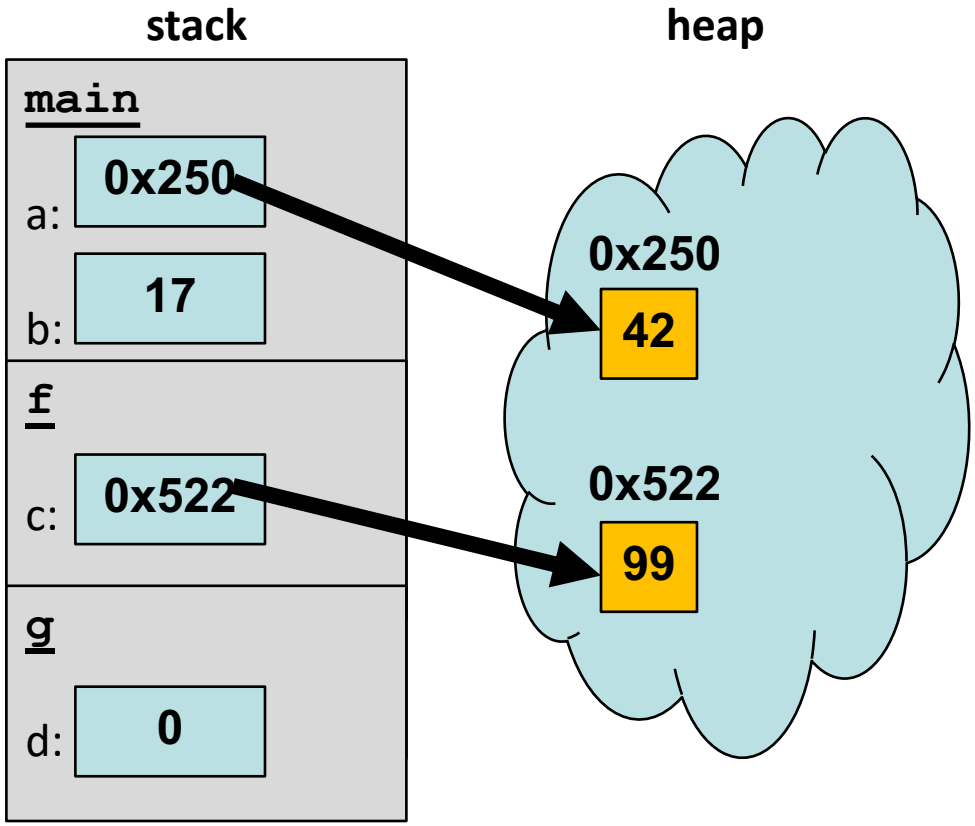
```
int main() {
    int a = 42;
    int b = 17;
    f();
}

void f() {
    int c = 99;
    g();
}

void g() {
    int d = 0;
}
```



The **stack** is the place where all local variables live. Anything you declare as a local variable in a function lives on the stack. A function's stack "frame" goes away when the function returns.

The Heap

```
int new() {
    int* a = new int(42);
    int b = 17;
    f();
}

void f() {
    int* c = new int(99);
    g();
}

void g() {
    int d = 0;
}
```

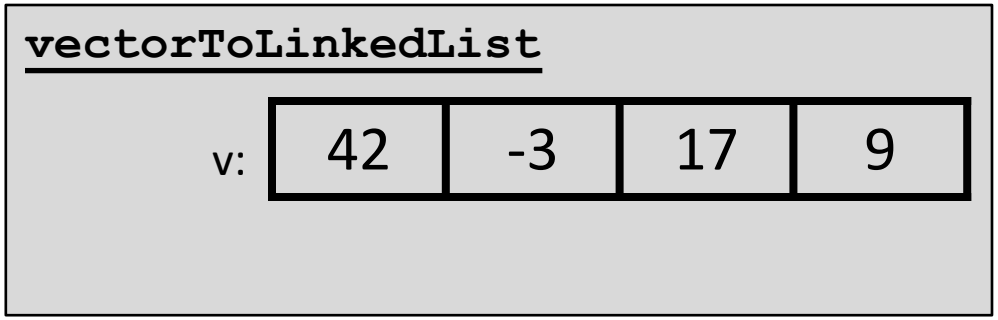


The **heap** is a part of memory that you can manage yourself. Unlike the stack, the memory only goes away when you delete it yourself. To allocate memory on the heap, use the **new** keyword. **new** returns a *the address on the heap of the new memory*.

Creating a List

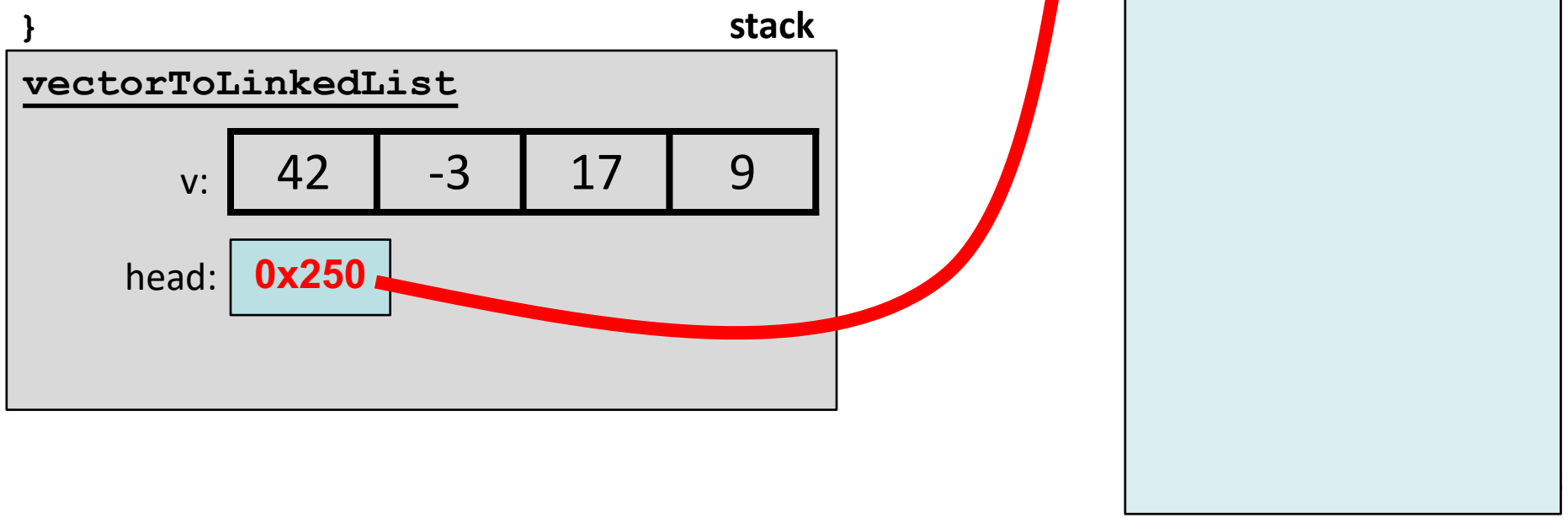
```
ListNode *vectorToLinkedList(const Vector<int>& v){
    if (v.size() == 0) return nullptr;
    ListNode *head = new Node(v[0], nullptr);
    ListNode *curr = head;
    for (int i = 1; i < v.size(); i++) {
        ListNode *node = new Node(v[i],nullptr);
        curr->next = node;
        curr = node;
    }
    return head;
}
```

stack



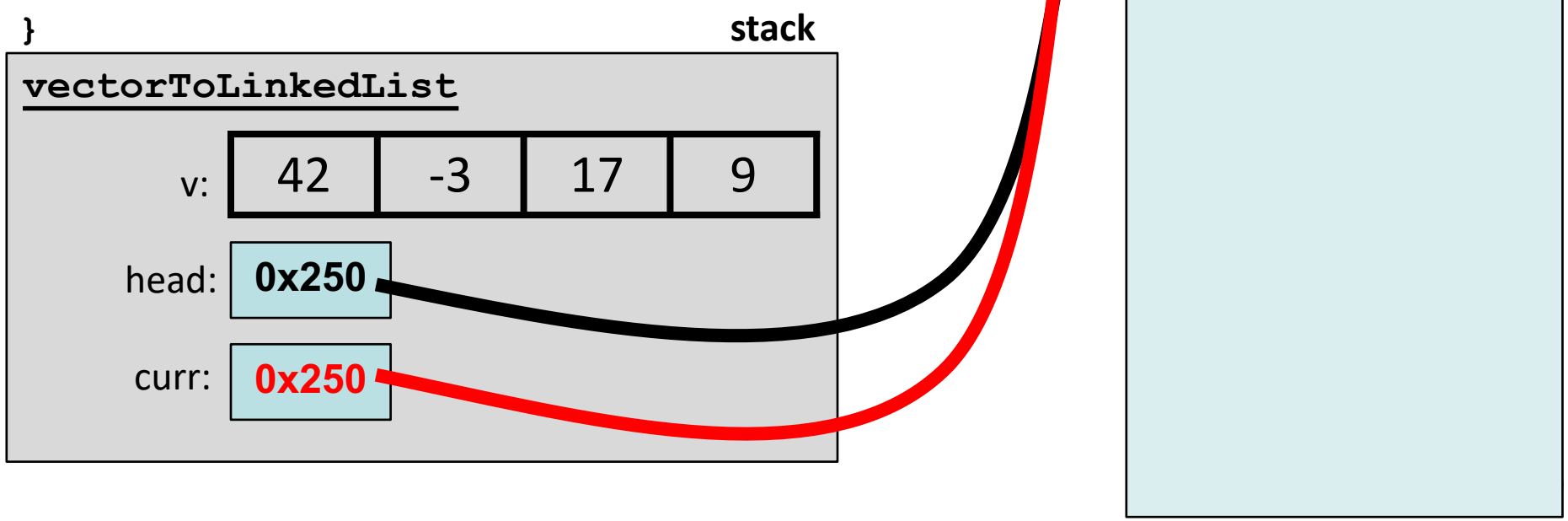
Creating a List

```
ListNode *vectorToLinkedList(const Vector<int>& v){
    if (v.size() == 0) return nullptr;
    ListNode *head = new Node(v[0], nullptr);
    ListNode *curr = head;
    for (int i = 1; i < v.size(); i++) {
        ListNode *node = new Node(v[i],nullptr);
        curr->next = node;
        curr = node;
    }
    return head;
}
```



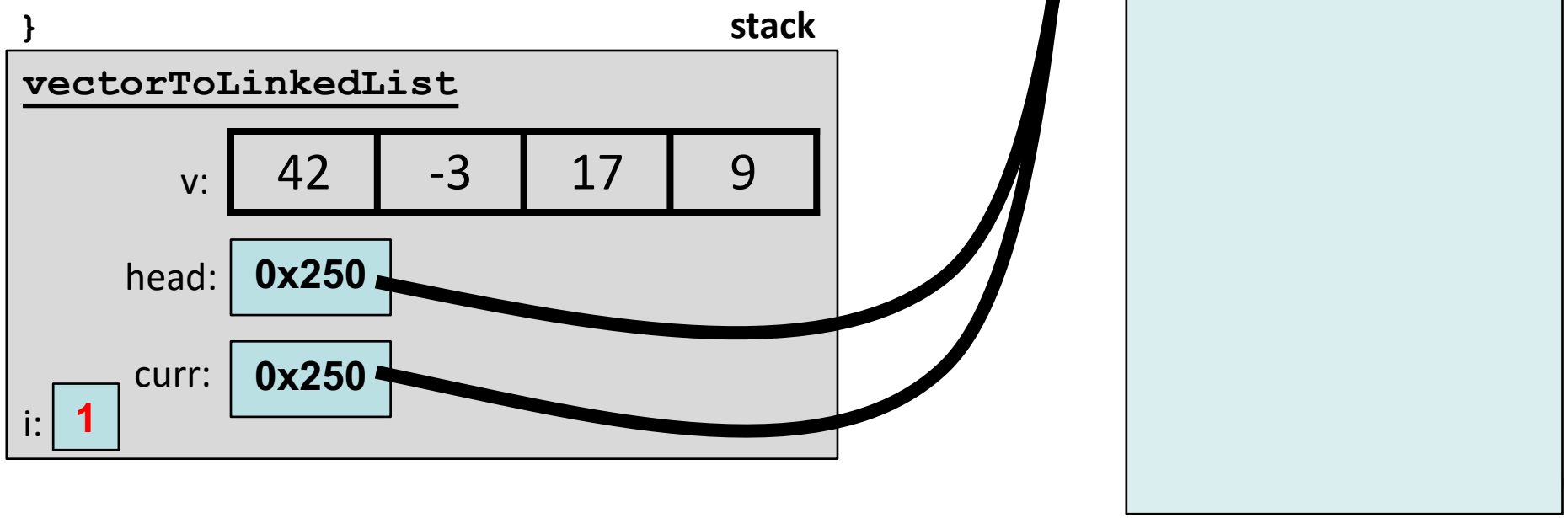
Creating a List

```
ListNode *vectorToLinkedList(const Vector<int>& v){
    if (v.size() == 0) return nullptr;
    ListNode *head = new Node(v[0], nullptr);
    ListNode *curr = head;
    for (int i = 1; i < v.size(); i++) {
        ListNode *node = new Node(v[i],nullptr);
        curr->next = node;
        curr = node;
    }
    return head;
}
```



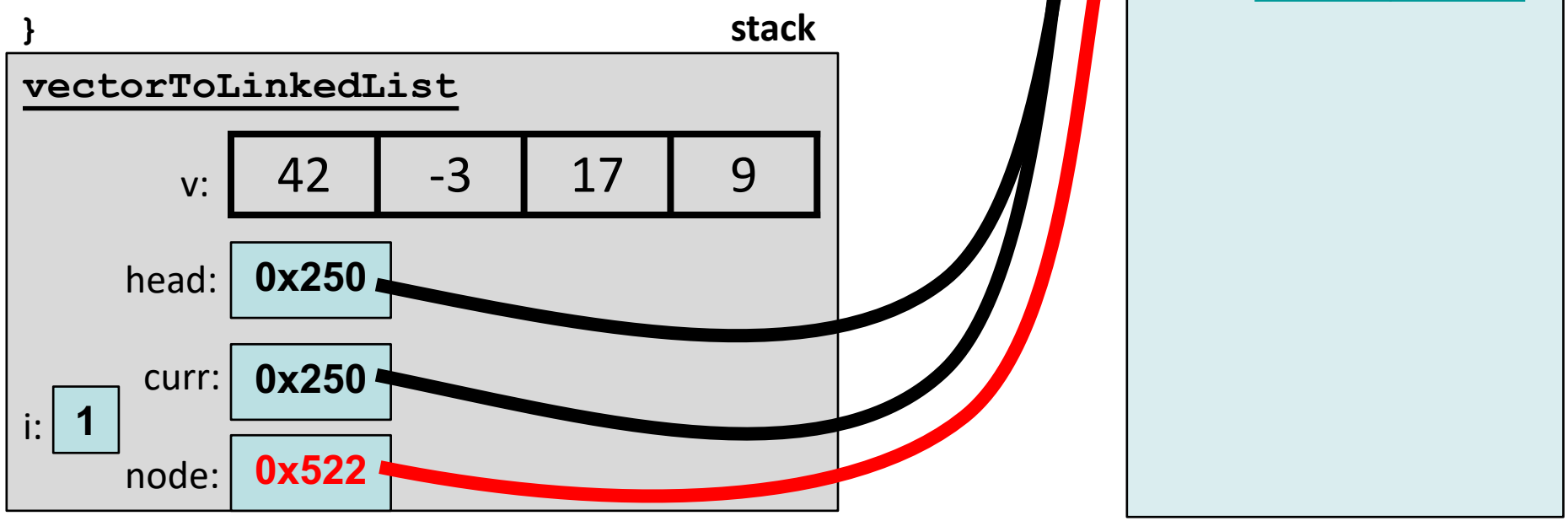
Creating a List

```
ListNode *vectorToLinkedList(const Vector<int>& v){
    if (v.size() == 0) return nullptr;
    ListNode *head = new Node(v[0], nullptr);
    ListNode *curr = head;
    for (int i = 1; i < v.size(); i++) {
        ListNode *node = new Node(v[i],nullptr);
        curr->next = node;
        curr = node;
    }
    return head;
}
```



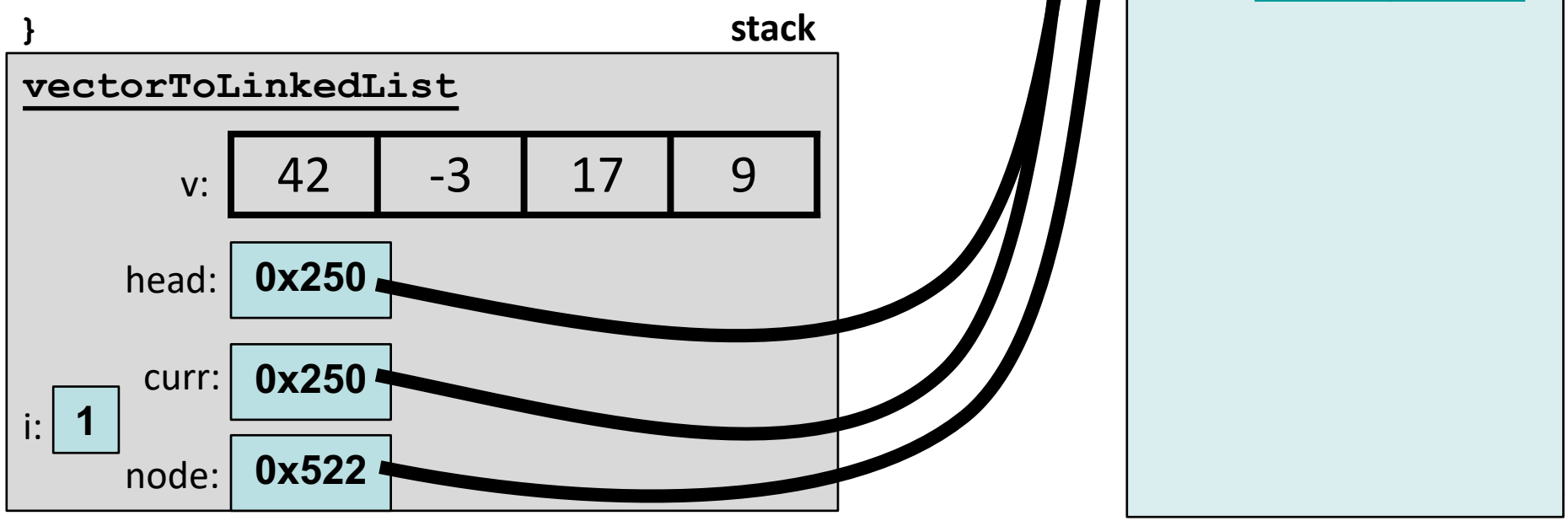
Creating a List

```
ListNode *vectorToLinkedList(const Vector<int>& v){
    if (v.size() == 0) return nullptr;
    ListNode *head = new Node(v[0], nullptr);
    ListNode *curr = head;
    for (int i = 1; i < v.size(); i++) {
        ListNode *node = new Node(v[i],nullptr);
        curr->next = node;
        curr = node;
    }
    return head;
}
```



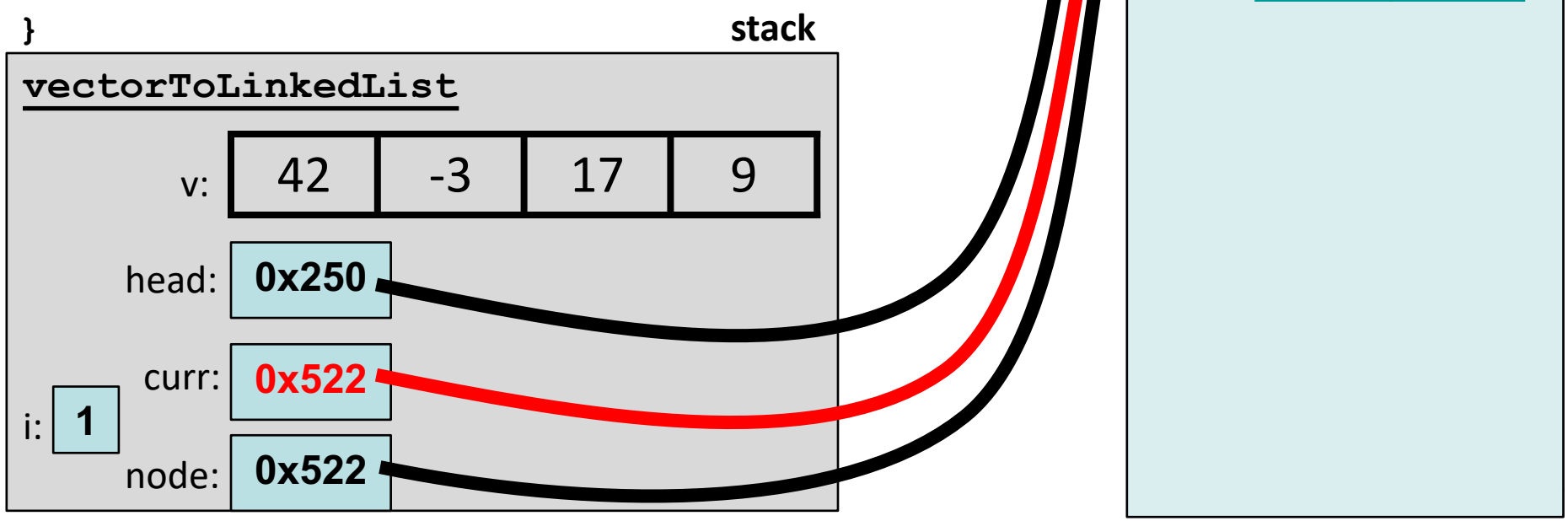
Creating a List

```
ListNode *vectorToLinkedList(const Vector<int>& v){
    if (v.size() == 0) return nullptr;
    ListNode *head = new Node(v[0], nullptr);
    ListNode *curr = head;
    for (int i = 1; i < v.size(); i++) {
        ListNode *node = new Node(v[i],nullptr);
        curr->next = node;
        curr = node;
    }
    return head;
}
```



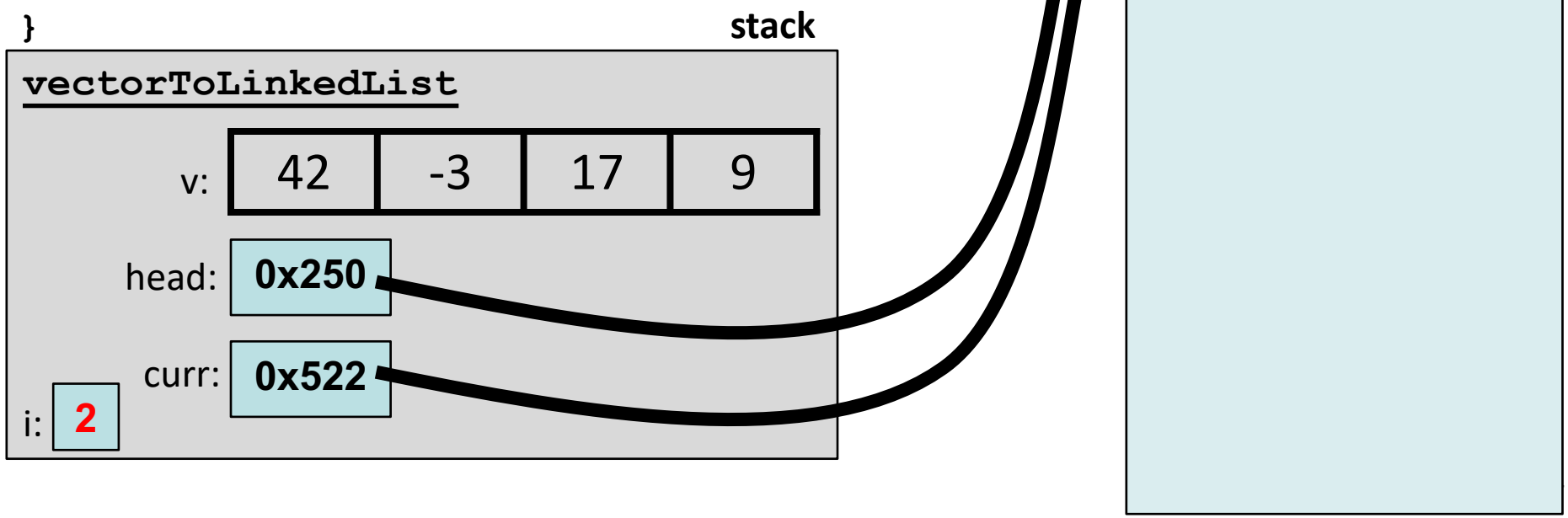
Creating a List

```
ListNode *vectorToLinkedList(const Vector<int>& v){
    if (v.size() == 0) return nullptr;
    ListNode *head = new Node(v[0], nullptr);
    ListNode *curr = head;
    for (int i = 1; i < v.size(); i++) {
        ListNode *node = new Node(v[i],nullptr);
        curr->next = node;
        curr = node;
    }
    return head;
}
```



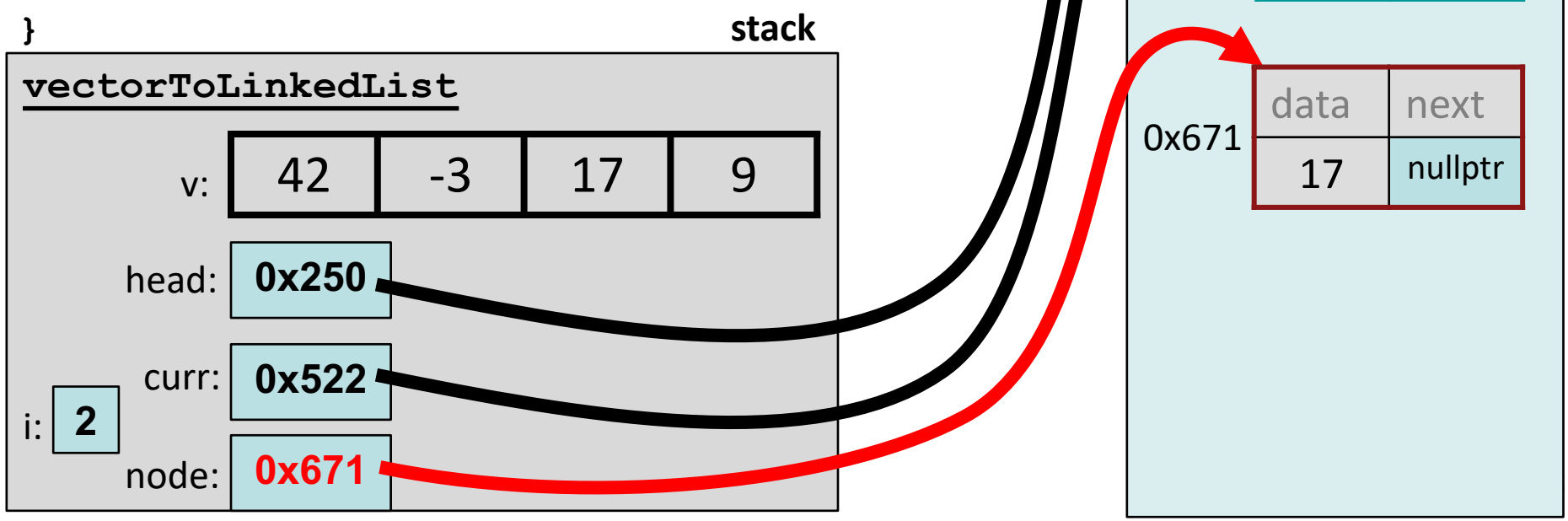
Creating a List

```
ListNode *vectorToLinkedList(const Vector<int>& v){
    if (v.size() == 0) return nullptr;
    ListNode *head = new Node(v[0], nullptr);
    ListNode *curr = head;
    for (int i = 1; i < v.size(); i++) {
        ListNode *node = new Node(v[i],nullptr);
        curr->next = node;
        curr = node;
    }
    return head;
}
```



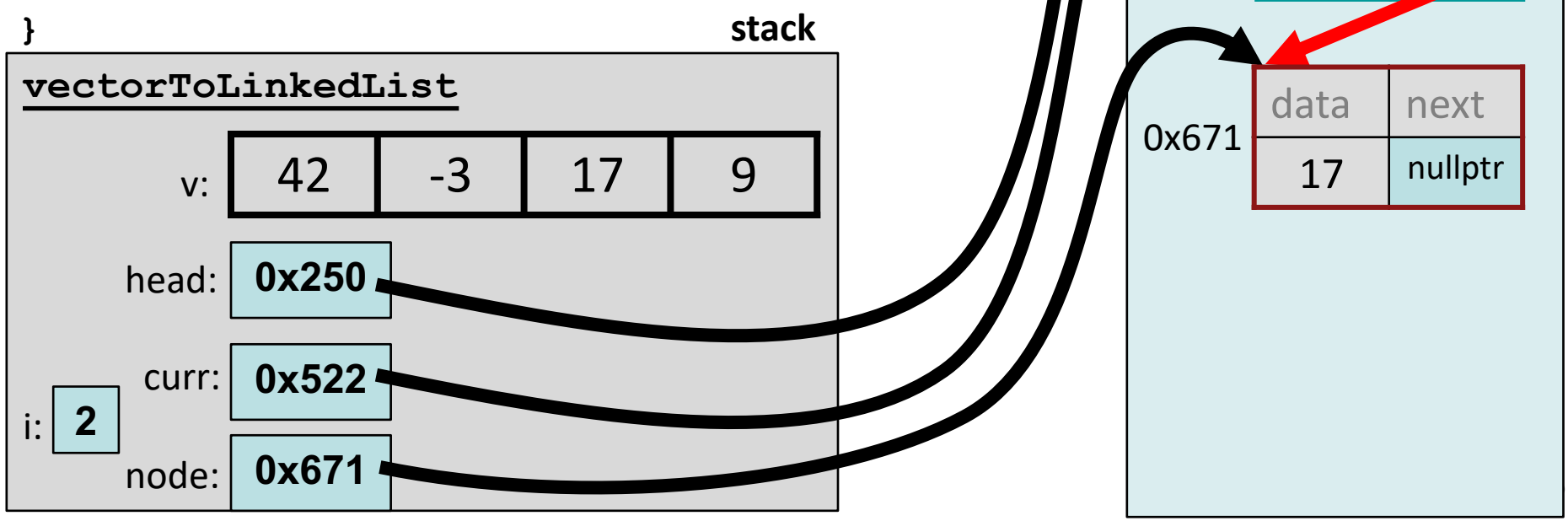
Creating a List

```
ListNode *vectorToLinkedList(const Vector<int>& v){
    if (v.size() == 0) return nullptr;
    ListNode *head = new Node(v[0], nullptr);
    ListNode *curr = head;
    for (int i = 1; i < v.size(); i++) {
        ListNode *node = new Node(v[i],nullptr);
        curr->next = node;
        curr = node;
    }
    return head;
}
```



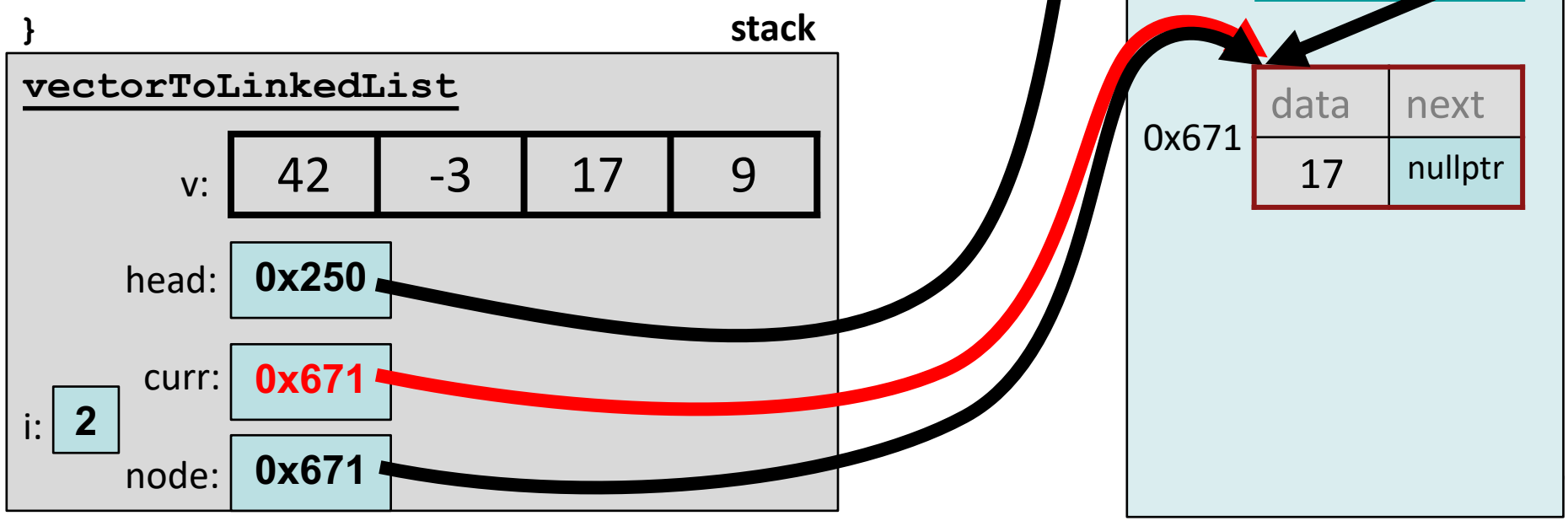
Creating a List

```
ListNode *vectorToLinkedList(const Vector<int>& v){
    if (v.size() == 0) return nullptr;
    ListNode *head = new Node(v[0], nullptr);
    ListNode *curr = head;
    for (int i = 1; i < v.size(); i++) {
        ListNode *node = new Node(v[i],nullptr);
        curr->next = node;
        curr = node;
    }
    return head;
}
```



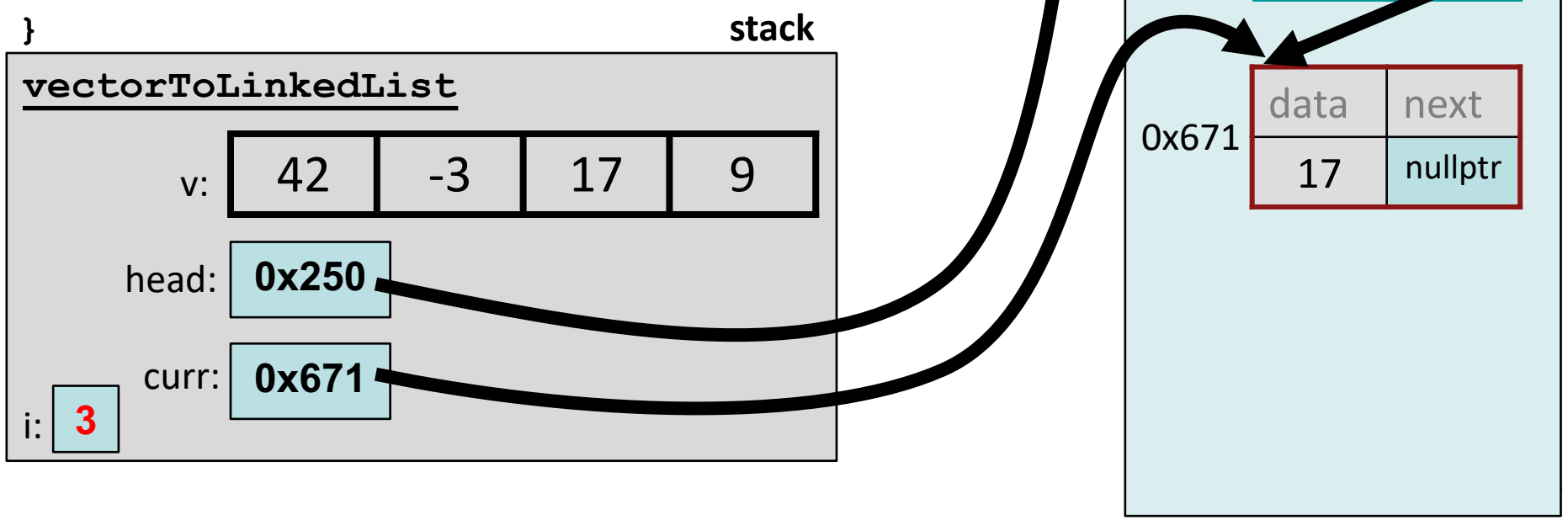
Creating a List

```
ListNode *vectorToLinkedList(const Vector<int>& v){
    if (v.size() == 0) return nullptr;
    ListNode *head = new Node(v[0], nullptr);
    ListNode *curr = head;
    for (int i = 1; i < v.size(); i++) {
        ListNode *node = new Node(v[i],nullptr);
        curr->next = node;
        curr = node;
    }
    return head;
}
```



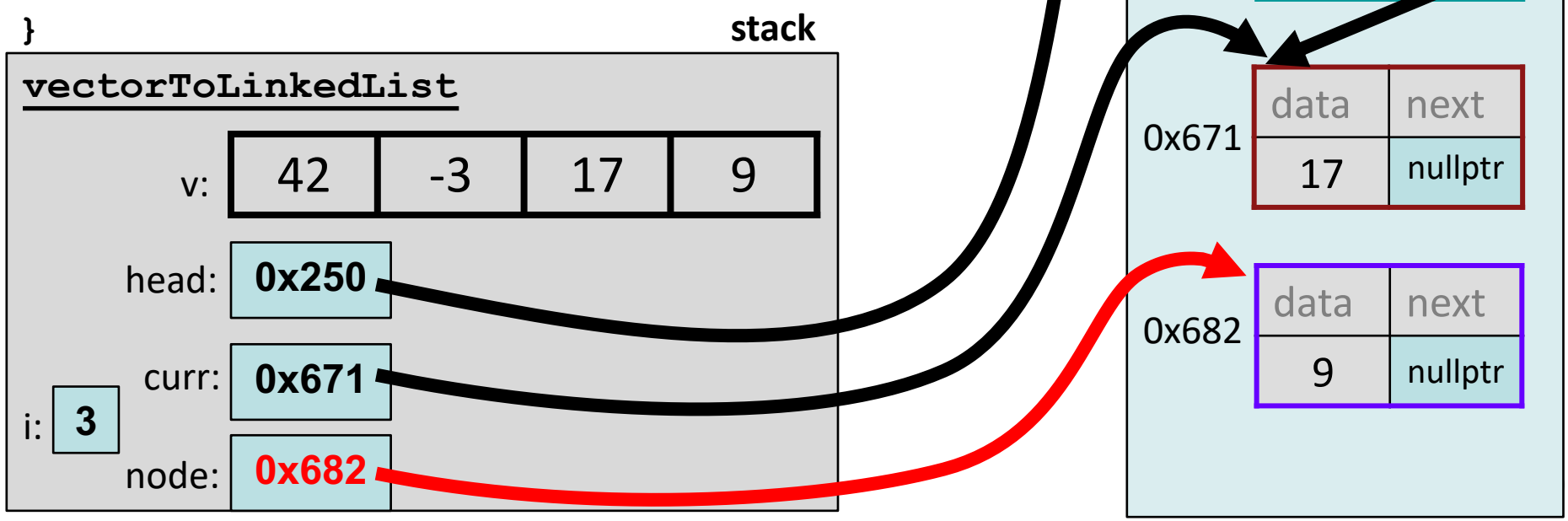
Creating a List

```
ListNode *vectorToLinkedList(const Vector<int>& v){
    if (v.size() == 0) return nullptr;
    ListNode *head = new Node(v[0], nullptr);
    ListNode *curr = head;
    for (int i = 1; i < v.size(); i++) {
        ListNode *node = new Node(v[i],nullptr);
        curr->next = node;
        curr = node;
    }
    return head;
}
```



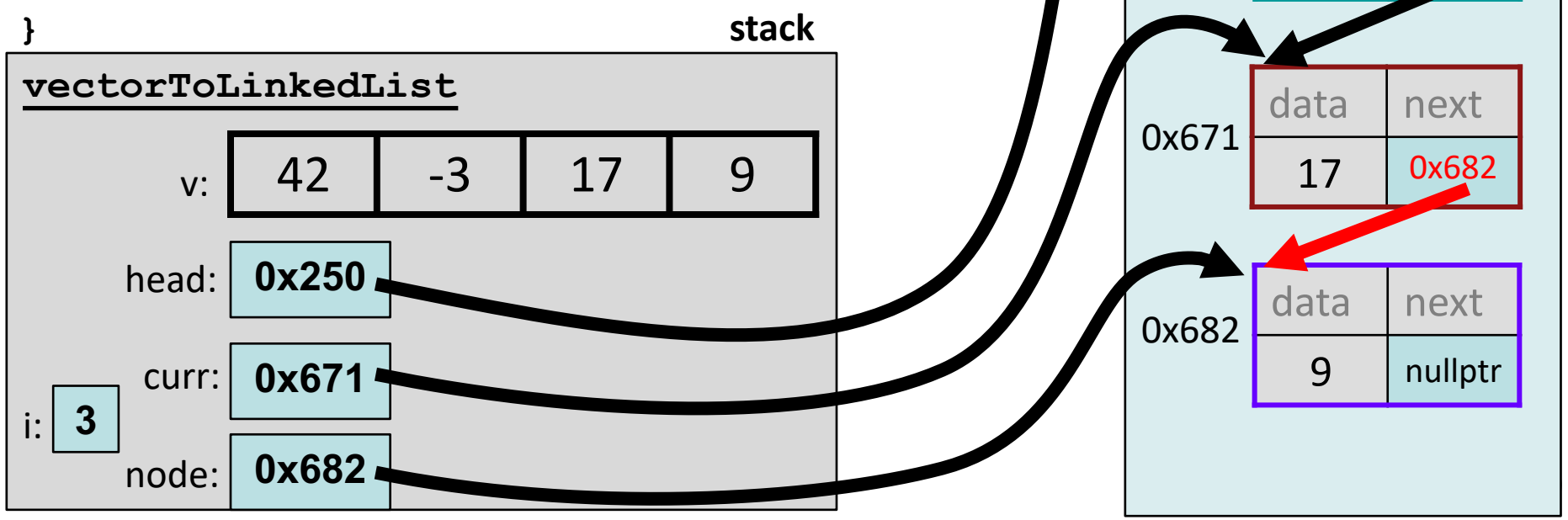
Creating a List

```
ListNode *vectorToLinkedList(const Vector<int>& v){
    if (v.size() == 0) return nullptr;
    ListNode *head = new Node(v[0], nullptr);
    ListNode *curr = head;
    for (int i = 1; i < v.size(); i++) {
        ListNode *node = new Node(v[i],nullptr);
        curr->next = node;
        curr = node;
    }
    return head;
}
```



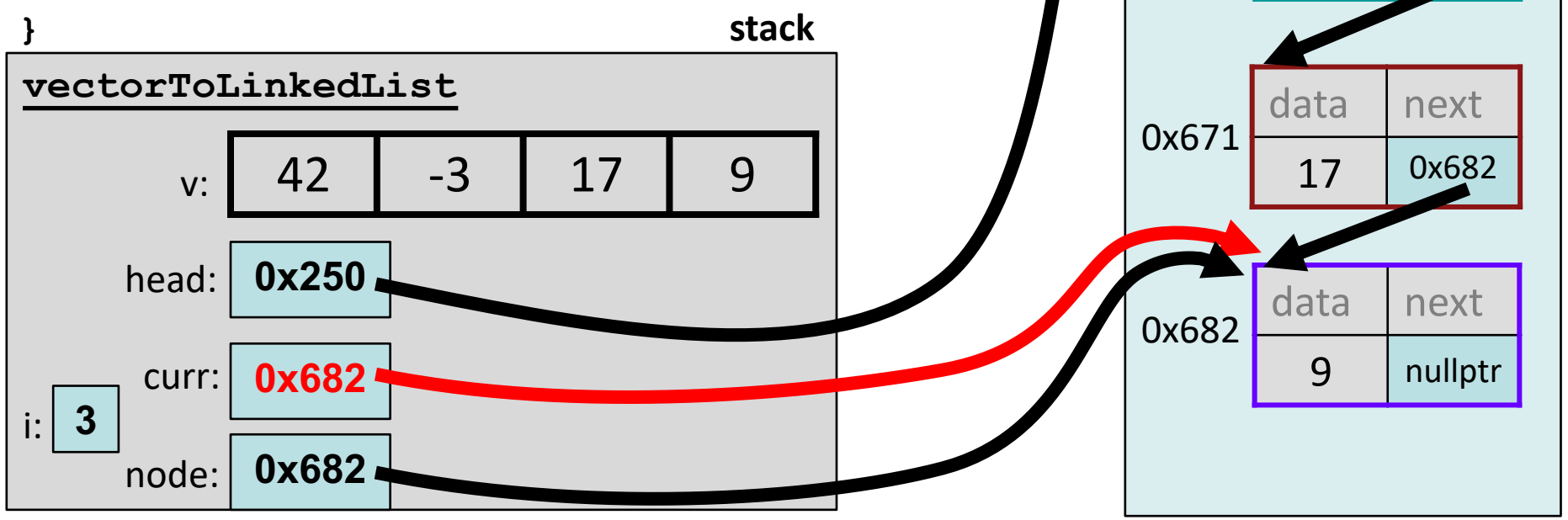
Creating a List

```
ListNode *vectorToLinkedList(const Vector<int>& v){
    if (v.size() == 0) return nullptr;
    ListNode *head = new Node(v[0], nullptr);
    ListNode *curr = head;
    for (int i = 1; i < v.size(); i++) {
        ListNode *node = new Node(v[i],nullptr);
        curr->next = node;
        curr = node;
    }
    return head;
}
```



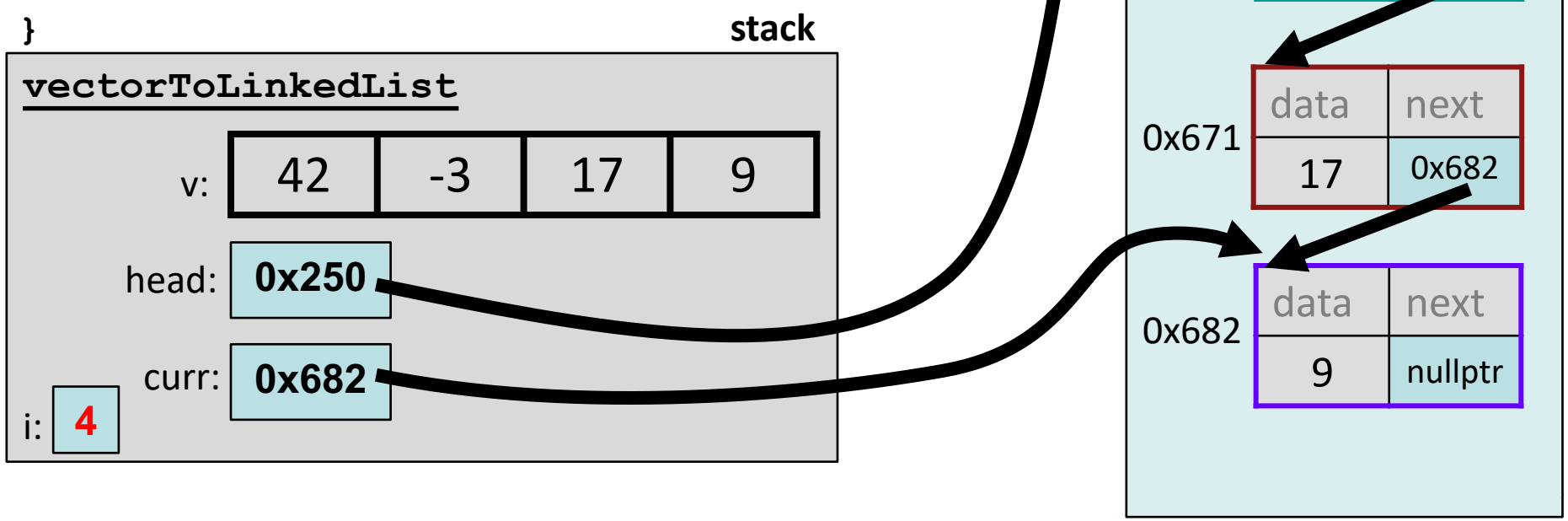
Creating a List

```
ListNode *vectorToLinkedList(const Vector<int>& v){
    if (v.size() == 0) return nullptr;
    ListNode *head = new Node(v[0], nullptr);
    ListNode *curr = head;
    for (int i = 1; i < v.size(); i++) {
        ListNode *node = new Node(v[i],nullptr);
        curr->next = node;
        curr = node;
    }
    return head;
}
```



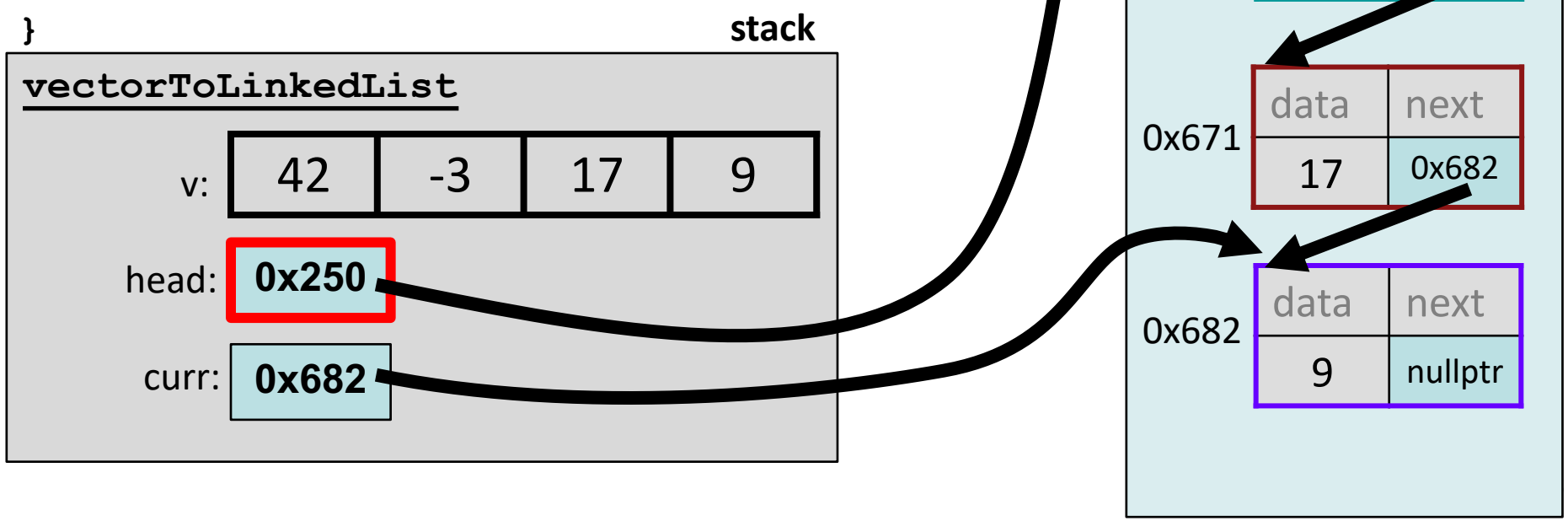
Creating a List

```
ListNode *vectorToLinkedList(const Vector<int>& v){
    if (v.size() == 0) return nullptr;
    ListNode *head = new Node(v[0], nullptr);
    ListNode *curr = head;
    for (int i = 1; i < v.size(); i++) {
        ListNode *node = new Node(v[i],nullptr);
        curr->next = node;
        curr = node;
    }
    return head;
}
```



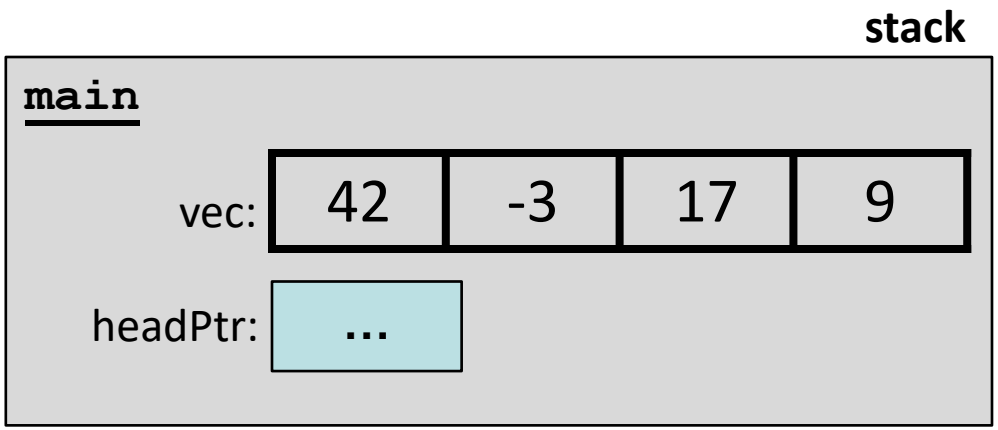
Creating a List

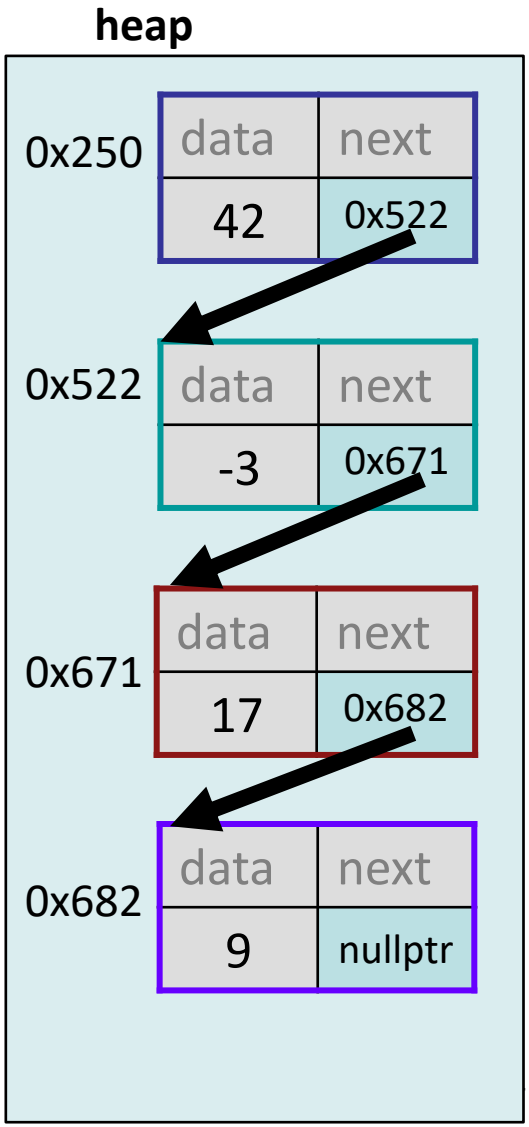
```
ListNode *vectorToLinkedList(const Vector<int>& v){
    if (v.size() == 0) return nullptr;
    ListNode *head = new Node(v[0], nullptr);
    ListNode *curr = head;
    for (int i = 1; i < v.size(); i++) {
        ListNode *node = new Node(v[i],nullptr);
        curr->next = node;
        curr = node;
    }
    return head;
}
```



Creating a List

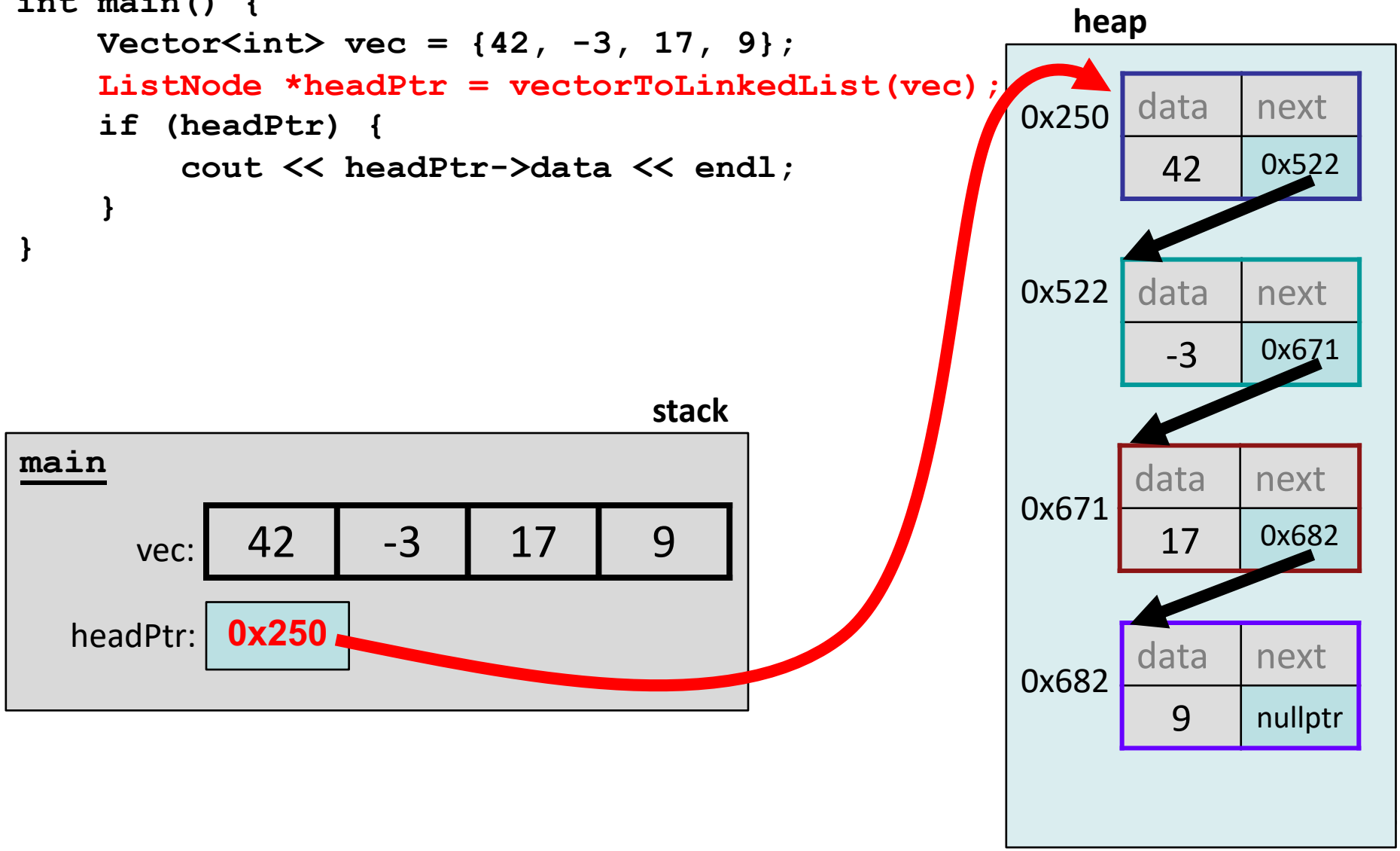
```
int main() {
    Vector<int> vec = {42, -3, 17, 9};
    ListNode *headPtr = vectorToLinkedList(vec);
    if (headPtr) {
        cout << headPtr->data << endl;
    }
}
```





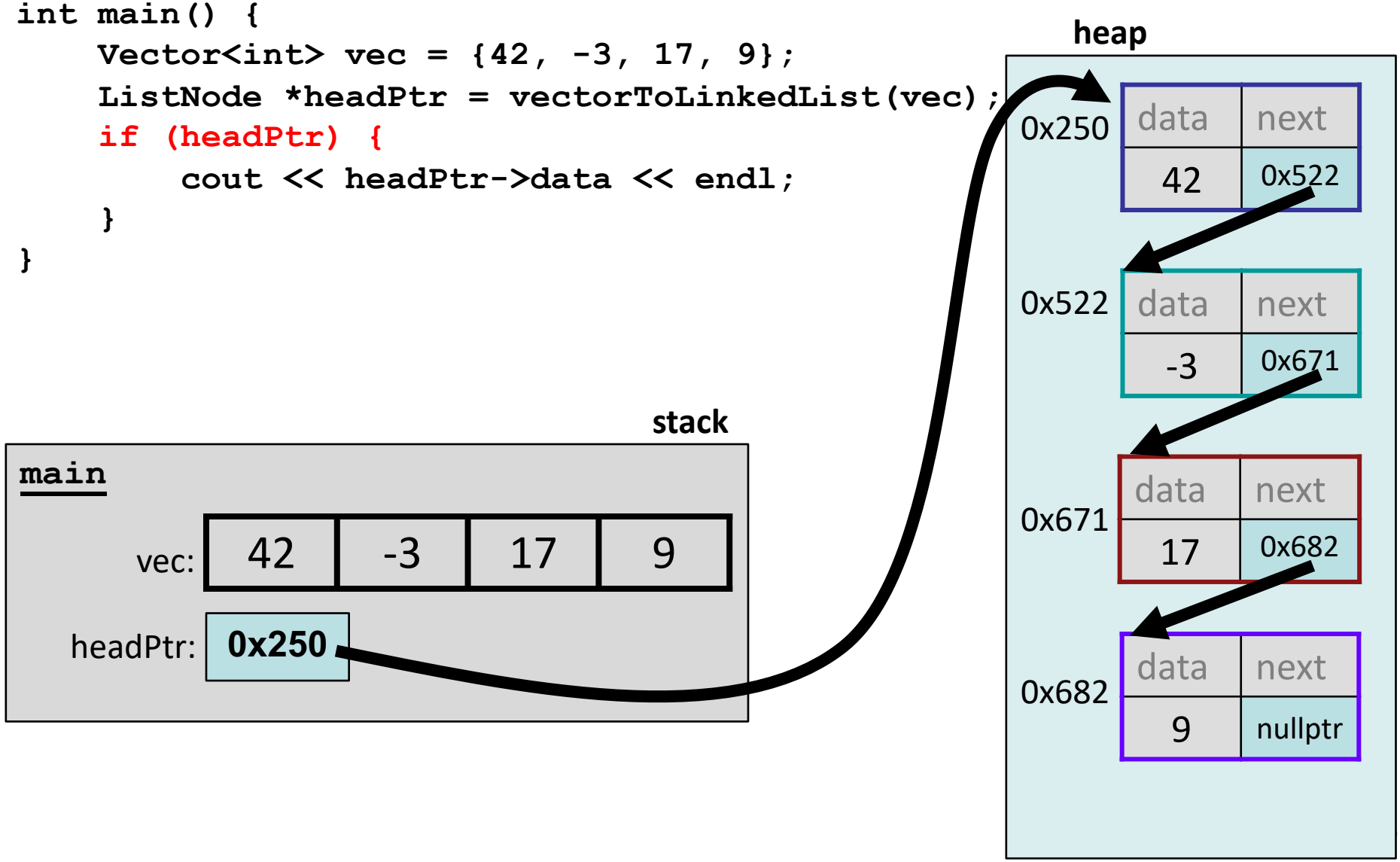
Creating a List

```
int main() {
    Vector<int> vec = {42, -3, 17, 9};
    ListNode *headPtr = vectorToLinkedList(vec);
    if (headPtr) {
        cout << headPtr->data << endl;
    }
}
```



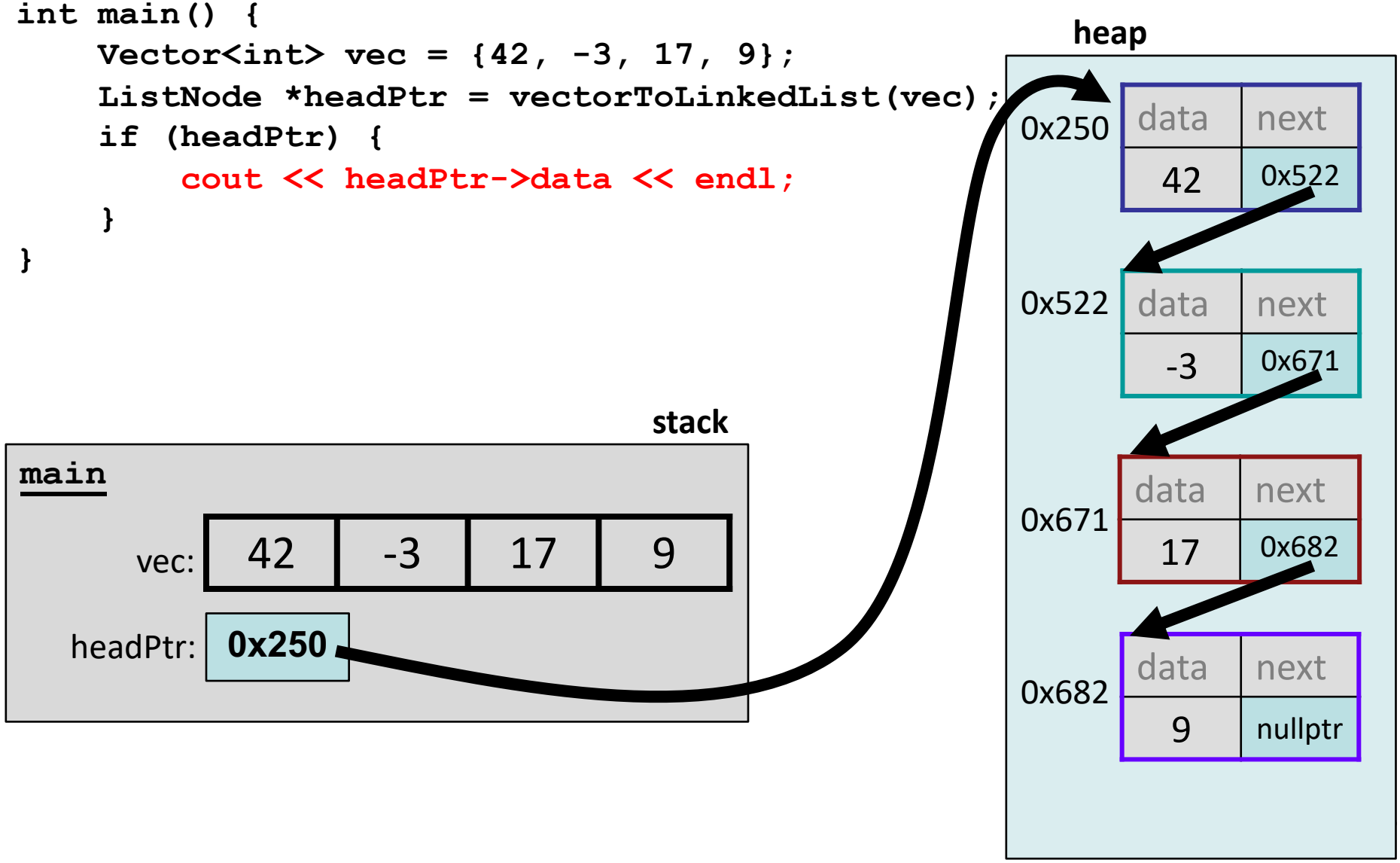
Creating a List

```
int main() {
    Vector<int> vec = {42, -3, 17, 9};
    ListNode *headPtr = vectorToLinkedList(vec);
    if (headPtr) {
        cout << headPtr->data << endl;
    }
}
```



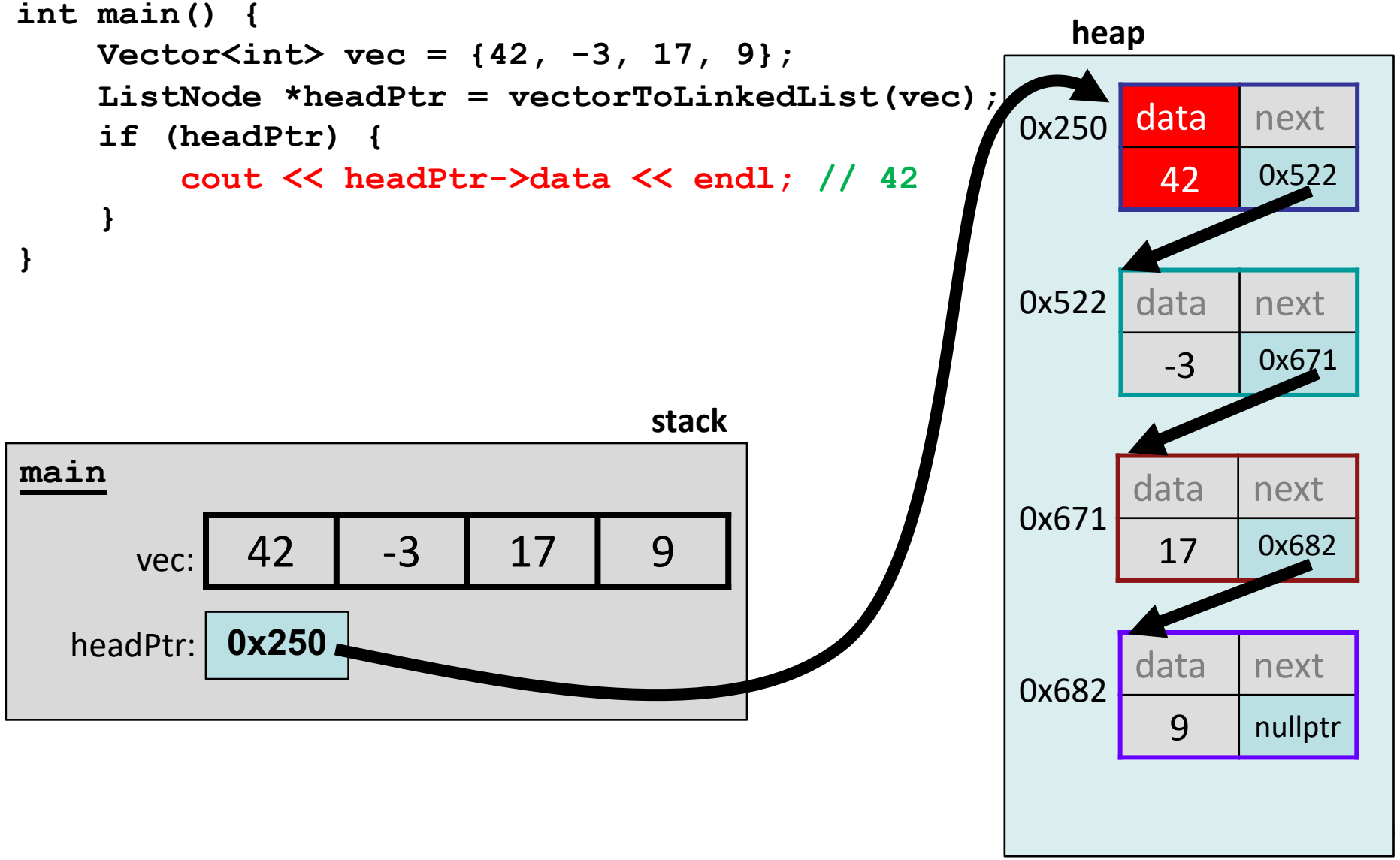
Creating a List

```
int main() {
    Vector<int> vec = {42, -3, 17, 9};
    ListNode *headPtr = vectorToLinkedList(vec);
    if (headPtr) {
        cout << headPtr->data << endl;
    }
}
```



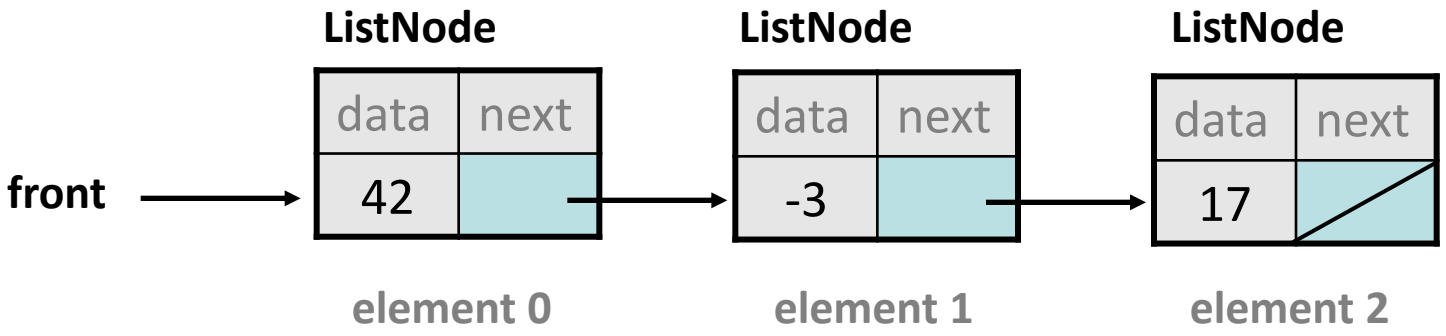
Creating a List

```
int main() {
    Vector<int> vec = {42, -3, 17, 9};
    ListNode *headPtr = vectorToLinkedList(vec);
    if (headPtr) {
        cout << headPtr->data << endl; // 42
    }
}
```



Linked list operations

- Let's write several common linked list operations:
  - `size()`
  - `print()`
  - `get(`***`index`***`)`
  - `add(`***`value`***`)`
  - `remove(`***`index`***`)`
  - `insert(`***`index`***`, `***`value`***`)`

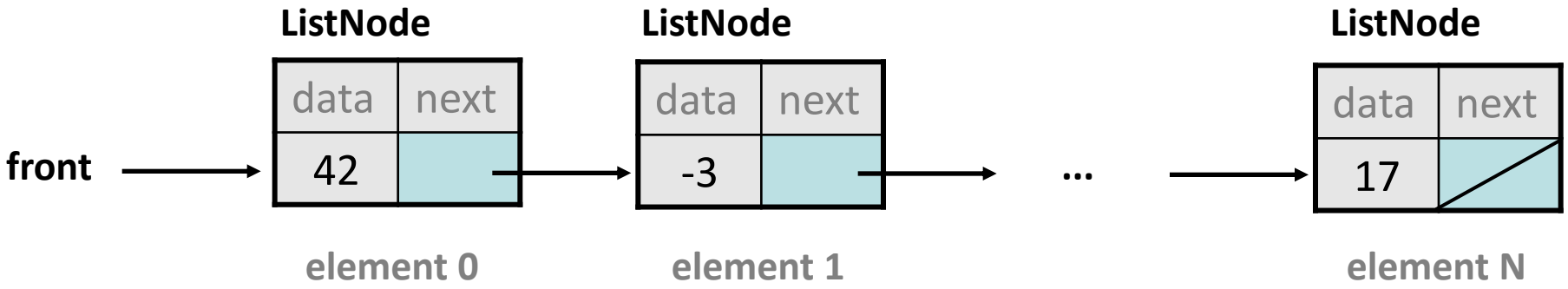


size

Implement size

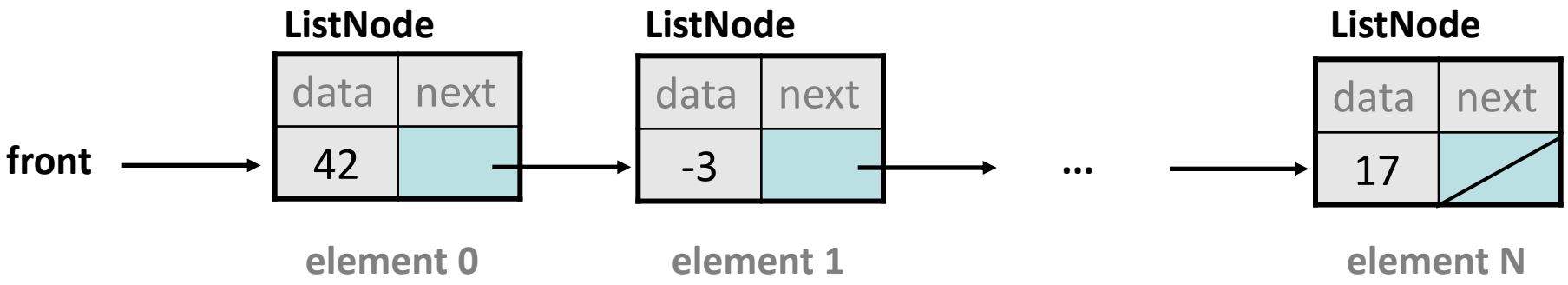
```
// Returns the length of the list.
int size(ListNode* front) {
    ...
}
```

– What different cases must we consider?



Implement size

```
// Returns the length of the list.
int size(ListNode* front) {
    ListNode* curr = front;
    int count = 0;
    while (curr != nullptr) {
        count++;
        curr = curr->next;
    }
    return count;
}
```

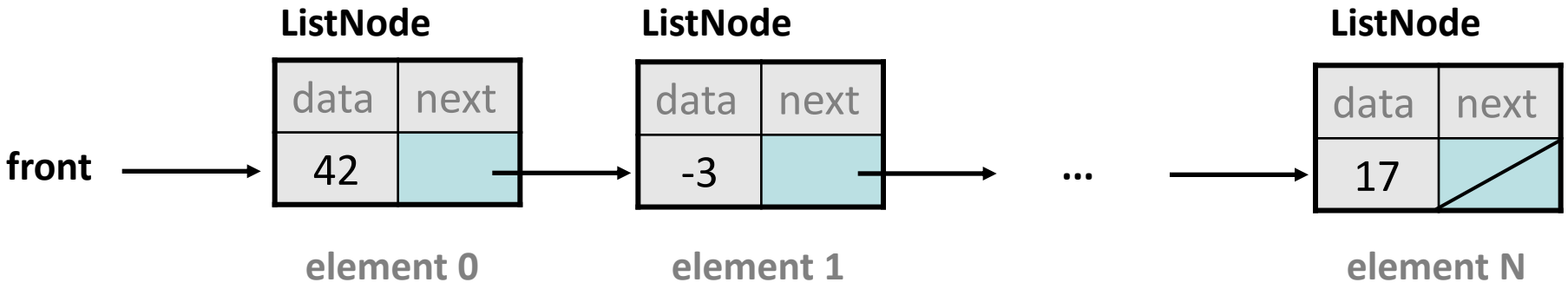


print

Implement print

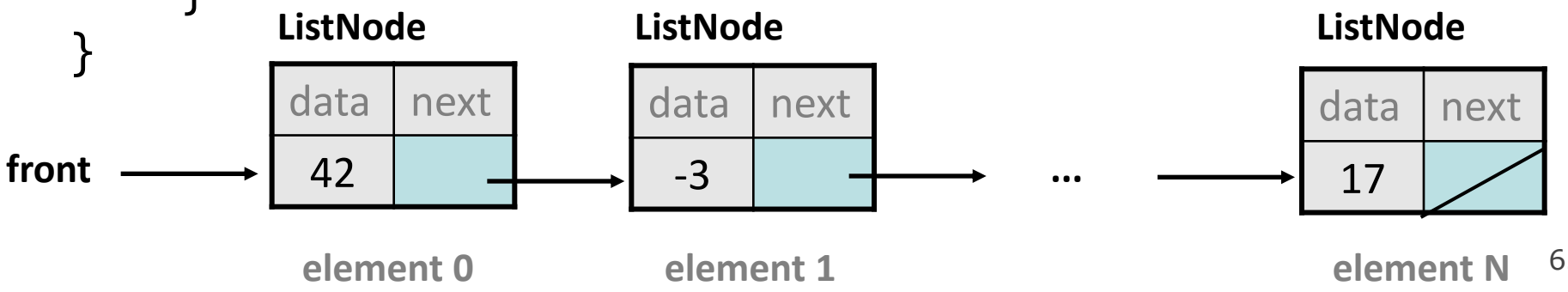
```
// Prints out the list on one line.
void print(ListNode* front) {
    ...
}
```

– What different cases must we consider?



Implement print

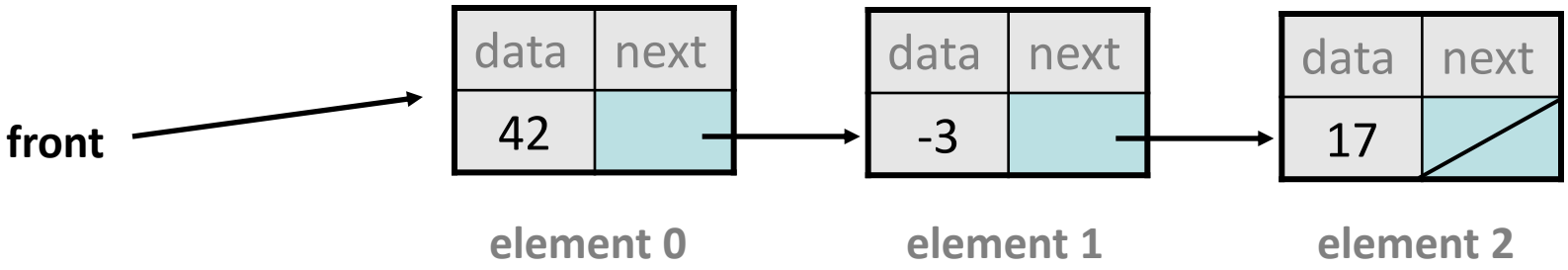
```
// Prints out the list on one line.
void print(ListNode* front) {
    if (front == nullptr) {
        cout << "(empty)" << endl;
    } else {
        ListNode* current = front;
        while (current != nullptr) {
            cout << current->data << " ";
            current = current->next;
        }
        cout << endl;
    }
}
```



get

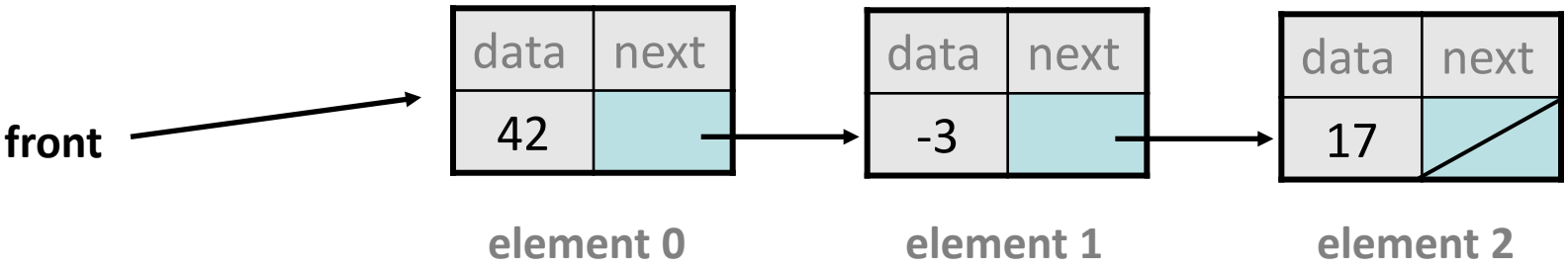
Implementing get

```
// Returns value in list at given index.
int get(ListNode* front, int index) {
    ...
}
```



Implementing get

```
// Returns value in list at given index.
int get(ListNode* front, int index) {
    ListNode* curr = front;
    for (int i = 0; i < index; i++) {
        curr = curr->next;
    }
    return curr->data;
}
```

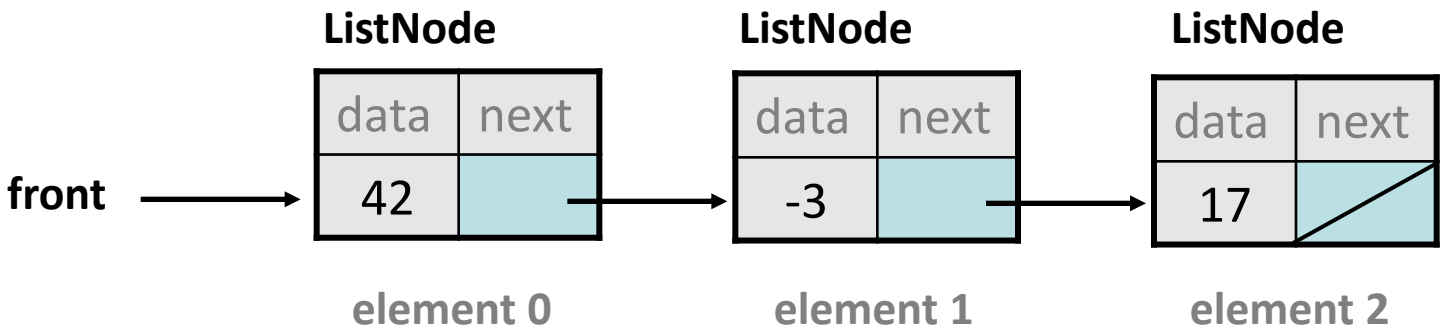


add

Implementing add

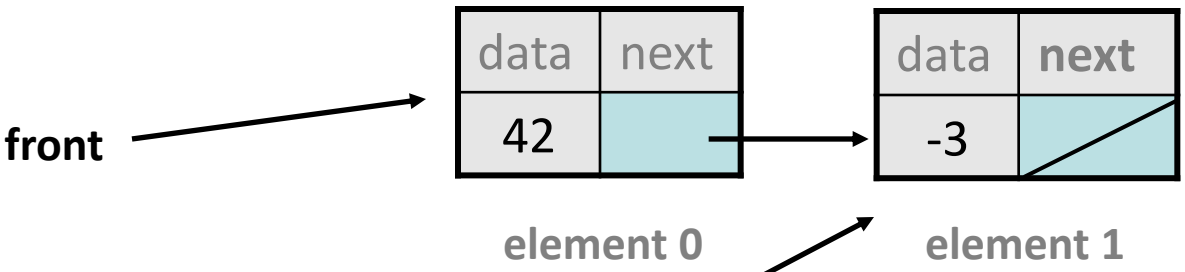
```
// Appends the given value to the end of the list.
void add(front, value) {
    ...
}
```

– What pointer(s) must be changed to add a node to the end of a list?
– What different cases must we consider?



Don't fall off the edge!

- Must modify the `next` pointer of the last node.



  - Where should `current` be pointing, to add 20 at the end?
    - `current` _____

Q: What loop test will stop us at this place in the list?

A. `while (current != nullptr) { ...`

B. `while (front != nullptr) { ...`

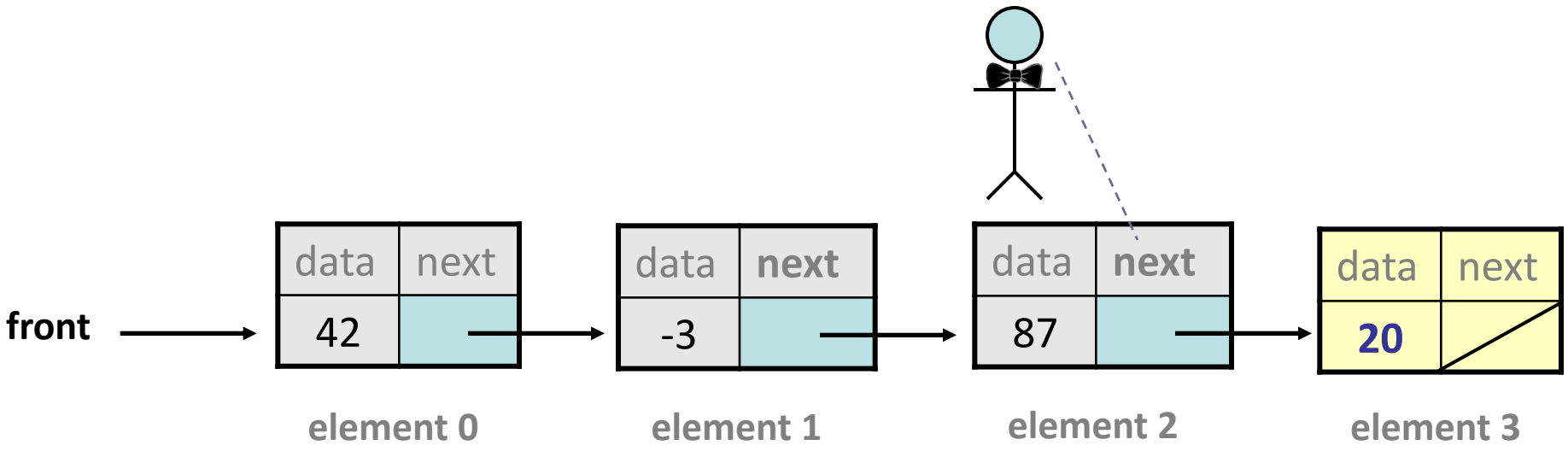
C. `while (current->next != nullptr) { ...`

D. `while (front->next != nullptr) { ...`

"James Bond" analogy

- James Bond is standing on the train cars and must attach/remove the crucial car by standing on the previous car.

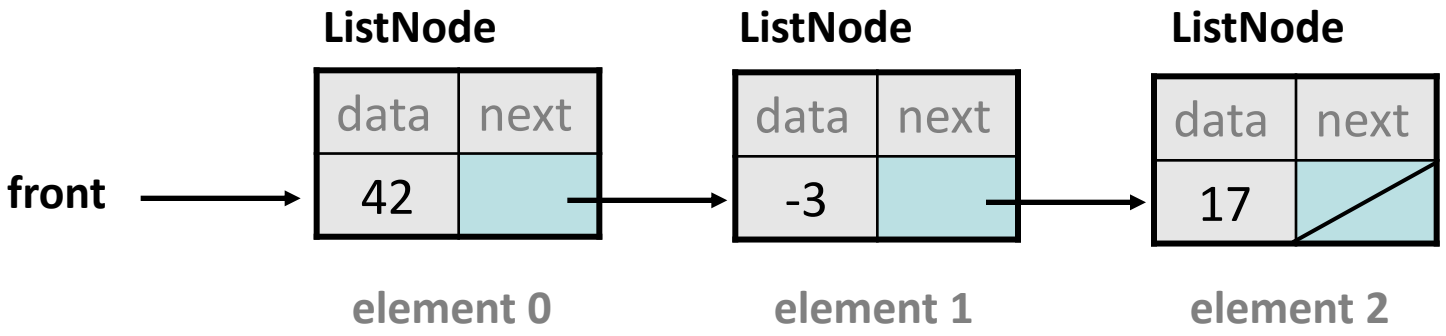
```
add(front, 20);
```



Reference to pointer

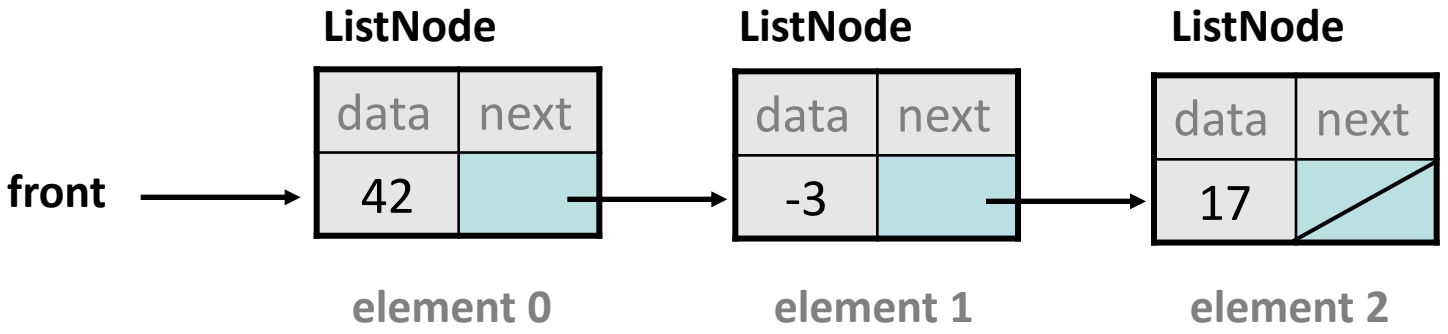
```
void functionName(ListNode*& front, parameters) {
    ...
}
```

- In our linked list operations, we will use the unusual syntax of passing a reference to a pointer as a parameter.
- This is so that the pointer is shared between main and our function, so that if we change the pointer, the change is seen in main as well.



Implementing add

```
void add(ListNode*& front, int value) {
    ListNode* newNode = new ListNode(value);

    if (front == nullptr) {
        front = newNode;
    } else {
        ListNode* current = front;
        while (current->next != nullptr) {
            current = current->next;
        }
        current->next = newNode;
    }
}
```



remove

Implementing remove

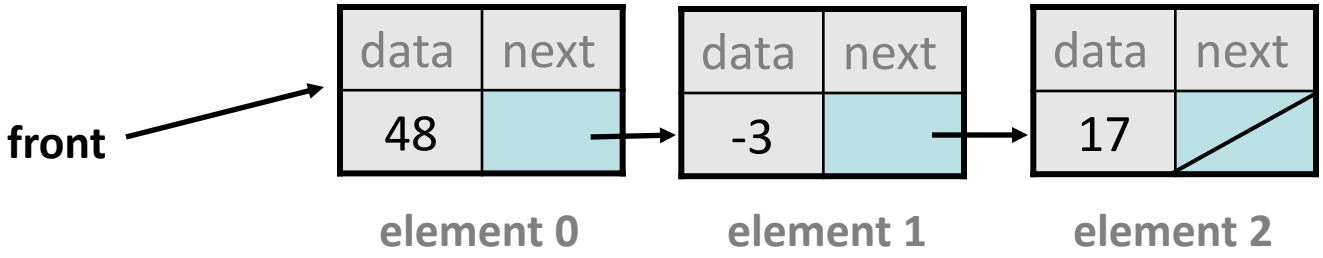
```
// Removes value at given index from list.
void remove(ListNode*& front, int index) {
    ...
}
```

– What pointer(s) must be changed to remove a node from a list?
– What different cases must we consider?

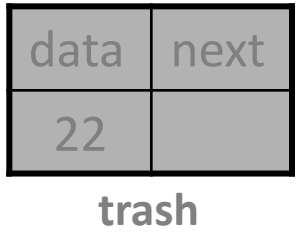
Removing from a list

- Before removing element at index 2:

- After:

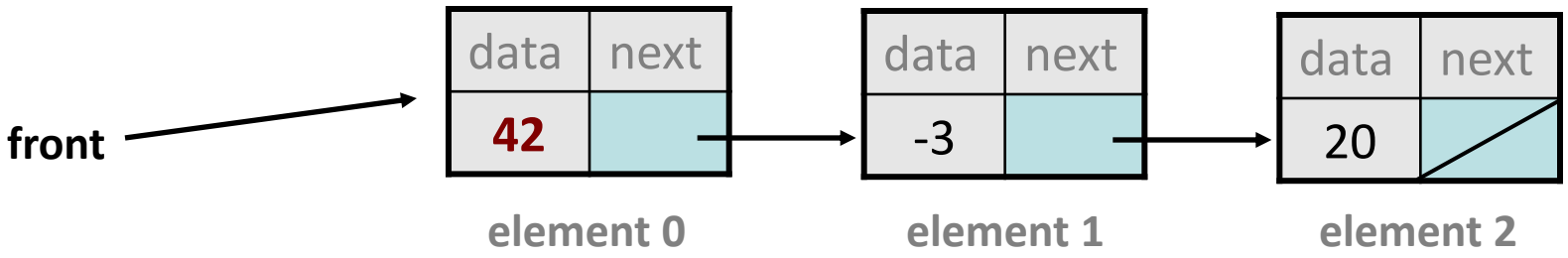


- Where should `current` be pointing?
  How many times should it advance from `front`?

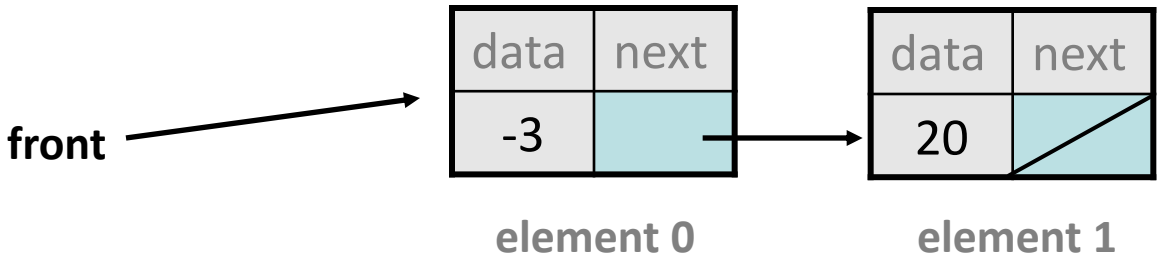


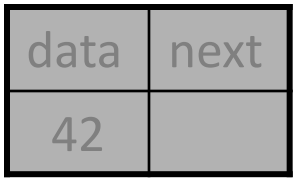
Removing from front

- Before removing element at index 0:



- After:

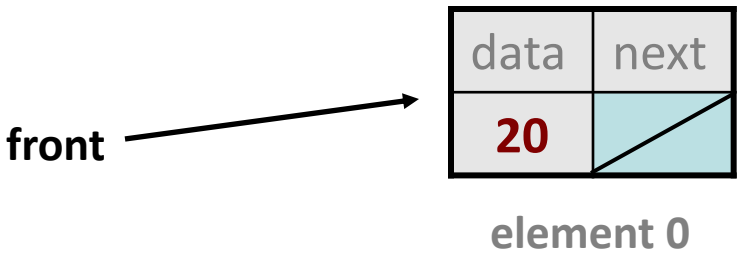




- To remove the first node, we must change `front`.

Removing the only element

- Before:



After:

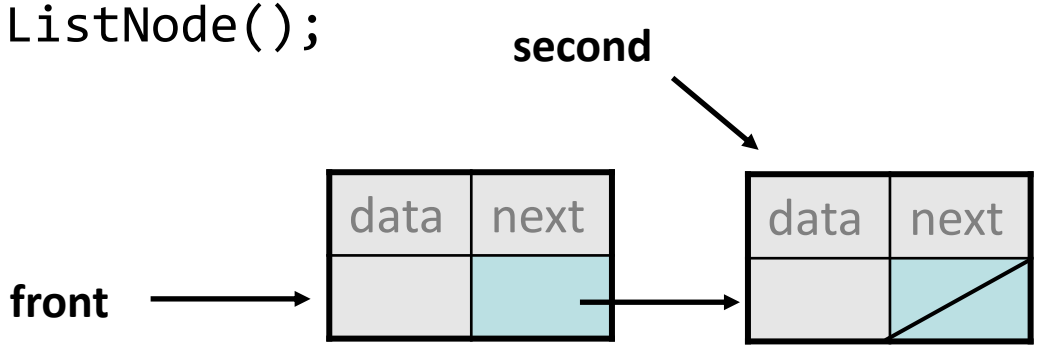
  - We must change the front field to store `nullptr` instead of a node.

Cleaning Up

- If we allocated memory on the heap and no longer need it, it is our responsibility to **delete** it.
- To do this, use the **delete** command and specify the *address on the heap for the memory you no longer need.*
- If you do not do this, your program is said to have a *memory leak*.

```
delete pointer;
```

```
ListNode* front = new ListNode();
ListNode* second = new ListNode();
front->next = second;
...
delete second;
delete front;
```



Implementing remove

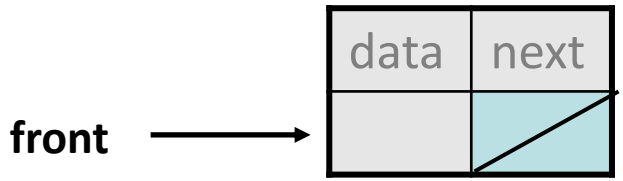
```
void remove(ListNode*& front, int index) {
    if (index == 0) {
        ListNode* nodeToDelete = front;
        front = front->next;
        delete nodeToDelete;
    } else {
        ListNode* current = front;
        for (int i = 0; i < index - 1; i++) {
            current = current->next;
        }

        ListNode* nodeToDelete = current->next;
        current->next = current->next->next;
        delete nodeToDelete;
    }
}
```

Cleaning Up

- If you delete something on the heap, it just deletes the *heap memory*, **not the pointer itself**. The pointer lives on the stack! You can reuse it to point to something else.
- Once you delete something on the heap, you should not refer to it again. Set a pointer to point somewhere else (or to **`nullptr`**) after you have deleted what it pointed to.

```
delete pointer;
```

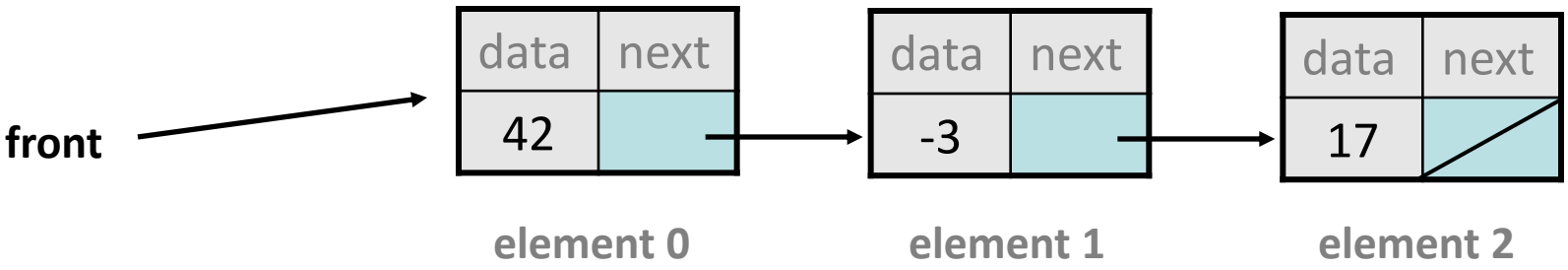
```
ListNode* front = new ListNode();
...
delete front;
front = otherPtr->next;
```



insert

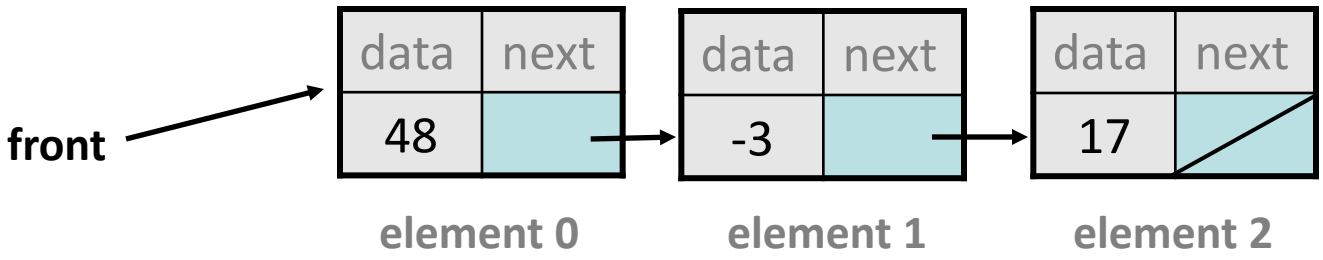
Implement insert

```
// Inserts the given value at the given index.
void insert(ListNode*& front, int index, int value) {
    ...
}
```



Inserting into a list

- Before inserting element at index 2:



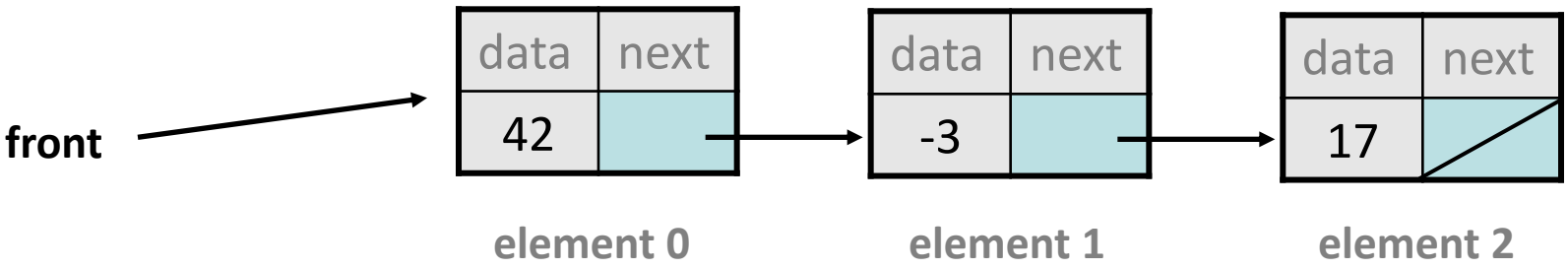
- After:

- **Q:** How many times to advance `current` to insert at index *i* ?

  A. *i* - 1 **B.** *i* **C.** *i* + 1 **D.** none of the above

Implement insert

```
// Inserts the given value at the given index.
void insert(ListNode*& front, int index, int value) {
    if (index == 0) {
        // insert as front element
        front = new ListNode(value, front);
    } else {
        // non-front: walk to proper place in list
        ListNode* curr = front;
        for (int i = 0; i < index - 1; i++) {
            curr = curr->next;
        }

        // insert the node
        ListNode* newNode = new ListNode(value, curr->next);
        curr->next = newNode;
    }
}
```

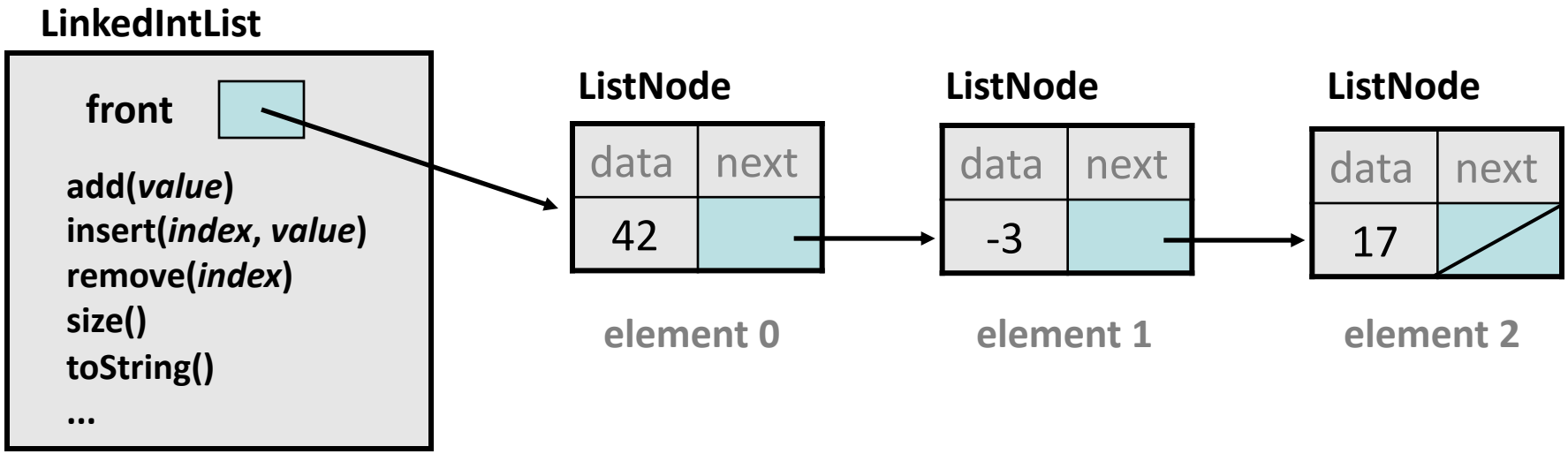


Plan For Today

- Implementing a Linked List
  - Pointers
  - Dynamic memory
  - Classes
  - Testing

A LinkedIntList class

- Let's write a collection class named **`LinkedIntList`**.
  - Has the same public members as `Vector`
    - `add`, `clear`, `get`, `insert`, `isEmpty`, `remove`, `size`, `toString`
  - The list is internally implemented as a chain of linked nodes
    - The `LinkedList` keeps a pointer to its `front` node as a field
    - `nullptr` is the end of the list; a null front signifies an empty list



LinkedIntList.h

```
/* Represents a linked list of integers. */
class LinkedIntList {
public:
    LinkedIntList();
    ~LinkedIntList();
    void add(int value);
    void clear();
    int get(int index) const;
    void insert(int index, int value);
    bool isEmpty() const;
    void remove(int index);
    void set(int index, int value);
    int size() const;

private:
    ListNode* front;   // null if empty
};
```