CS 106B, Lecture 18
Linked Lists

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

Plan for Today

- Continuing discussion of ArrayStack from last week
- Learn about a new way to store information: the linked list

A Stack Class

- Recall from last Monday our ArrayStack
- By storing the array on the heap, the memory existed for all the ArrayStack member functions

Flaws with Arrays

- Some adds are very costly (when we have to resize)
  - Adding just **one** element requires copying **all the elements**
- Imagine if everything were like that?
  - Instead of just grabbing a new sheet of paper, re-copy all notes to a bigger sheet when you run out of space

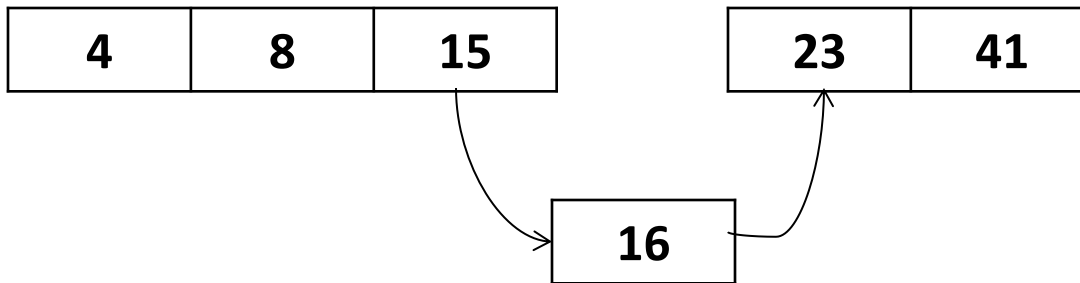
- Idea: what if we could just add the amount of memory we need?

4	8	15	16	23

42

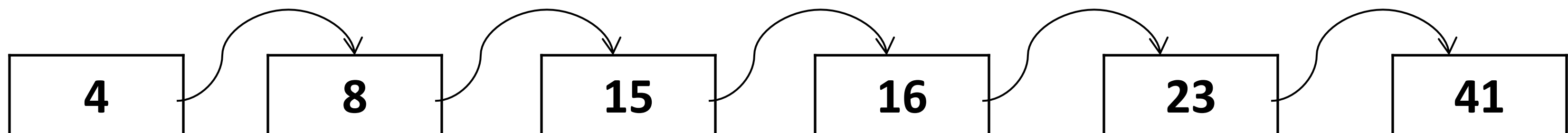
Vector and arrays

- Inserting into an array involves **shifting** all the elements over
  - That's O(N)
- What if we were able to easily insert?



Linked List

- Main idea: let's store every element in **its own block of memory**
- Then we can just add one block of memory!
- Then we can efficiently insert into the middle (or front)!
- A **Linked List** is good for storing elements in an order (similar to Vector)
- Elements are chained together in a sequence
- Each element **is allocated on the heap**

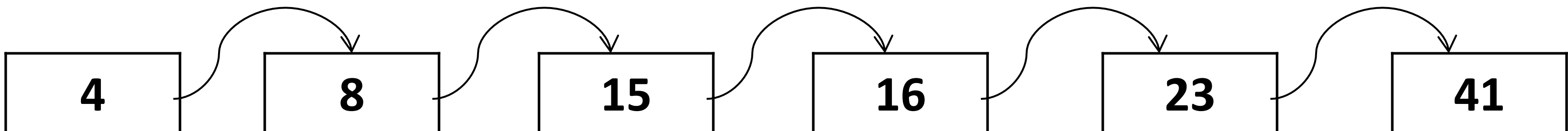


Parts of a Linked List

- What does each part of a Linked List need to store?
  - element
  - pointer to the next element
  - We'll say the last node points to **nullptr**
- The ListNode struct:

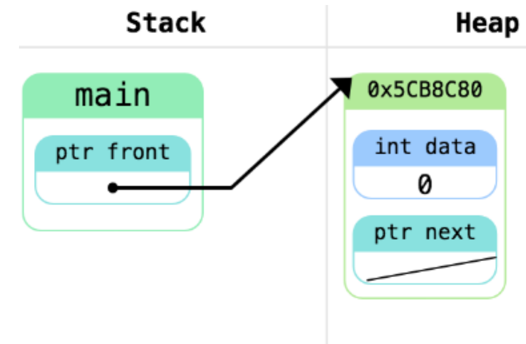
```
struct ListNode {
    int data; // assume all elements are ints
    ListNode *next;

    // constructor
    ListNode(int data, ListNode *next): data(data), next(next) {}
    // constructor without params
    ListNode(): data(0), next(nullptr) {}
};
```



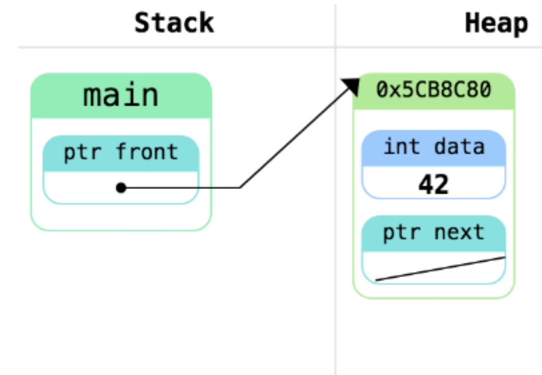
Creating a Linked List

```
ListNode* front = new ListNode();
```



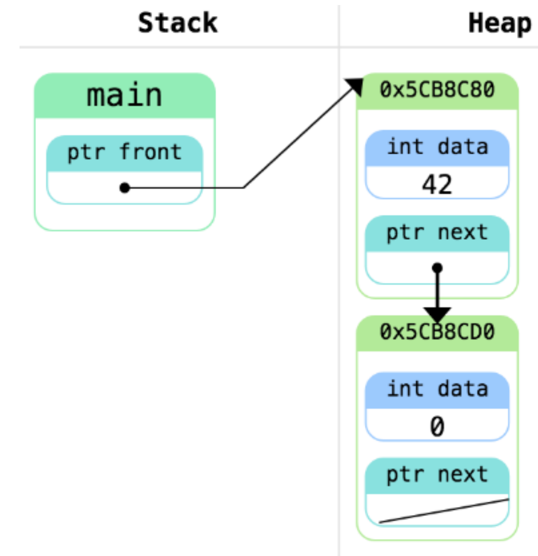
Creating a Linked List

```
ListNode* front = new ListNode();
front->data = 42;
```



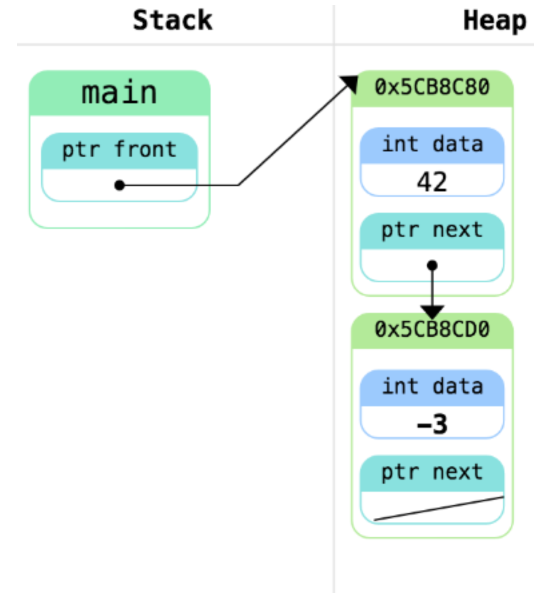
Creating a Linked List

```
ListNode* front = new ListNode();
front->data = 42;
front->next = new ListNode();
```



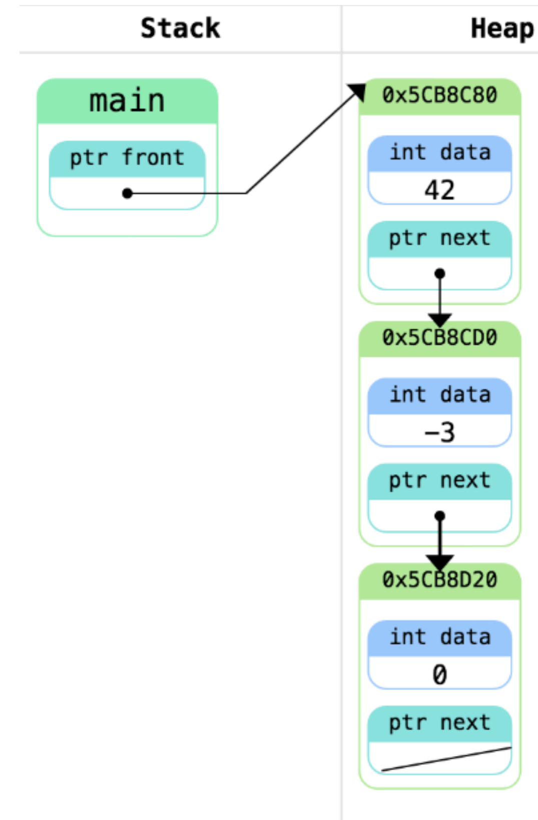
Creating a Linked List

```
ListNode* front = new ListNode();
front->data = 42;
front->next = new ListNode();
front->next->data = -3;
```



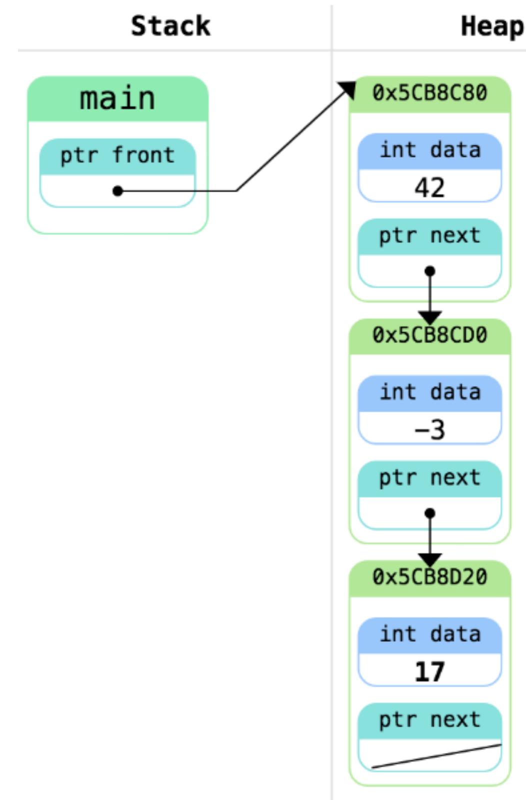
Creating a Linked List

```
ListNode* front = new ListNode();
front->data = 42;
front->next = new ListNode();
front->next->data = -3;
front->next->next = new ListNode();
```



Creating a Linked List

```
ListNode* front = new ListNode();
front->data = 42;
front->next = new ListNode();
front->next->data = -3;
front->next->next = new ListNode();
front->next->next->data = 17;
front->next->next->next = nullptr;
```

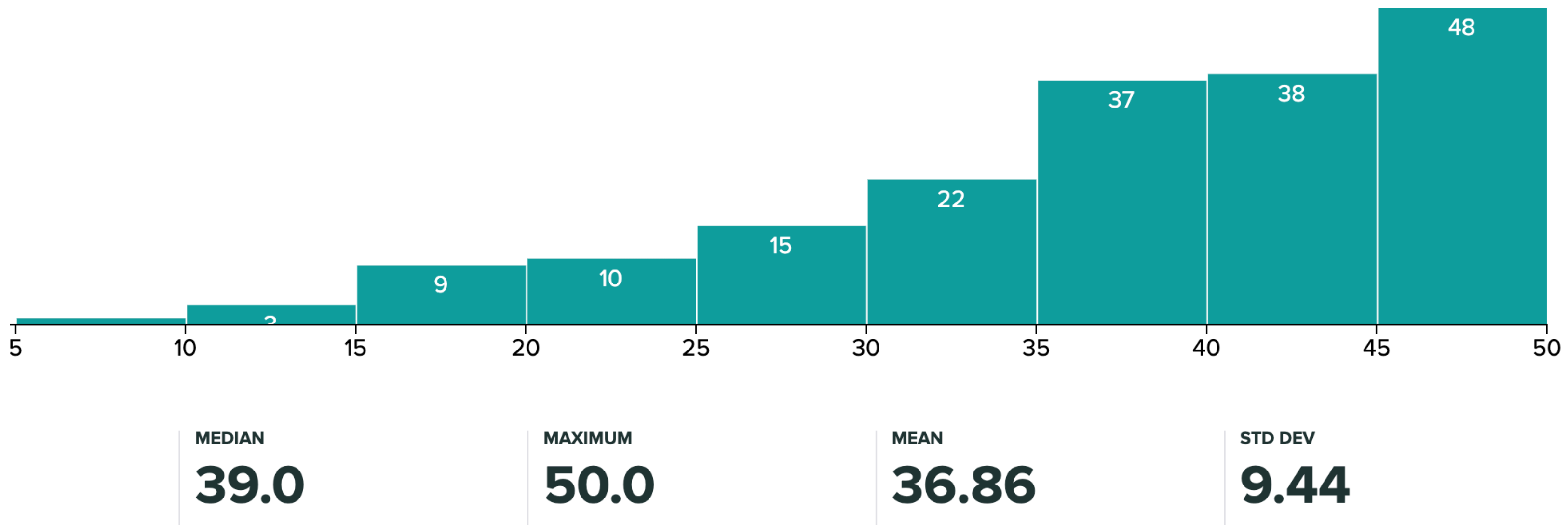


Announcements

- Assignment 5 is due on **Thursday**
- Section attendance is **mandatory**
- **You can only partner with people in your section.**
  - If you partner with someone not in your section. It will result in a bucket grade deduction.
- Midterm/Regrade Requests
  - To check grade, go to gradescope.com and make an account. **Use your Stanford email for the account**.

Announcements

- Assignment 5 is due on **Thursday**
- Section attendance is **mandatory**
- **You can only partner with people in your section.**
  - If you partner with someone not in your section. It can result in a bucket grade deduction.
- Midterm/Regrade Requests



Boolean Zen

- Boolean Zen

```
if (functionCall() == true) {
    return true;
} else {
    return false;
}
```

Boolean Zen

- Better

```
if (functionCall()) {
    return true;
} else {
    return false;
}
```

Boolean Zen

- Best

```
return functionCall();
```

Collections

- Maps (and other ADTs)

```
string possibleKey = “Biden”;
for (string mapKey : map) {
    if (possibleKey == mapKey) {
        int value = map[possibleKey];
    }
}
```

- Review your ADTs! You need to be able to choose which ADT is appropriate and be able to use all of the ADTs we have learned.

Assn. 5

- The header file (.h file) is meant for others to peruse when using your class. This is the interface. It should describe how to use the functions of your class, not how your class is implemented.

- The implementation file (.cpp file) should have comments describing the implementation of your class. The users of your class typically wouldn't look at the .cpp file.

Linked List iteration

- Idea: travel each ListNode one at a time
  - No easy way to "index in" like with Vector. Why?
- General syntax:

```
for (ListNode* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```

Linked List iteration

- Idea: travel each ListNode one at a time
  - No easy way to "index in" like with Vector. Why?
- General syntax:

```
for (ListNode* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```

Initialize ptr to the first node in (front node of) the list

Linked List iteration

- Idea: travel each ListNode one at a time
  – No easy way to "index in" like with Vector. Why?
- General syntax:

```
for (ListNode* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```

Move ptr to point to the next node of the list

Linked List iteration

- Idea: travel each ListNode one at a time
  - No easy way to "index in" like with Vector. Why?
- General syntax:

```
for (ListNode* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```

Continue doing this until we hit the end of the list

Practice Iteratively!

- Write a function that takes in the pointer to the front of a Linked List and prints out all the elements of a Linked List

```
void printList(ListNode *front) {





}
```

Practice Iteratively!

- Write a function that takes in the pointer to the front of a Linked List and prints out all the elements of a Linked List

```
void printList(ListNode *front) {
  for (ListNode* ptr = front; ptr != nullptr; ptr = ptr->next) {
      cout << ptr->data << endl;
  }
}
```

Alternative Iteration

```
for (ListNode* ptr = front; ptr != nullptr; ptr = ptr->next)
{
    // do something with ptr
}
```

is equivalent to:

```
ListNode *ptr = front;
while (ptr != nullptr) { // or while (ptr)
    // do something with ptr
    ptr = ptr->next;
}
```

A Temporary Solution

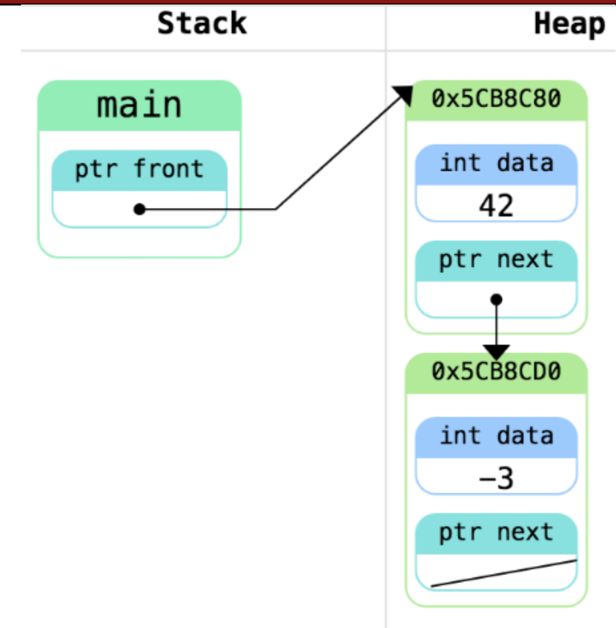
What's wrong?

```
int main() {
  ListNode* front = new ListNode();
  front->data = 42;
  front->next = new ListNode();
  front->next->data = -3;
  front->next->next = nullptr;
  while (front != nullptr) {
    cout << front->data << " ";
    front = front->next;
  }
  // continue using front
  return 0;
}
```

A Temporary Solution

What's wrong?

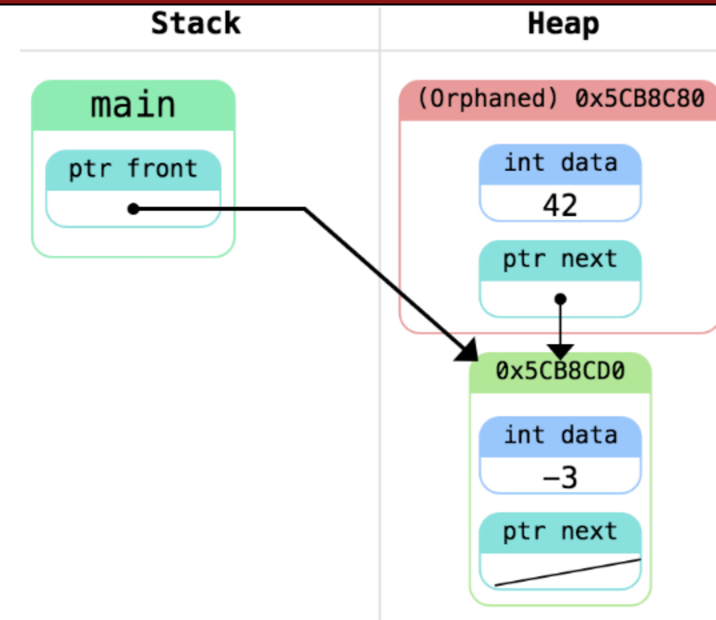
```
int main() {
  ListNode* front = new ListNode();
  front->data = 42;
  front->next = new ListNode();
  front->next->data = -3;
  front->next->next = nullptr;
  while (front != nullptr) {
    cout << front->data << " ";
    front = front->next;
  }
  // continue using front
  return 0;
}
```



A Temporary Solution

What's wrong?

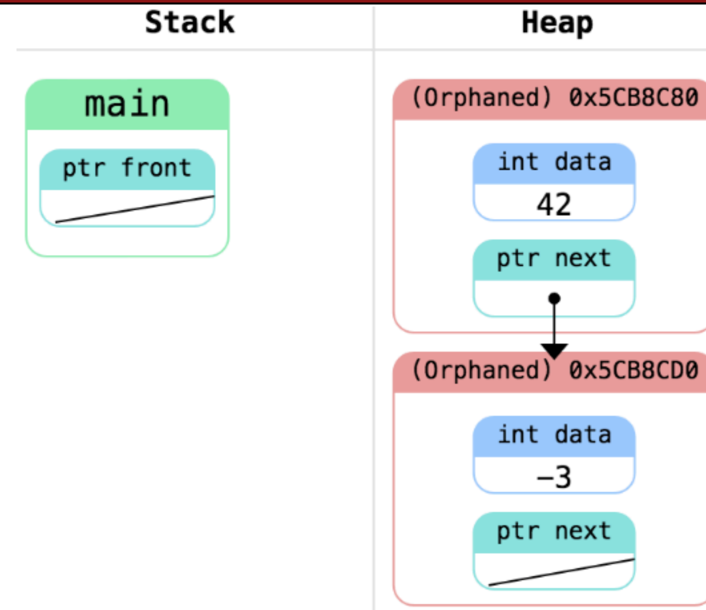
```
int main() {
  ListNode* front = new ListNode();
  front->data = 42;
  front->next = new ListNode();
  front->next->data = -3;
  front->next->next = nullptr;
   while (front != nullptr) {
    cout << front->data << " ";
    front = front->next;
  }
  // continue using front
  return 0;
}
```



A Temporary Solution

What's wrong?

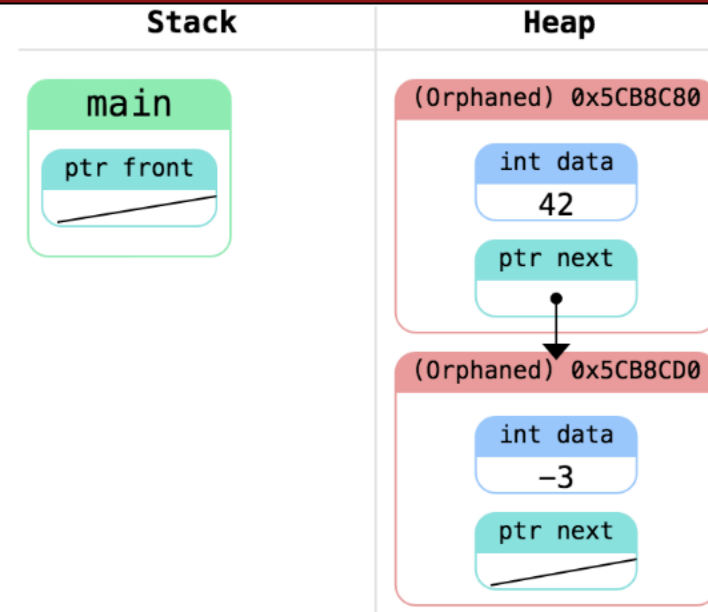
```
int main() {
  ListNode* front = new ListNode();
  front->data = 42;
  front->next = new ListNode();
  front->next->data = -3;
  front->next->next = nullptr;
   while (front != nullptr) {
    cout << front->data << " ";
    front = front->next;
  }
  // continue using front
  return 0;
}
```



A Temporary Solution

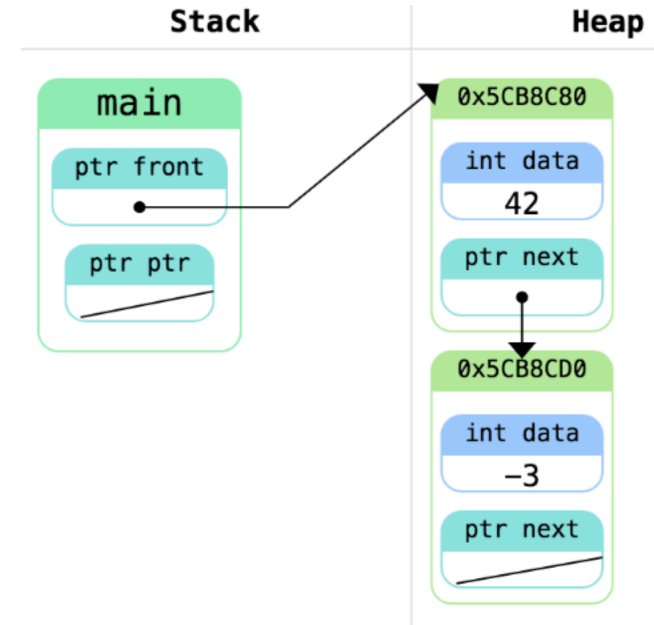
What's wrong?

```
int main() {
  ListNode* front = new ListNode();
  front->data = 42;
  front->next = new ListNode();
  front->next->data = -3;
  front->next->next = nullptr;
   while (front != nullptr) {
    cout << front->data << " ";
    front = front->next;
  }
  // orphaned memory and empty list!
  return 0;
}
```



Correct Version

```
int main() {
  ListNode* front = new ListNode();
  front->data = 42;
  front->next = new ListNode();
  front->next->data = -3;
  front->next->next = nullptr;
  ListNode *ptr = front;
  while (ptr != nullptr) {
    cout << ptr->data << " ";
    ptr = ptr->next;
  }
  // front still has pointer to list
  return 0;
}
```



Overflow

- From the book: 12.7 Copying Objects
  - Shallow copying. C++ defaults to copy all instance variables. But if any are dynamically allocated, it will just copy the pointer (the address). Deep copying copies the underlying data as well.
  - Two choices for user-defined classes: Either implement deep copying or forbid copying.