CS 106X, Lecture 16
Linked Lists Summary

reading:

Programming Abstractions in C++, Chapters 11-12, 14.1-14.2

This document is copyright (C) Stanford Computer Science and Nick Troccoli, licensed under Creative Commons Attribution 2.5 License. All rights reserved.
Based on slides created by Keith Schwarz, Julie Zelenski, Jerry Cain, Eric Roberts, Mehran Sahami, Stuart Reges, Cynthia Lee, Marty Stepp, Ashley Taylor and others.

Personal Note on Linked List

/*Summary:
* Use pointer to represent a list: **ListNode* front;**
* Store **pointer in stack**; Store **list in heap: new ListNode(value);**
* When iterating a list: make a **copy of front**, i.e ListNode* current = front, use curr to iterate the list. Nodes are like ballons, we need a string-namely front- to tie them up.
* For functions that change the list: **use reference to pointer: ListPointer*& name;**
* Iterate over the list: while (curr) {} go all the way down
* Operation over list: **James Bond Analogy**: curr goes to the **node BEFORE the node of interest!**
* Consider special case when index = 0 since there is no carriage you Bond can stand on it.
* Also consider: empty list
* Remove element: clean up memory
* Removal Mode: ListNode* trash = object;
* Operations;
* delete trash;
*/
* Meaning of ->:Movement: a->next means follow the pointer a and walk to the object that it points to, and use dot to find attribute.

核心总结:

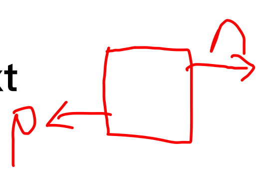
1. 链表相关操作设计通法：仔细的do an instance yourself. write what you have done in detail

2. 操作函数传参数:node* or node*&? 更改指针所指对象node用*，改变指针本身指的位置用*&

3. 习惯delete完一个指针之后要把指针设置成nullptr，不要出现dangling ptr

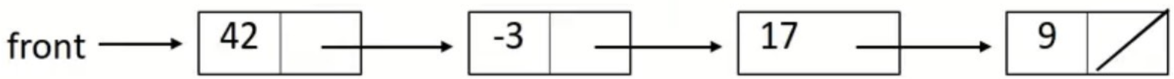
5. 存链表：stack里面存个front指针，节点都存在heap里面

6. 双向链表的node的prev和next
一般双向链表还有一个end指针



Linked data structures

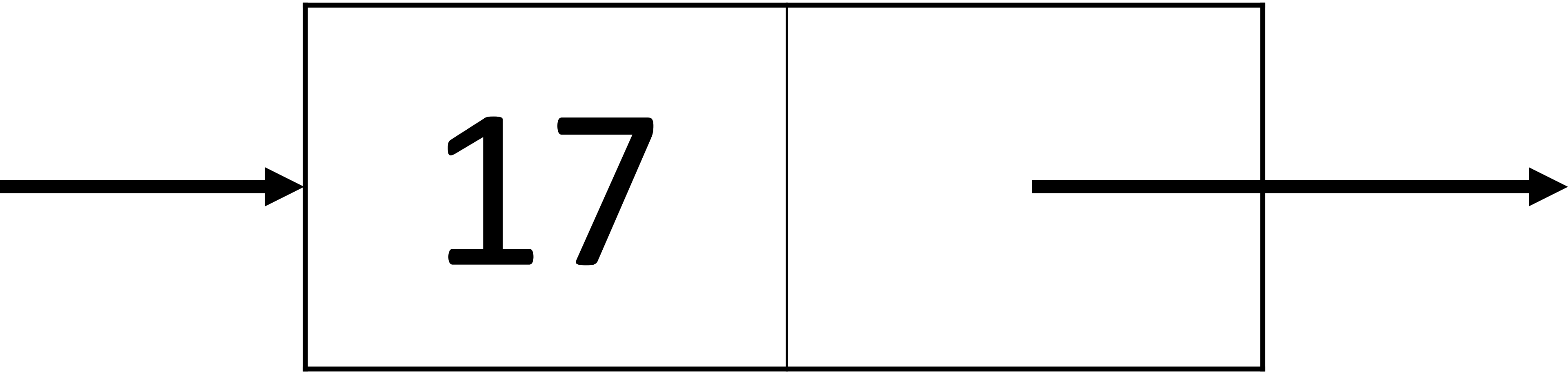
- Some collections store their data using **linked node** structures.
  - Each node stores one element and a link to another node(s).
  - examples: `Set`, `Map`, `LinkedList`



- *Pros:* fast to add/remove at any point; no shifting, no resizing.
- *Cons:* slow to access certain parts of the list.

- This week we will learn how to create a *linked list*.
- To do linked lists, we must understand some new C++ concepts.
  - To represent nodes, we need to know about ***structs***.
  - To link nodes to each other, we must learn ***pointers***.

Nodes

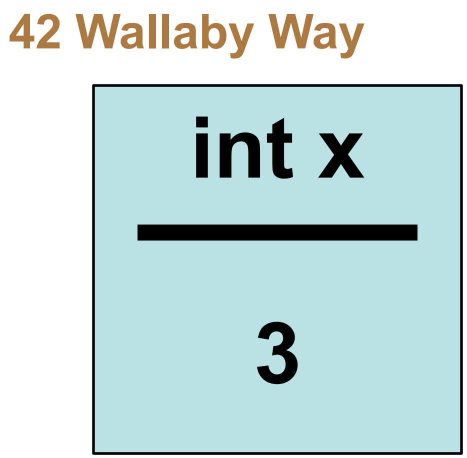



```
struct ListNode {
    int data;
    ListNode *next;
};
```

Pointers

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

Addresses




```
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;
```

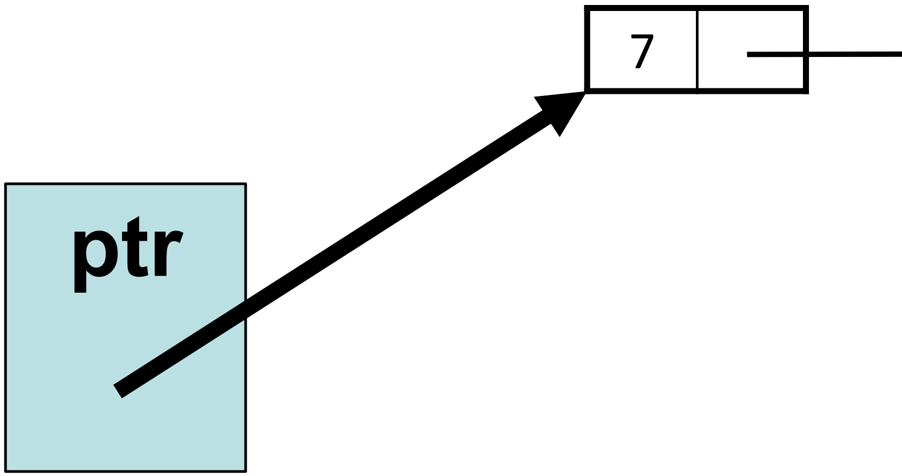
The `*` operator is the **dereference** operator. It tells C++ to *go to the variable* at the address stored in that pointer.

Dereference Classes/Structs

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

ptr->data = 7;
```

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

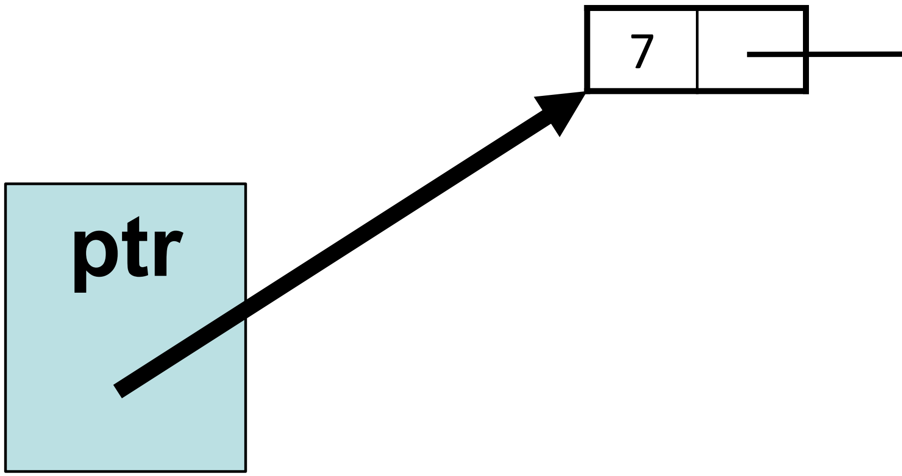


Dereference Classes/Structs

```
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

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

`nullptr` is a special value that represents “no address”.

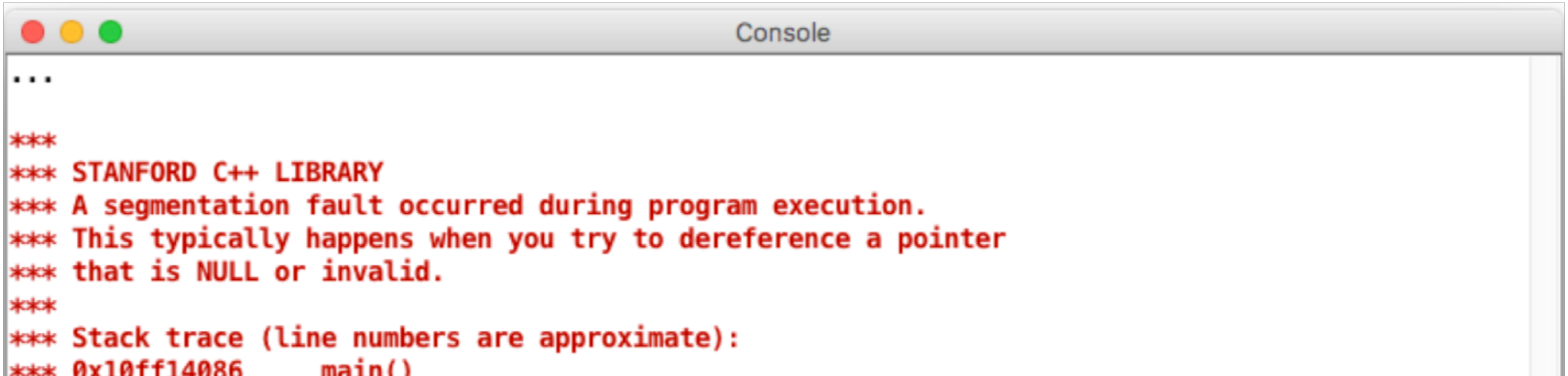
Dereferencing nullptr

int *xAddress

nullptr

`int *xAddress = nullptr;`

`cout <<` ~~`*xAddress`~~ `<< endl;`

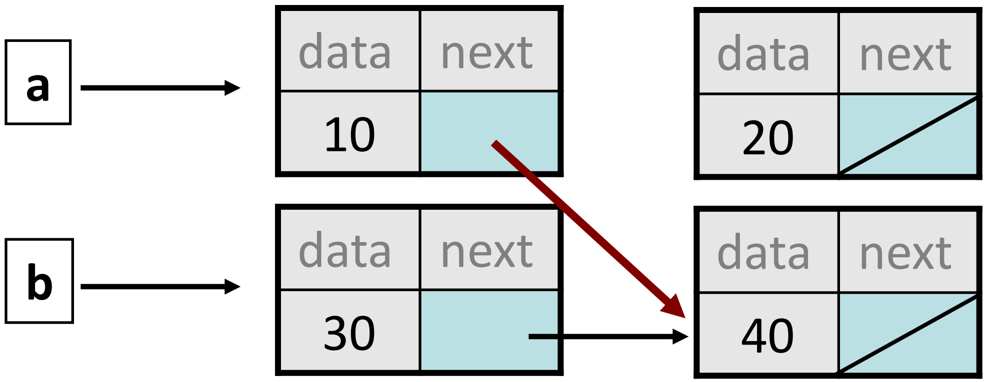


Garbage Pointers


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


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

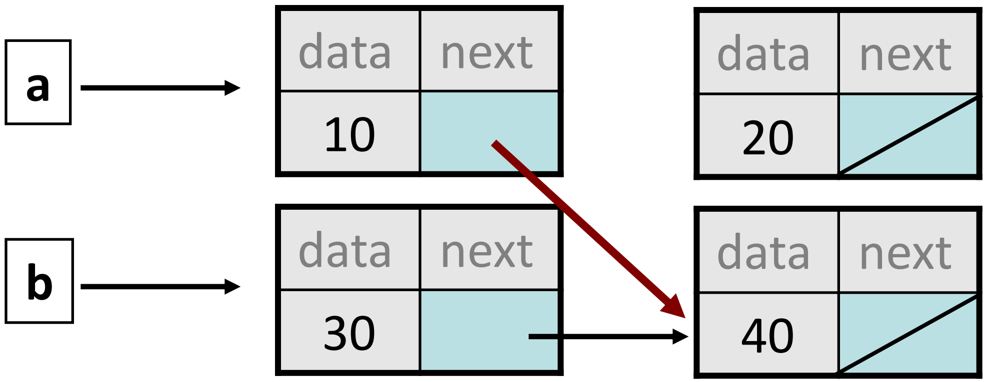
Reassigning Pointers




```
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

IMPORTANT!

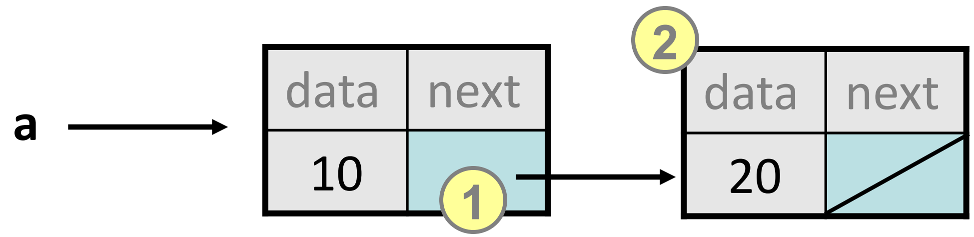
`variable = value;`

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)*

Assignment between pointers are different from assignment between varables.
Understanding of pointer: an arrow points to "pointee"

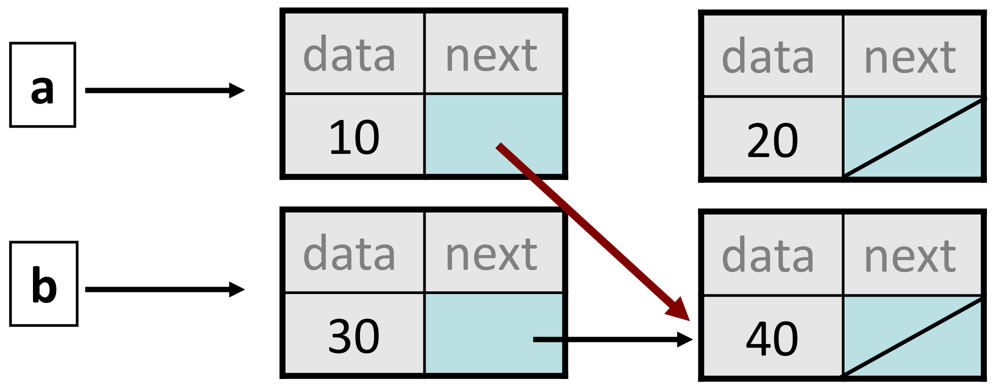
- For the list at right:



`a->next = p;` **tell the next arrow points to the same place where p points.**
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

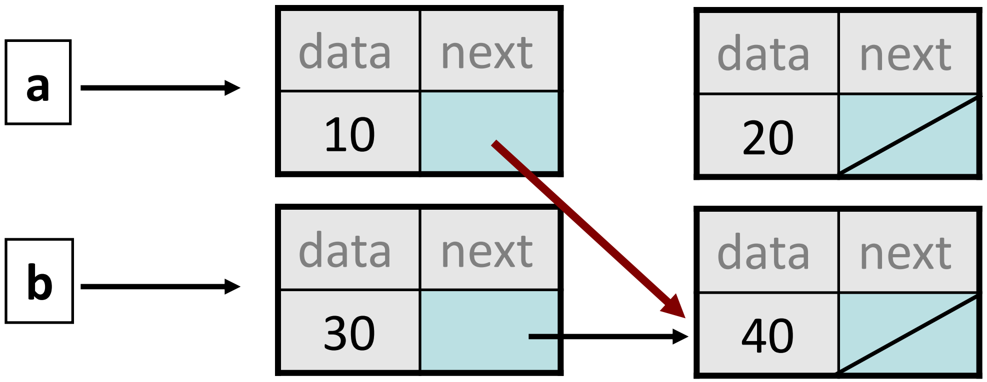
Reassigning Pointers




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

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

Reassigning Pointers

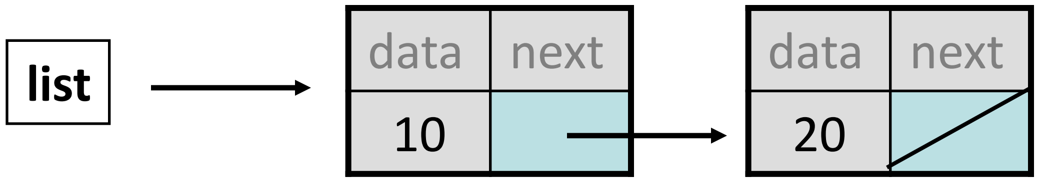


~~`a->next = firstNode;`~~

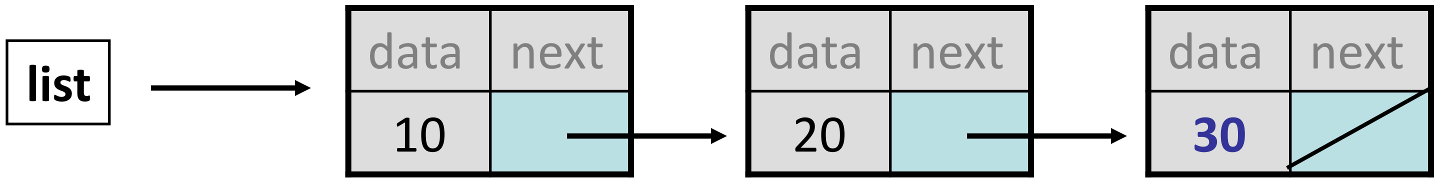
Tip: the types on the left- and right-hand sides must always match!

Linked node problem 1

- Which statement turns this picture:



- Into this?




```
ListNode node = {30, nullptr};
```

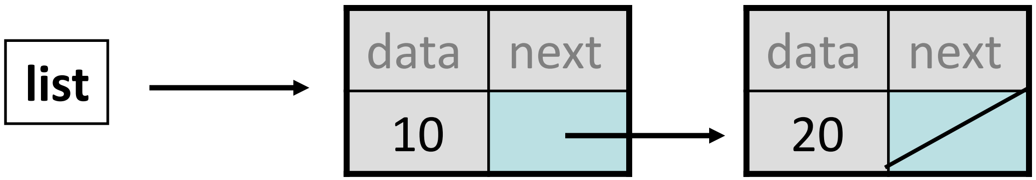
A. `list->next = node;`

B. `list->next->next = &node;`

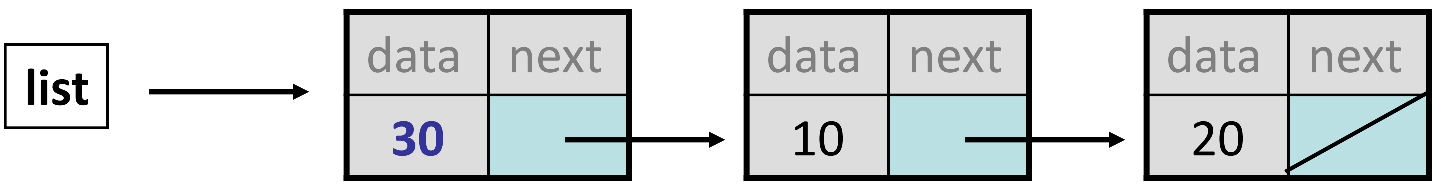
C. `list->next->next->next = node;`

Linked node problem 2

- Which statements turn this picture:



- Into this?




```
ListNode temp = {30, nullptr};
```

A. `temp.next = list;` `list = &temp;`

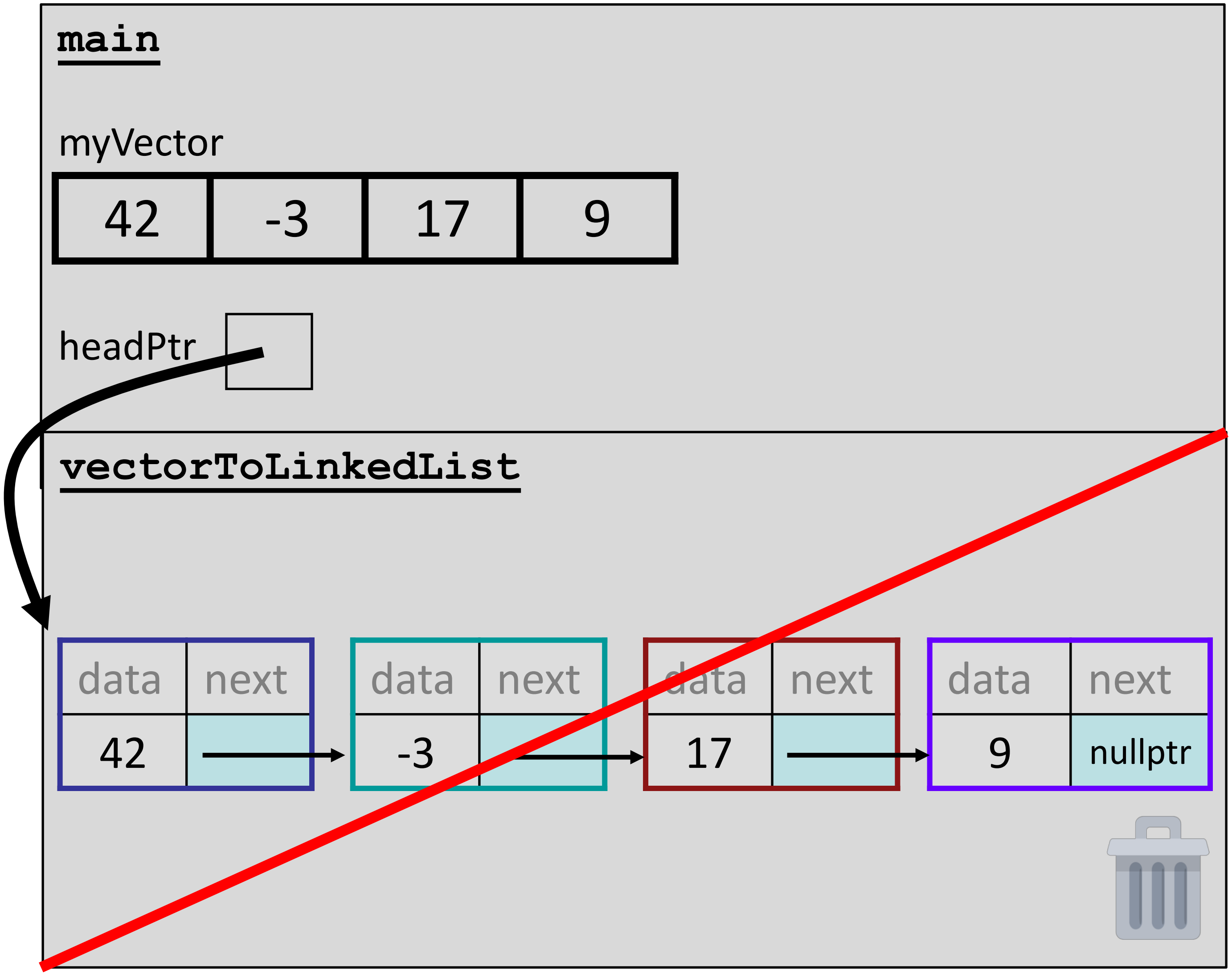
B. `temp = &list;` `list = temp.next;`

C. `temp.next = list->next;` `list->next = &temp;`

Creating a List

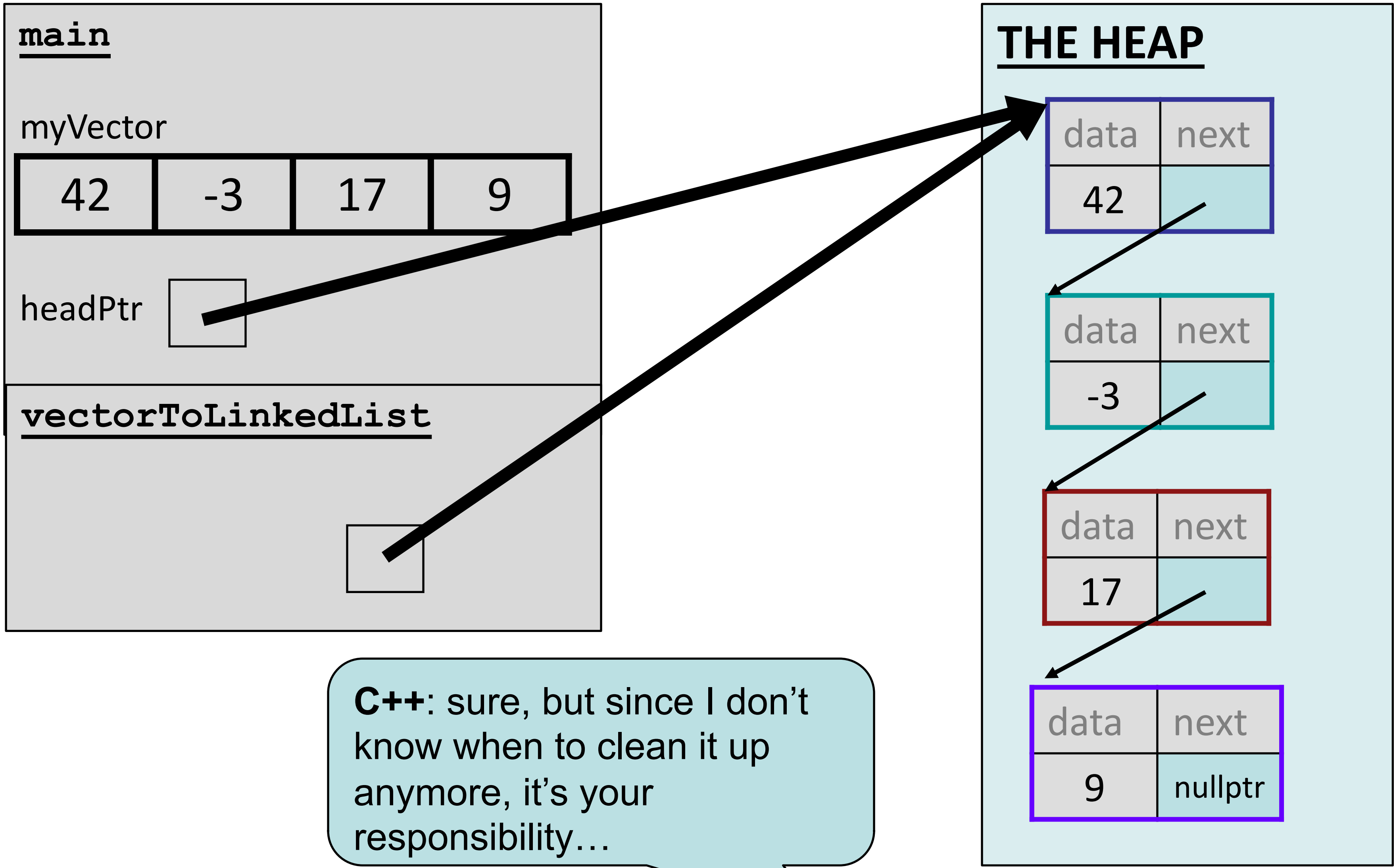
We need a way to have memory that doesn't get cleaned up when a function exits.

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



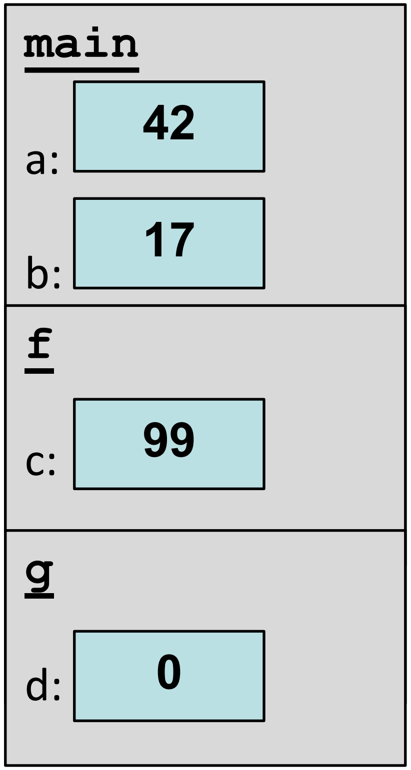
C++: sure, but since I don't know when to clean it up anymore, it's your responsibility…

The Stack

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

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

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



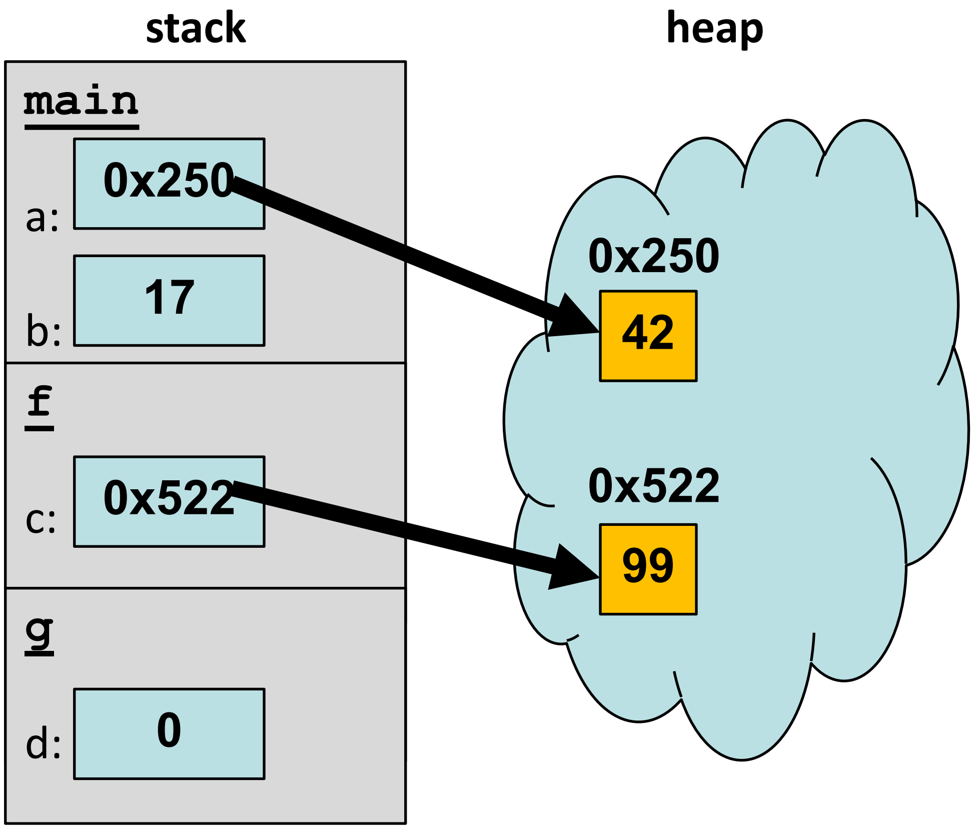
The **stack** is the place where all local variables live. Anything you declare as a local variable in a function lives on the stack. A function's stack "frame" goes away when the function returns.

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;
}
```

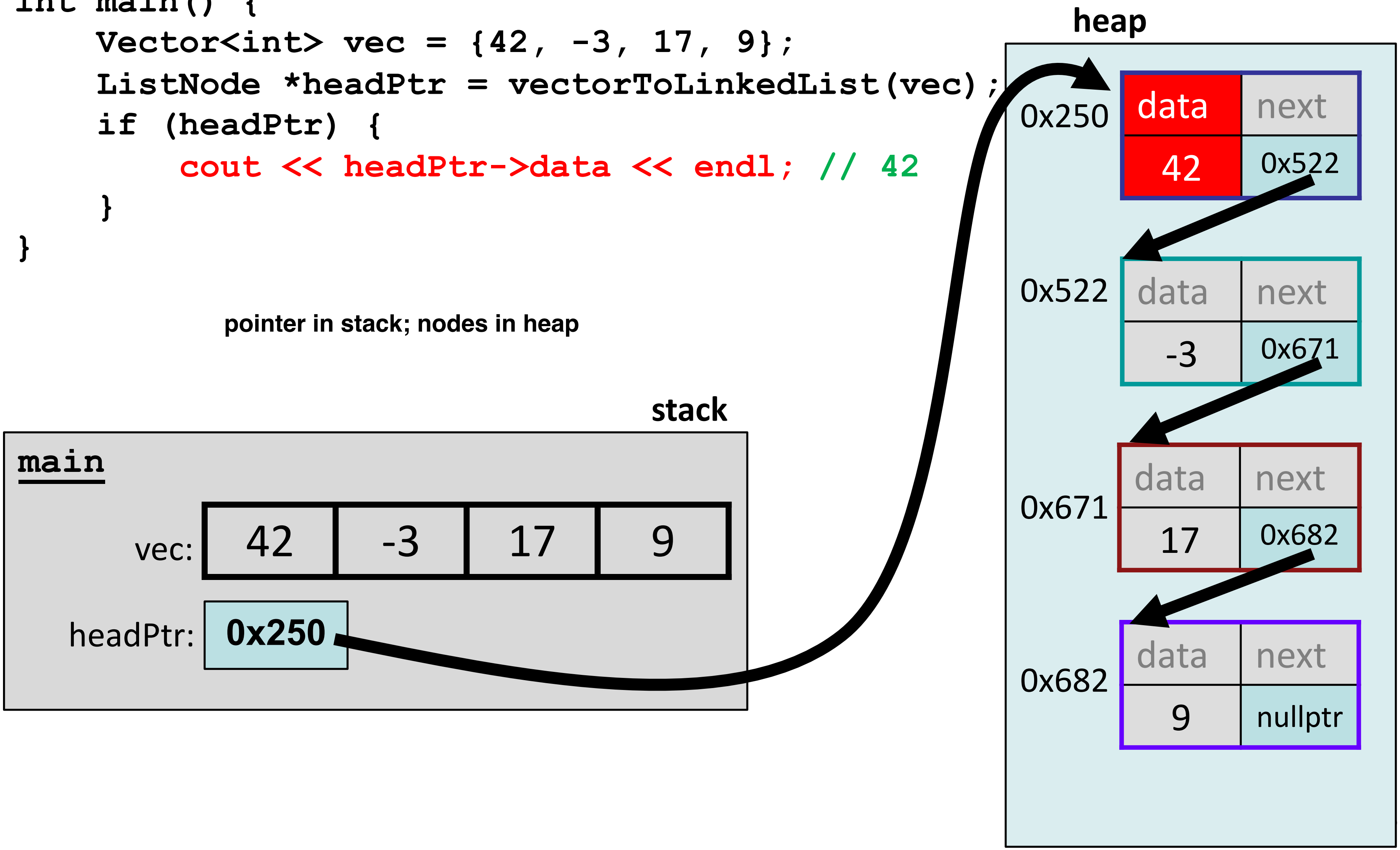


The **heap** is a part of memory that you can manage yourself. Unlike the stack, the memory only goes away when you delete it yourself. To allocate memory on the heap, use the **new** keyword. **new** returns a *the address on the heap of the new memory*.

Creating a List

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

pointer in stack; nodes in heap



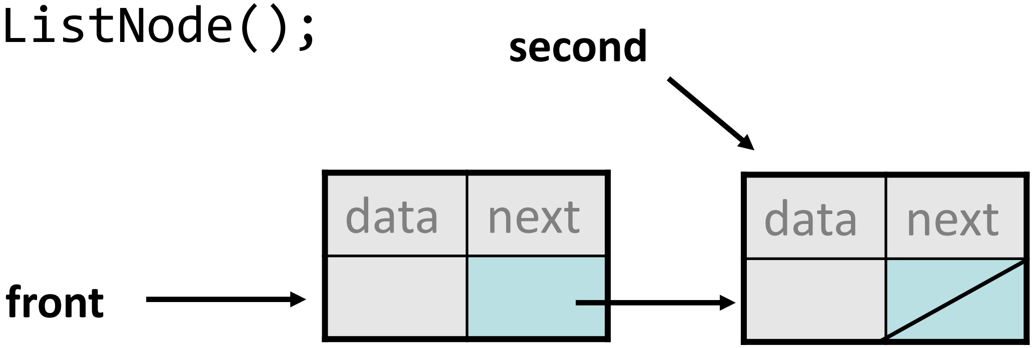
防内存泄漏，要用delete删除，相当于c中的free

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;
    }
}
```

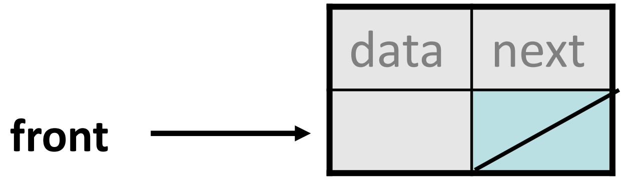
Cleaning Up

- 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删除的不是ptr本身，而是其指向的位置；删完一个ptr指向位置之后最好将ptr设成nullptr或是别的具体的位置；没设置不要再次**调用/删除**！

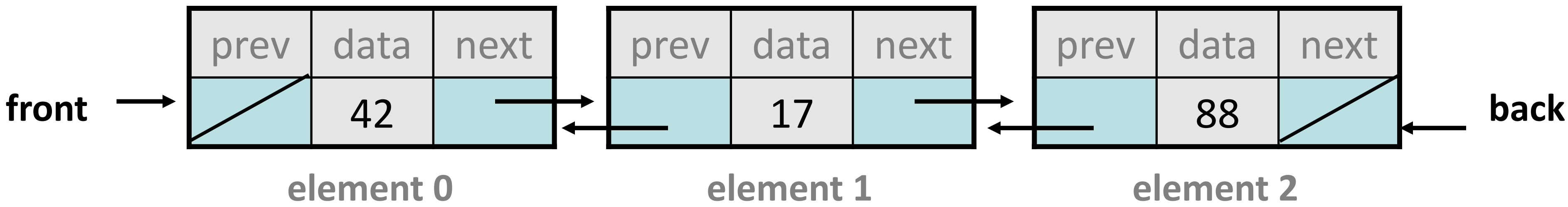
```
delete pointer;
```

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



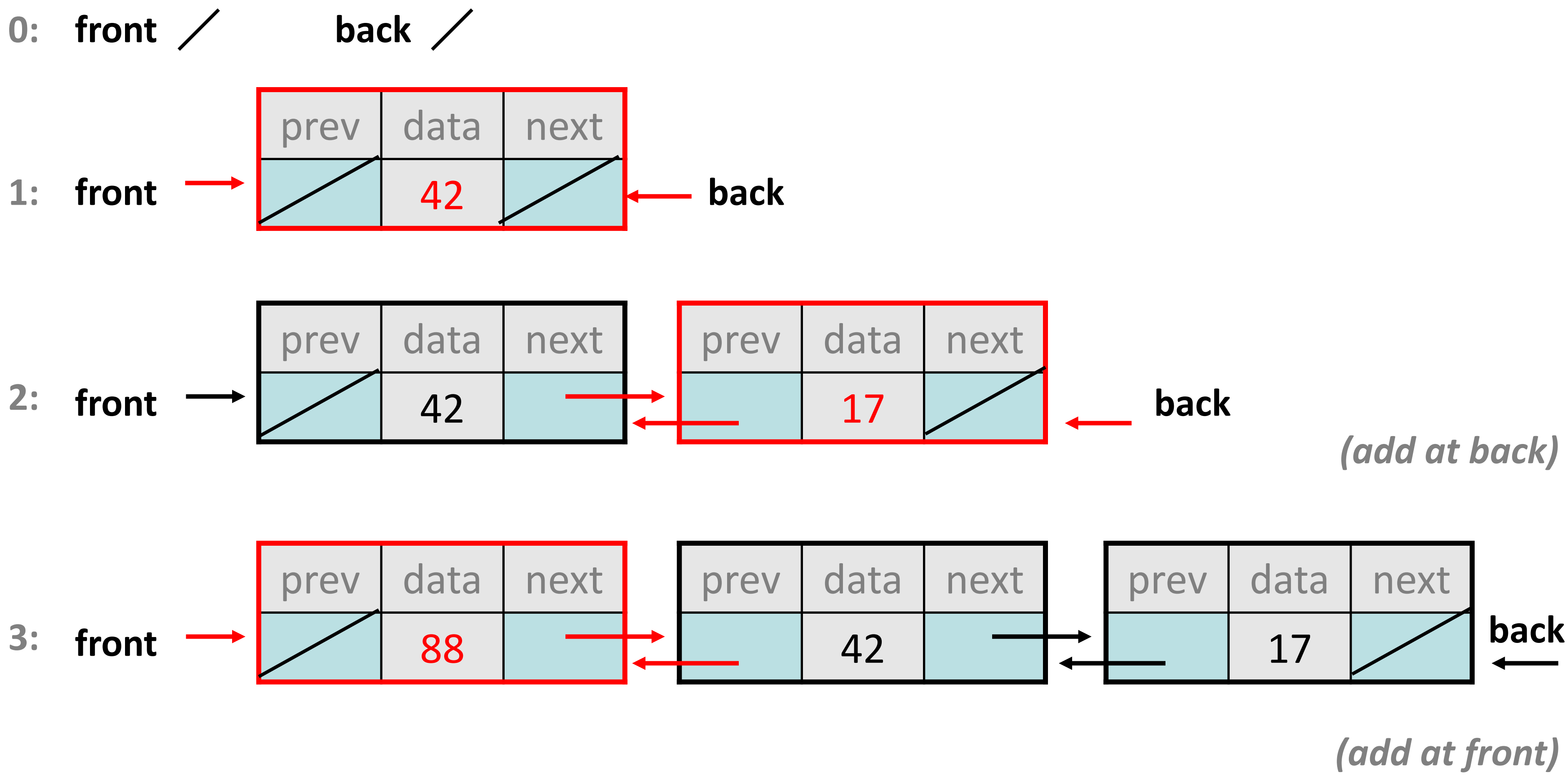
Doubly linked list

- **doubly linked list**: Each node has a pointer to next and prev node.
  - Allows walking forward and backward in list efficiently.
  - Overall list often maintains a **back** pointer to end of list.



D.L. list growth

- State of a doubly linked list of 0, 1, 2, *N* nodes:



D.L. list remove

- When removing a node, must change two pointers.
  - Might also need to change front and/or back.
  - Example: Try removing each of the three nodes below.