# CS 106B, Lecture 16
# Linked Lists

# 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 Thursday our ArrayStack
- By storing the array on the heap, the memory existed for all the Stack member functions
- One limitation: our Stack only stored ints
  - How could we expand it to be able to store every type, like the real Stack?

# Template class

- **Template class**: A class that accepts a type parameter(s).
  - In the header and cpp files, mark each class/function as templated.
  - Replace occurrences of the previous type `int` with T in the code.
  - See GeneralStack in today's starter code for example

```
// ClassName.h
template<typename T>
class ClassName {
    ...
};

// ClassName.cpp
template<typename T>
type ClassName<T>::name(parameters) {
    ...
}
```

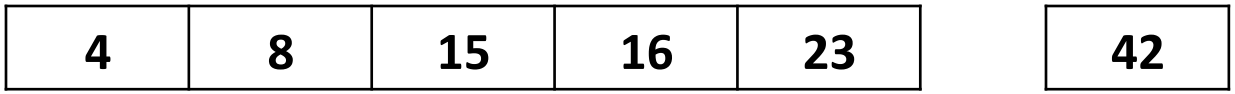
# Template .h and .cpp

- Because of an odd quirk with C++ templates, the separation between .h header and .cpp implementation must be reduced.
  - Either write all the bodies in the .h file (suggested),

```
// ClassName.h
#ifndef _classname_h
#define _classname_h

template<typename T>
class ClassName {
    ...
};

template<typename T>
type ClassName<T>::method1(...) {...}
...

#endif   // _classname_h
```

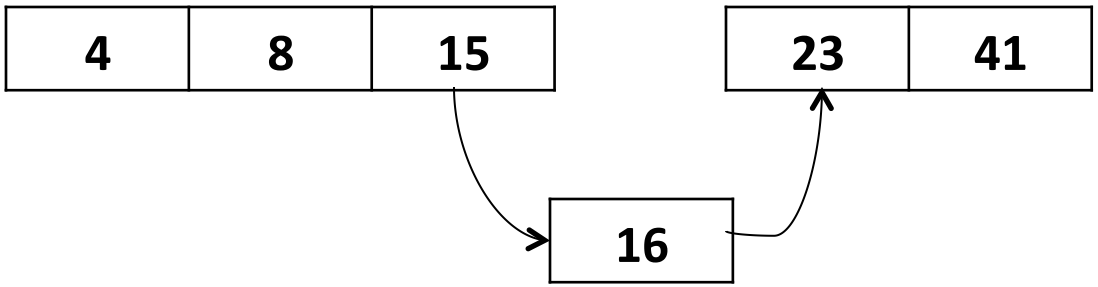
# Flaws with Arrays

- 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
  - Instead of just making a new bend in a line, make everyone move to a larger area
- Idea: what if we could just add the amount of memory we need?



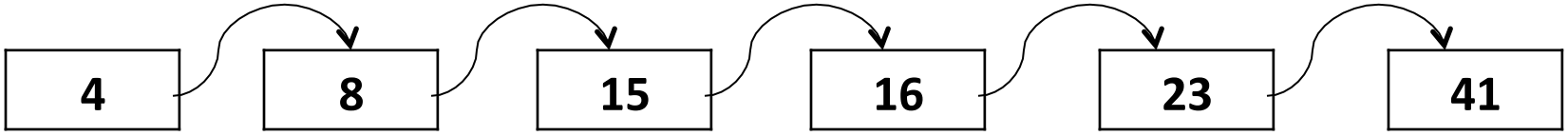
# Vector and arrays

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



# Linked List

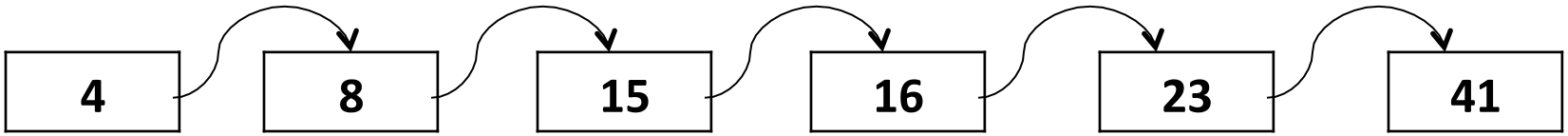
- Main idea: let's store every element in **its own block of memory**
- Then we can just add one block of memory!
- Then we can efficiently insert into the middle (or front)!
- A **Linked List** is good for storing elements in an order (similar to Vector)
- Elements are chained together in a sequence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



# Parts of a Linked List

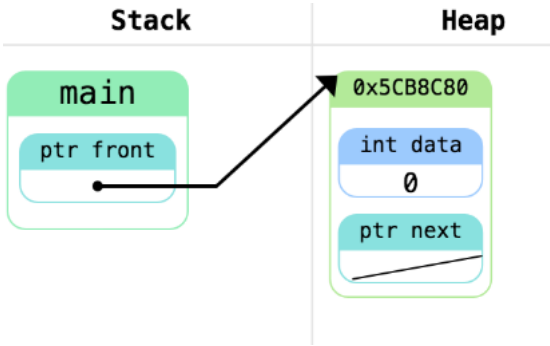
- What does each part of a Linked List need to store?
  - element
  - pointer to the next element
  - We'll say the last node points to **nullptr**
- The ListNode struct:

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

    // constructor
    ListNode(int data, ListNode *next): data(data), next(next) {}
    // constructor w/out params
    ListNode(): data(0), next(nullptr) {}
};
```



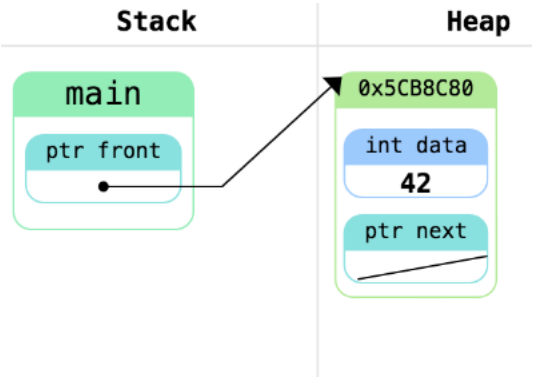
# Creating a Linked List

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



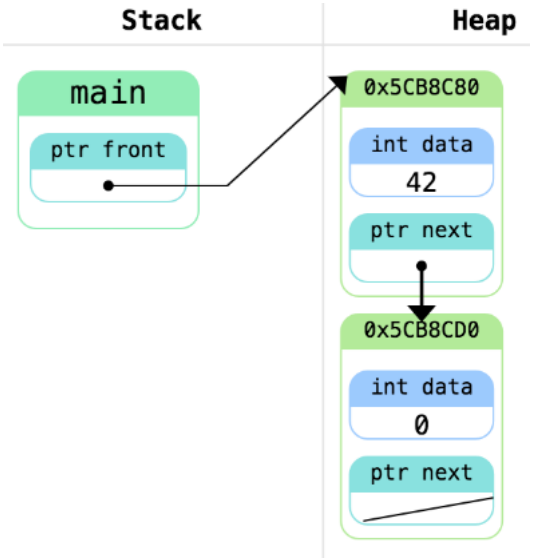
# Creating a Linked List

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



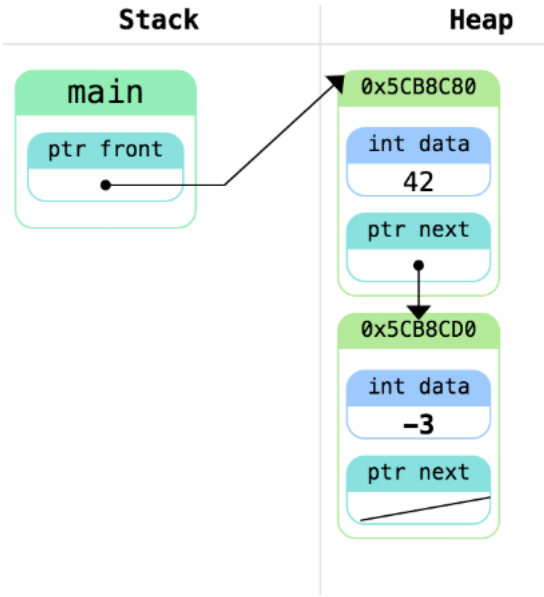
# Creating a Linked List

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



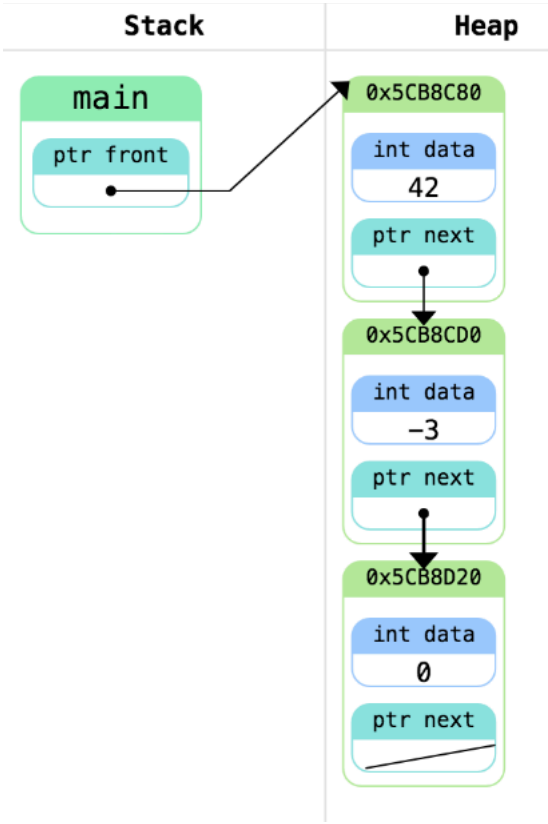
# Creating a Linked List

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



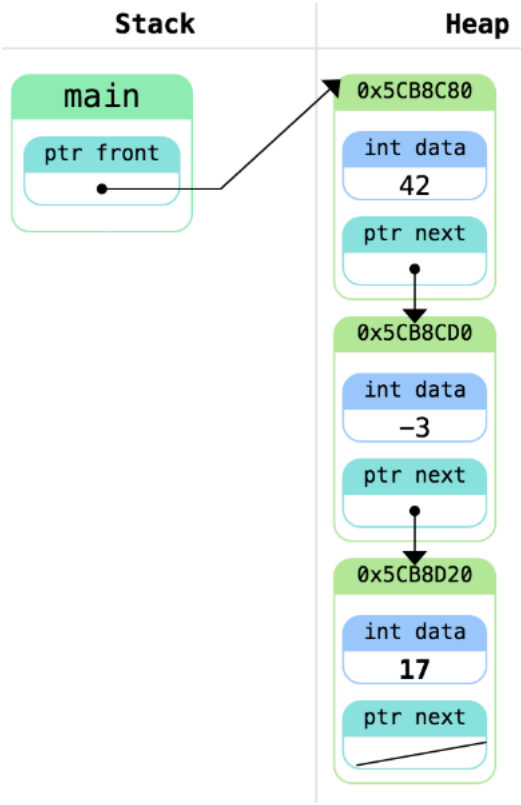
# Creating a Linked List

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



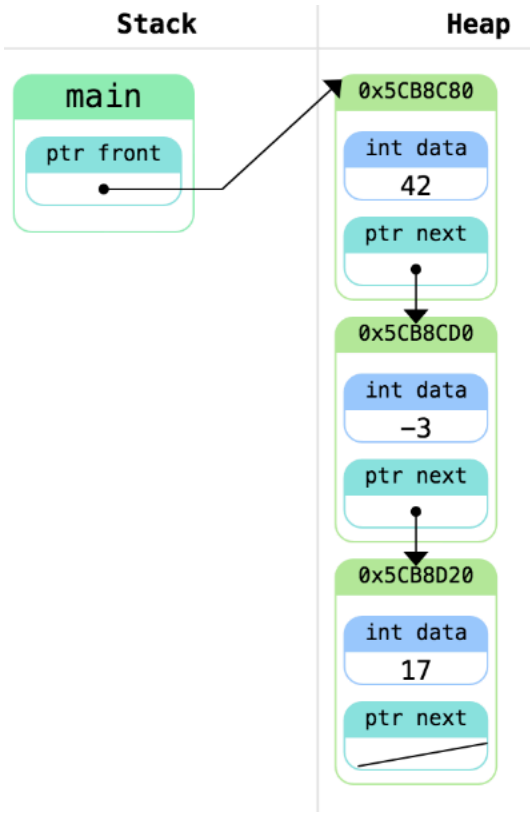
# Creating a Linked List

```
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;
```



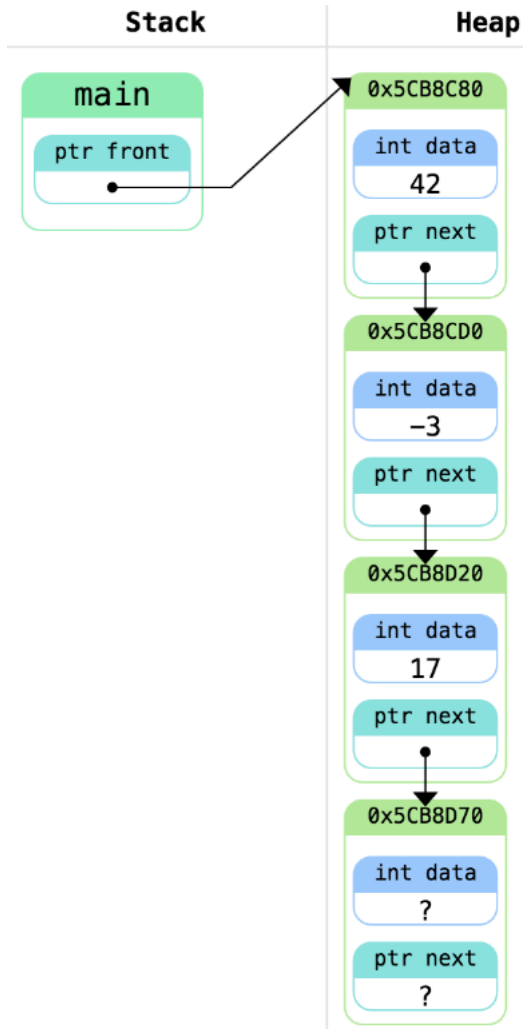
# Creating a Linked List

```
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;
```



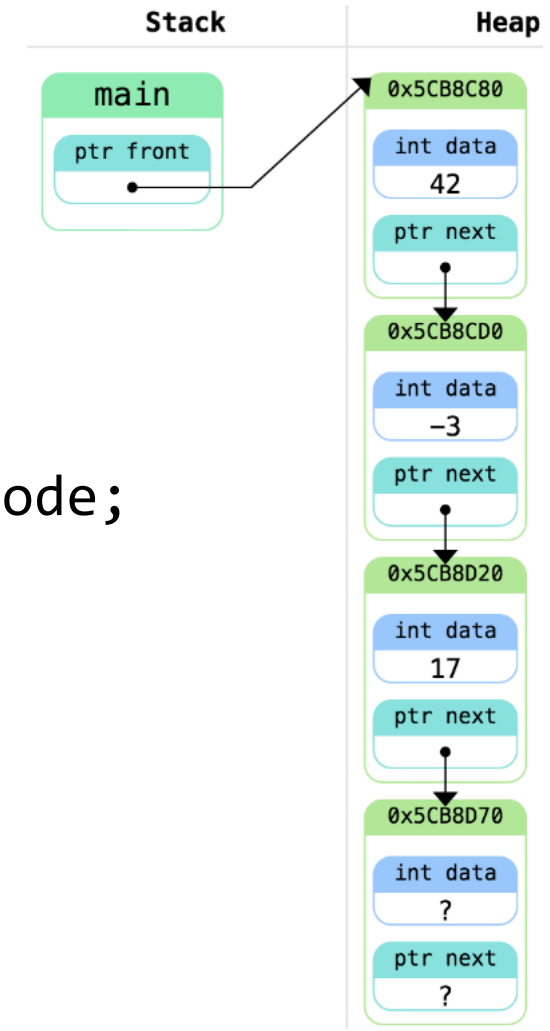
# No constructor?

```
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 = new ListNode;
```



# No constructor?

```
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 = new ListNode;
front->next->next->next->next = new ListNode;
    // KABOOM
```



# Announcements

- Assignment 4 is due on **Thursday** – please finish it before then
- You will get assignment 3 feedback on **Wednesday**
- Exam logistics
  - Midterm review session on Tuesday (tomorrow!), from 7:00-8:30PM, in Gates B01, led by SL Peter
  - Midterm is on Wednesday, July 25, from 7:00-9:00PM in Hewlett 200
  - Complete assignment 4 before the midterm – backtracking will be tested

# 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!

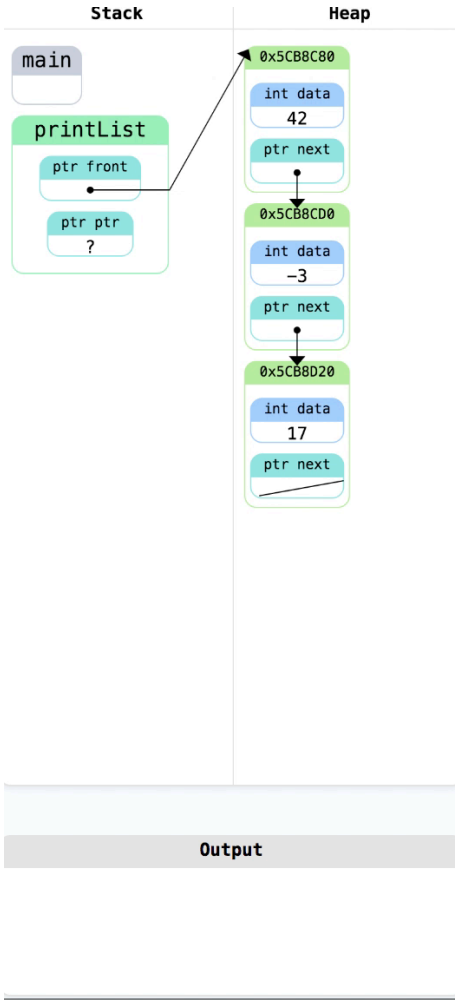
- Write a function that takes in the pointer to the front of a Linked List and prints out all the elements of a Linked List

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

# Iterative Trace

- 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

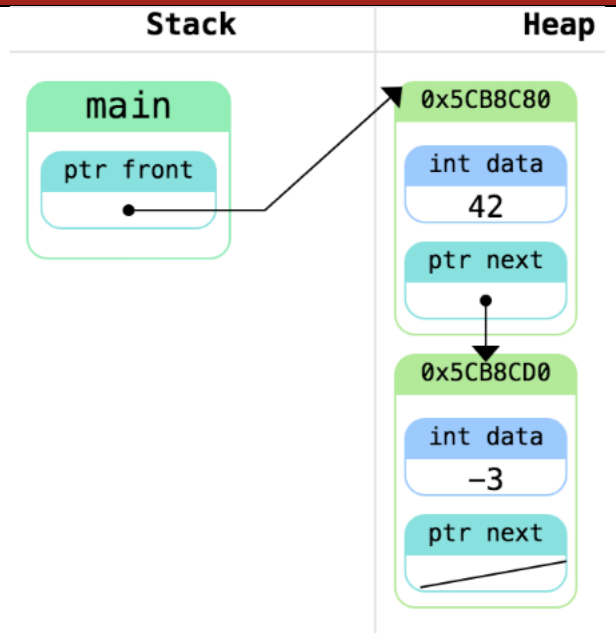
What's wrong?

```
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?

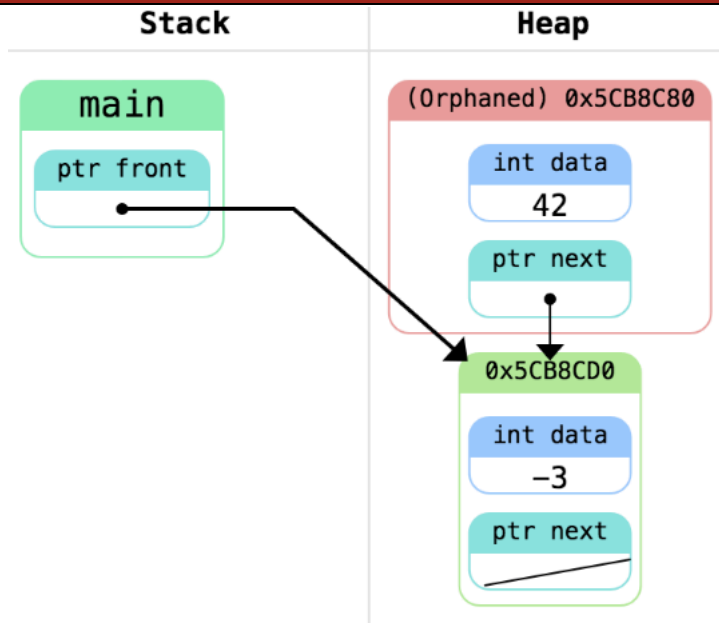
```
int main() {
  ListNode* front = new ListNode();
  front->data = 42;
  front->next = new ListNode();
  front->next->data = -3;
  front->next->next = nullptr;
  while (front != nullptr) {
    cout << front->data << " ";
    front = front->next;
  }
  // continue using front
  return 0;
}
```



# A Temporary Solution

What's wrong?

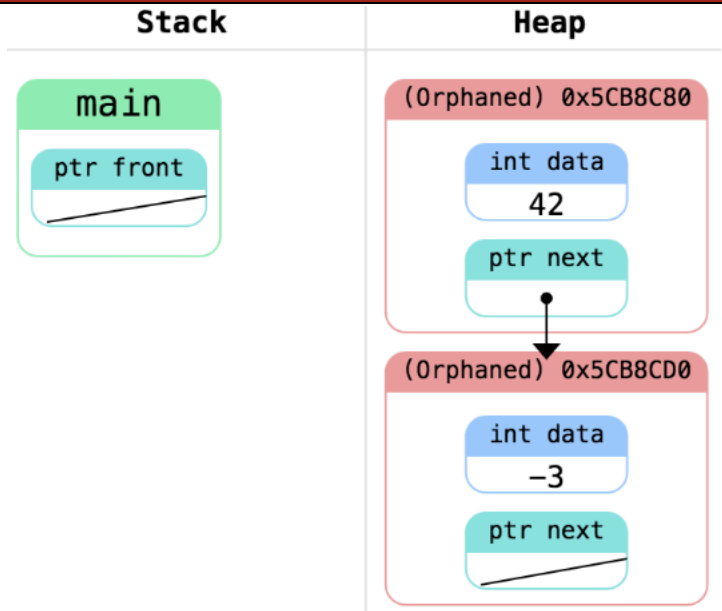
```
int main() {
  ListNode* front = new ListNode();
  front->data = 42;
  front->next = new ListNode();
  front->next->data = -3;
  front->next->next = nullptr;
   while (front != nullptr) {
     cout << front->data << " ";
     front = front->next;
  }
  // continue using front
  return 0;
}
```



# A Temporary Solution

What's wrong?

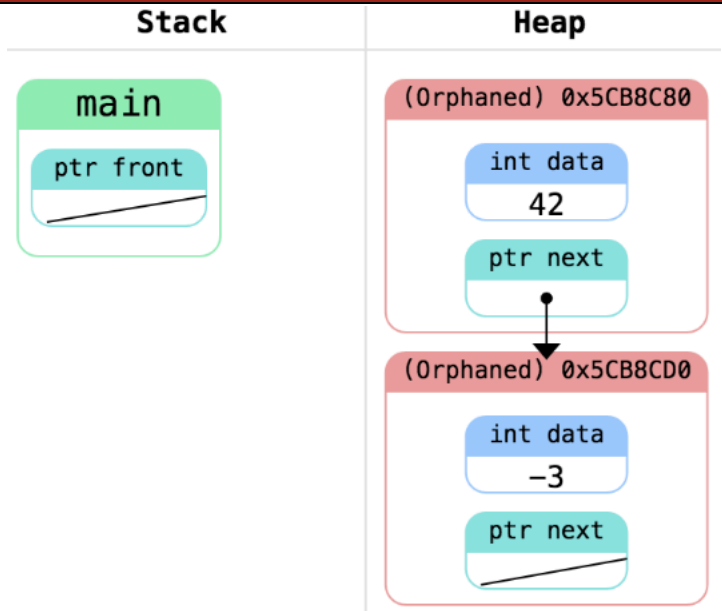
```
int main() {
  ListNode* front = new ListNode();
  front->data = 42;
  front->next = new ListNode();
  front->next->data = -3;
  front->next->next = nullptr;
   while (front != nullptr) {
     cout << front->data << " ";
     front = front->next;
  }
  // continue using front
  return 0;
}
```



# 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;
}
```