CS 106X, Lecture 14
Classes and Pointers

reading:

Programming Abstractions in C++, Chapter 6, 11

Plan For Today and Friday

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

Learning Goals

- Understand why classes are useful to encapsulate and abstract away logic.
- Understand why pointers and dynamic memory are necessary to implement a Linked List.
- Understand how to create our own classes, with unit tests.

Plan For Today and Friday

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

Classes

A class is a definition of your own custom variable type!

The Classes Checklist

- ❑ **Specify instance variables.** What information is inside this new variable type?
- ❑ **Specify public methods**. What can this variable type do for others?
- ❑ **Specify constructor(s).** How do you create a new variable of this type?

.h Files

```
// in ClassName.h
#pragma once
class ClassName {
public:
    ClassName(parameters);           // constructor
    returnType func1(parameters);    // member functions
    returnType func2(parameters);    // (behavior inside
    returnType func3(parameters);    //  each object)

private:
    type var1;     // member variables
    type var2;     // (data inside each object)
    type func4(); // (private function)
};
```

.cpp Files

- In ***ClassName*.cpp**, we write bodies (definitions) for the member functions that were declared in the ***.h*** file:

```
// ClassName.cpp
#include "ClassName.h"

// member function (may be multiple)
returnType ClassName::methodName(parameters) {
    statements;
}
```

  - Member functions/constructors can refer to the object's instance variables.

Example: BankAccount

- Let's define a new variable type that represents a bank account.
- You should be able to create one by specifying the account name and the initial balance.
- You should be able to deposit and withdraw money, which should return whether or not that action was successful. You should also be able to update the account name.
- You should also be able to obtain the account balance and name.

BankAccount.h

```
class BankAccount {
public:
    // Step 3: how to create a BankAccount
    BankAccount(string accountName, double startBalance);

    // Step 2: the things a BankAccount can do
    bool withdraw(double amount);
    bool deposit(double amount);
    double getBalance();
    string getName();
private:
    // Step 1: the data inside a BankAccount
    string name;
    double balance;
};
```

BankAccount.cpp

```
#include “BankAccount.h”

bool BankAccount::withdraw(double amount) {
    if (amount <= balance && amount >= 0) {
      balance -= amount;
       return true;
    }
    return false;
}

bool BankAccount::deposit(double amount) {
    if (amount >= 0) {
      balance += amount;
       return true;
    }
    return false;
}
...
```

BankAccount.cpp

```
double BankAccount::getBalance() {
    return balance;
}

double BankAccount::getName() {
    return name;
}
```

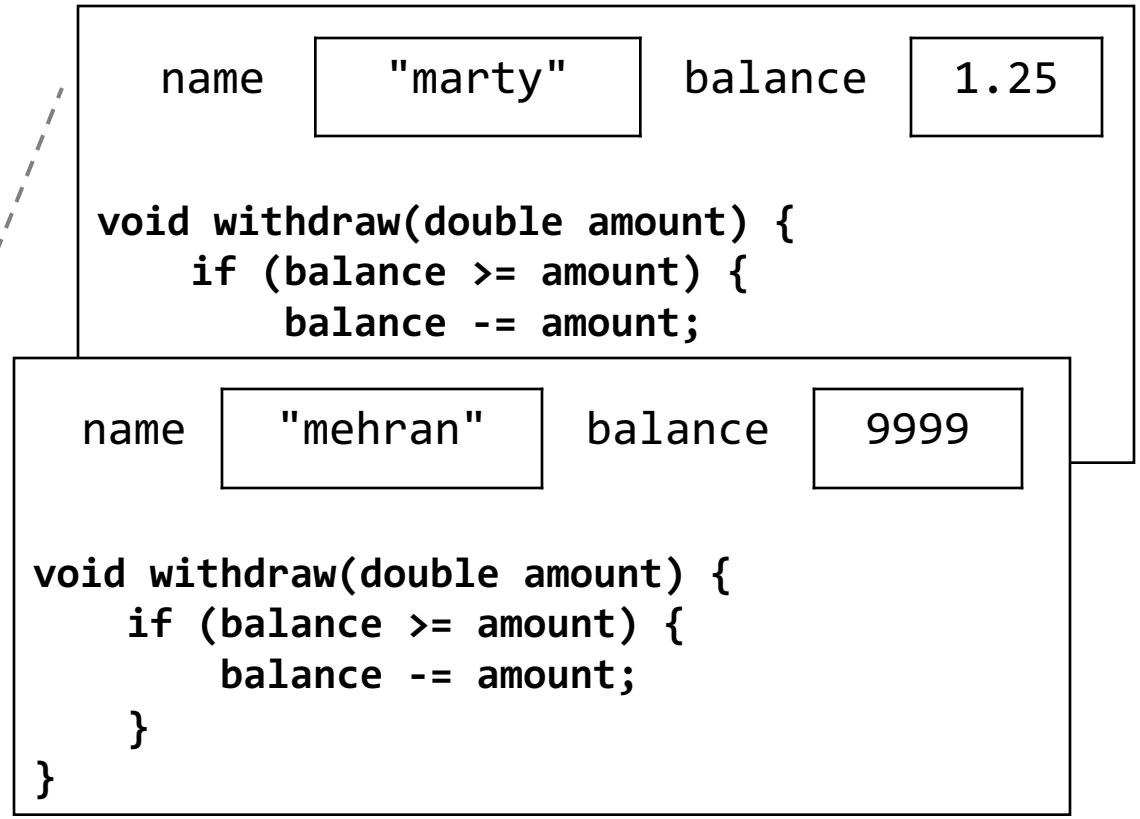
The implicit parameter

- **implicit parameter**:
  The object on which a member function is called.
  - During the call `marty.withdraw(...)`,
    the object named `marty` is the implicit parameter.
  - During the call `mehran.withdraw(...)`,
    the object named `mehran` is the implicit parameter.
  - The member function can refer to that object's member variables.
    - We say that it executes in the *context* of a particular object.
    - The function can refer to the data of the object it was called on.
    - It behaves as if each object has its own *copy* of the member functions.

Member func diagram

```
// BankAccount.cpp
void BankAccount::withdraw(double amount) {
    if (balance >= amount) {
        balance -= amount;
    }
}
```

```
// client program
BankAccount marty;
BankAccount mehran;
...
marty.withdraw(5.00);

mehran.withdraw(99.00);
```



Constructors

```
// Constructor
BankAccount::BankAccount(string accountName, double
                         startBalance) {
    name = accountName;
    balance = startBalance;
}

...
```

The keyword this

- C++ has a `this` keyword to refer to the current object.
  - Syntax: `this->`***`member`***
  - *Common usage:* In constructor, so parameter names can match the names of the object's member variables:

```
BankAccount::BankAccount(string name,
                         double balance) {
    this->name = name;
    this->balance = balance;
}
```

`this` uses `->` not `.` because it is a *pointer* to the current object

The keyword const

- C++ **`const` keyword** indicates that a value cannot change.

```
const int x = 4;                    // x will always be 4
```

- a **`const` reference parameter** can't be modified by the function:

```
void foo(const BankAccount& ba) {   // won't change ba
```

  - Any attempts to modify d inside `foo`'s code won't compile.

- a **`const` member function** can't change the object's state:

```
class BankAccount { ...
    double getBalance() const;  // won't change account
```

  - On a `const` reference, you can only call `const` member functions.

Static data

- **static:** Shared by all objects of a class.
  - Opposite of regular member, which are duplicated in each object.
  - Useful when a class has some class-global shared state.

```
// BankAccount.h
class BankAccount {
    ...
private:
    static int ACCOUNT_ID = 1;
};
```

Class constants

- **class constant**: An unmodifiable static variable in the .h file.
  - Assign its value in the .cpp, outside of any method.
    - Don't write `static` when assigning the value in the .cpp.
  - For integral types, you can actually assign the variable in the .h file.

```
// BankAccount.h
class BankAccount {
    static const int BANK_ROUTING_NUM = 006029593;
    static const double INTEREST_RATE;
};

// BankAccount.cpp
// set the constant to store 3.25%
const double BankAccount::INTEREST_RATE = 0.0325;
```

Plan For Today and Friday

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

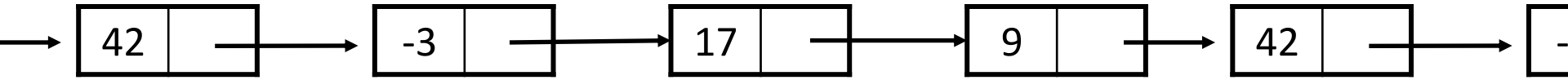
Announcements

- Midterm Exam is **Thurs. 11/1 7-9PM in 420-040**
  - Covering material through unit testing on Mon. 10/22
  - Open-book, closed note (reference sheet provided)
  - Administered via BlueBook software (on your laptop)
  - Practice materials and BlueBook download available on Friday
  - Review session **Tues. 10/30 5-6:30PM** in Hewlett 102
  - If you have a university or academic conflict, you **must** let us know by **tomorrow (Thurs. 10/25) @ 5PM**
  - If you have academic accommodations, e.g. through OAE, please let us know by **tomorrow (Thurs. 10/25) @5PM** if possible.
  - If you do not have a workable laptop for the exam, you **must** let us know by **Friday 10/26 @ 5PM**. Limited charging outlets will be available for those who need them during the exam.

Plan For Today and Friday

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

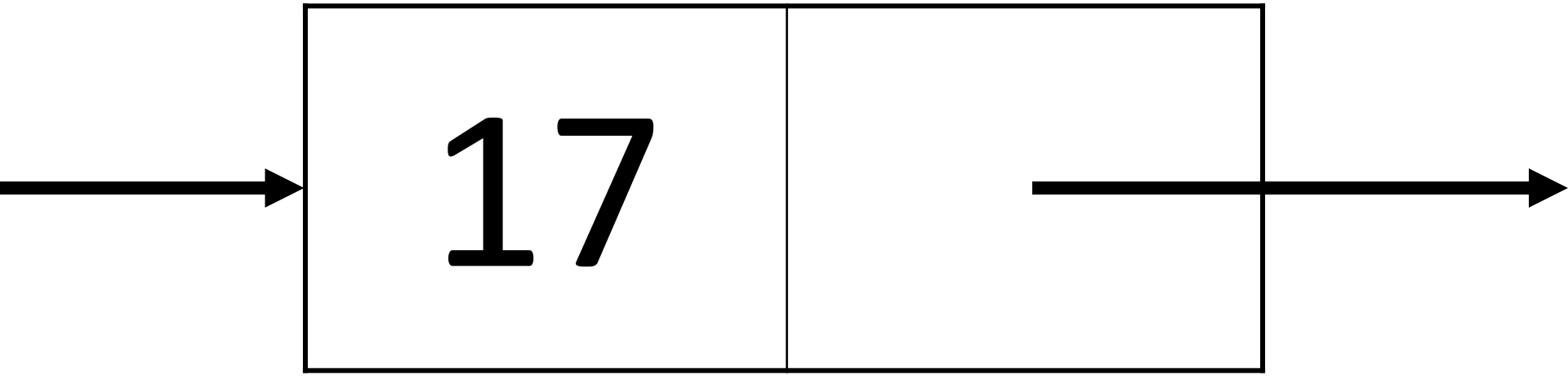
Linked Lists 🔗



(+) Fast to add/remove at any point

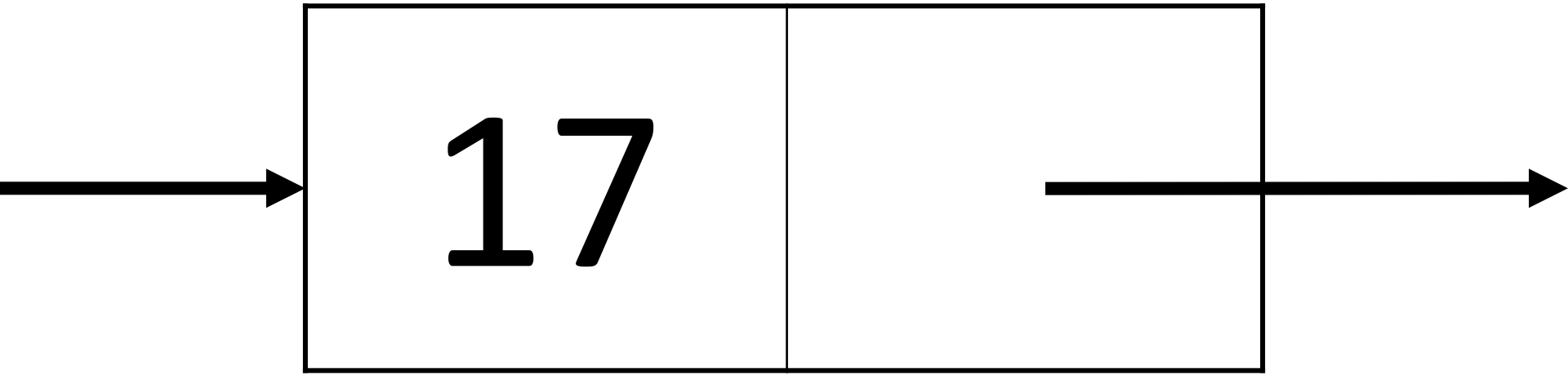
(–) Slow to access certain nodes

Nodes



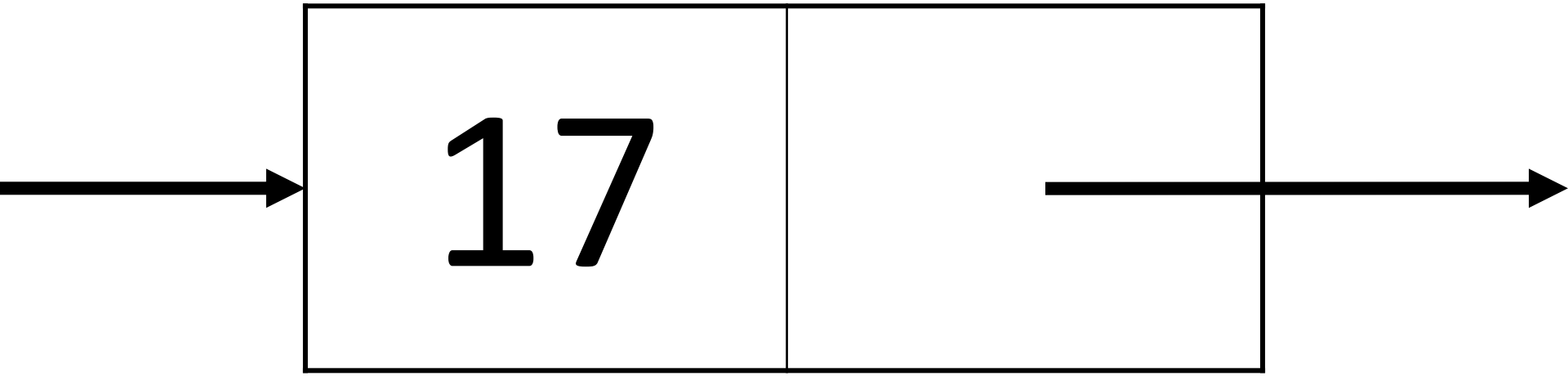

```
struct Node {
    ??? data;
    ??? next;
};
```

Nodes



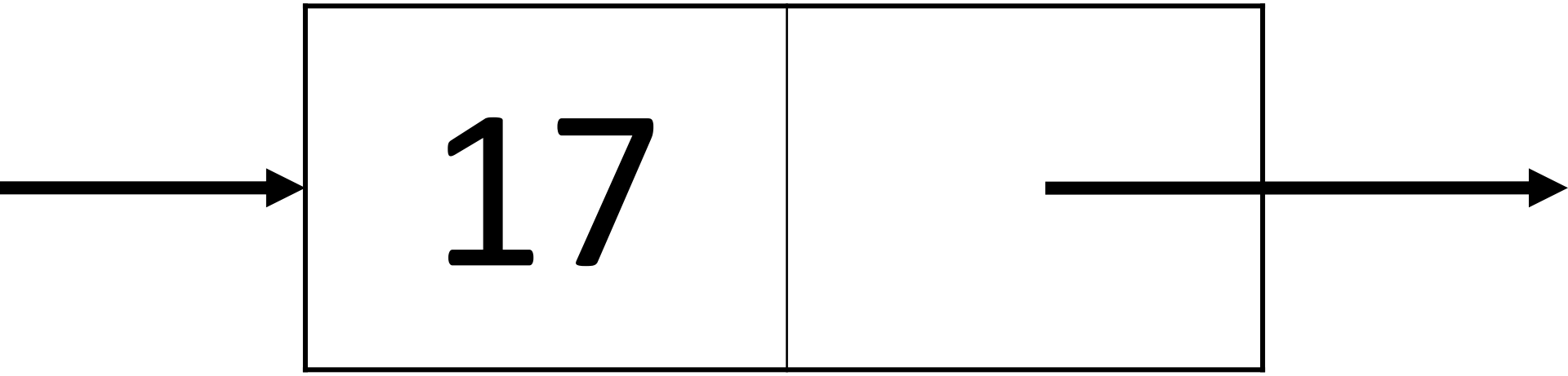

```
struct Node {
    int data;
    ??? next;
};
```

Nodes




```
struct Node {
    int data;
    Node next;
};
```

Nodes

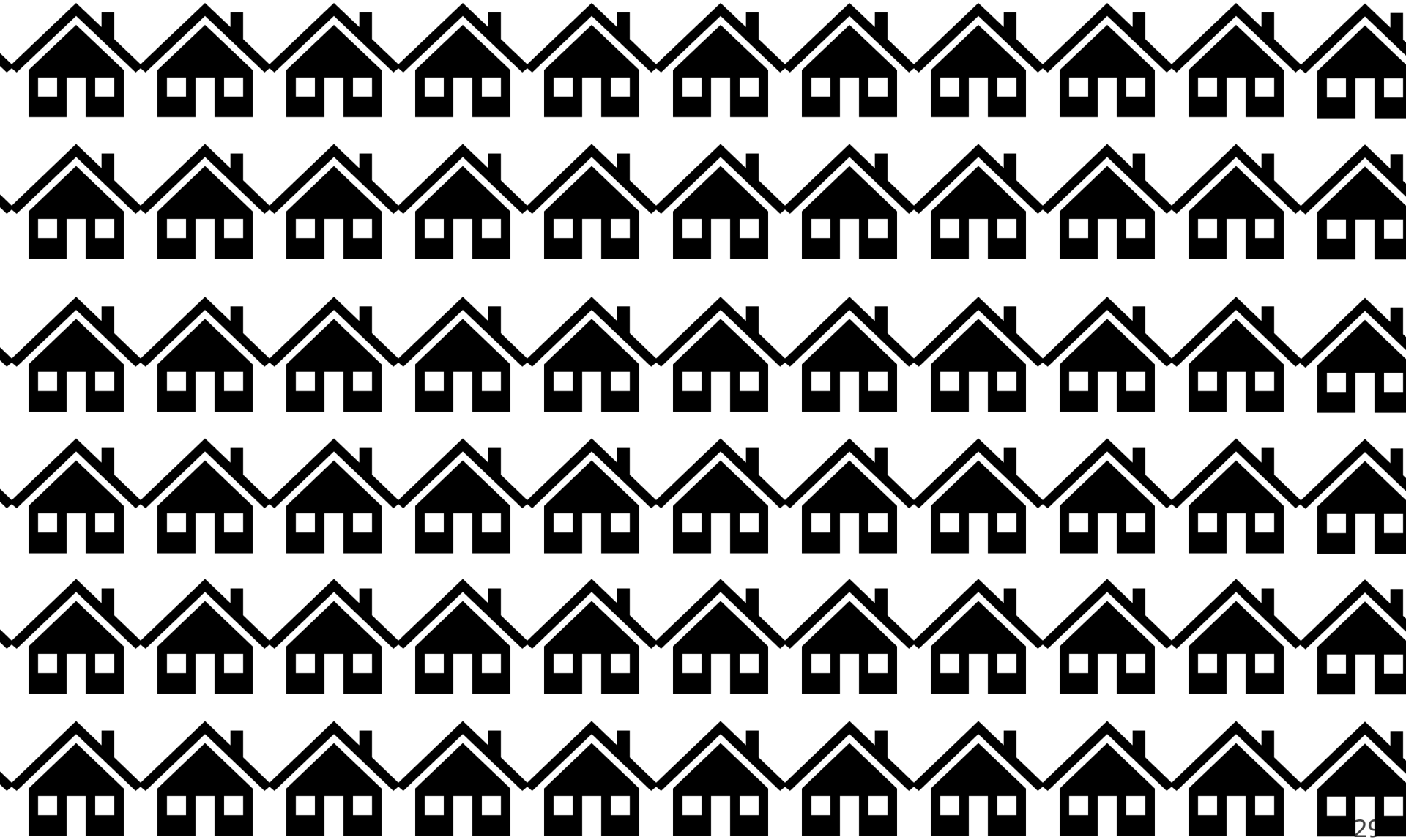



```
struct Node {
    int data;
    Node next;
};
```

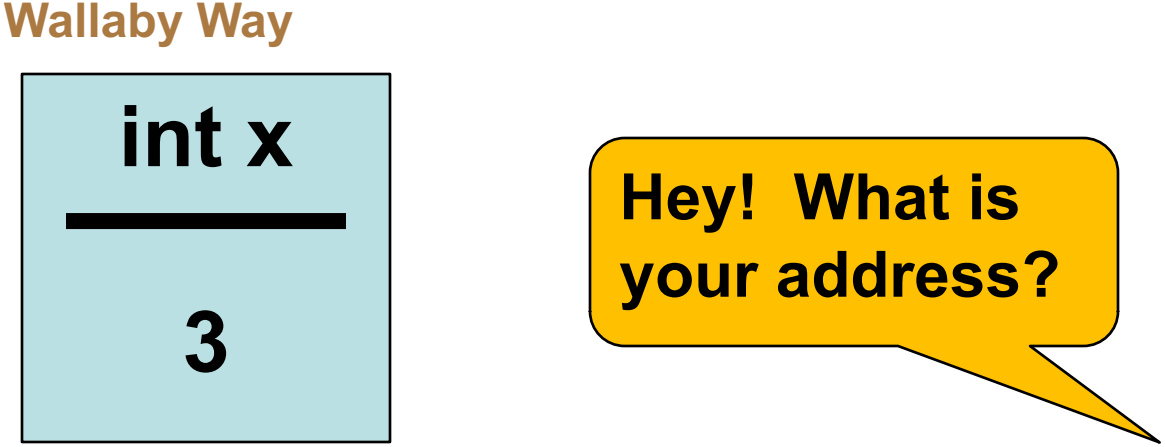
This would be infinitely recursive!

Addresses

Addresses



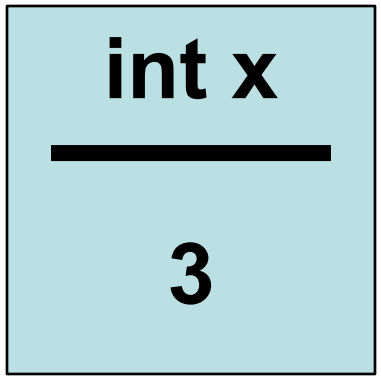
Addresses




```
int x = 3;
cout << &x << endl;
```

Addresses

42 Wallaby Way




```
int x = 3;
cout << &x << endl;
```

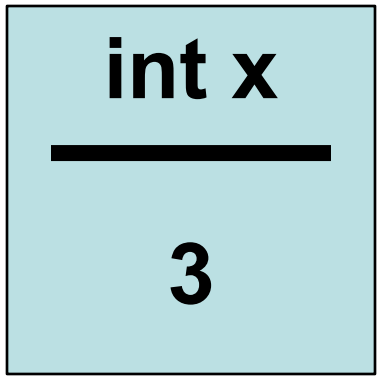
The `&` operator is the **address of** operator. It gets the address of a variable in memory.

Plan For Today and Friday

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

Addresses

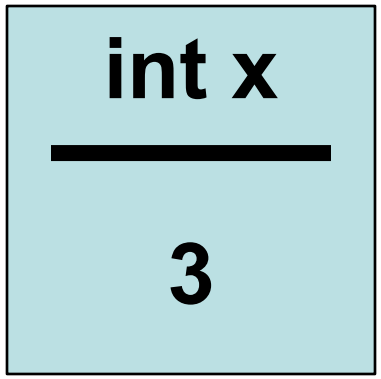
42 Wallaby Way




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

Addresses

42 Wallaby Way

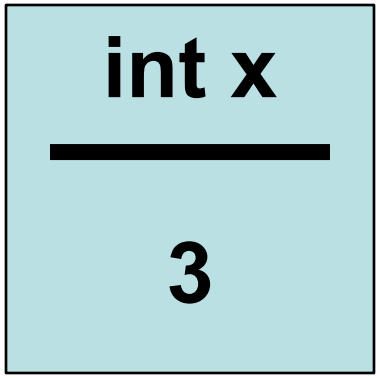



```
int x = 3;
int *xAddress = &x;
```

This is a variable named **`xAddress`**…

Addresses

42 Wallaby Way

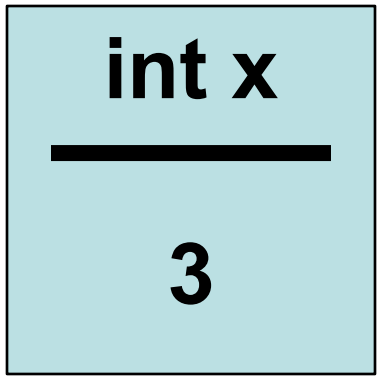



```
int x = 3;
int *xAddress = &x;
```

That stores the address of an `int`…

Addresses

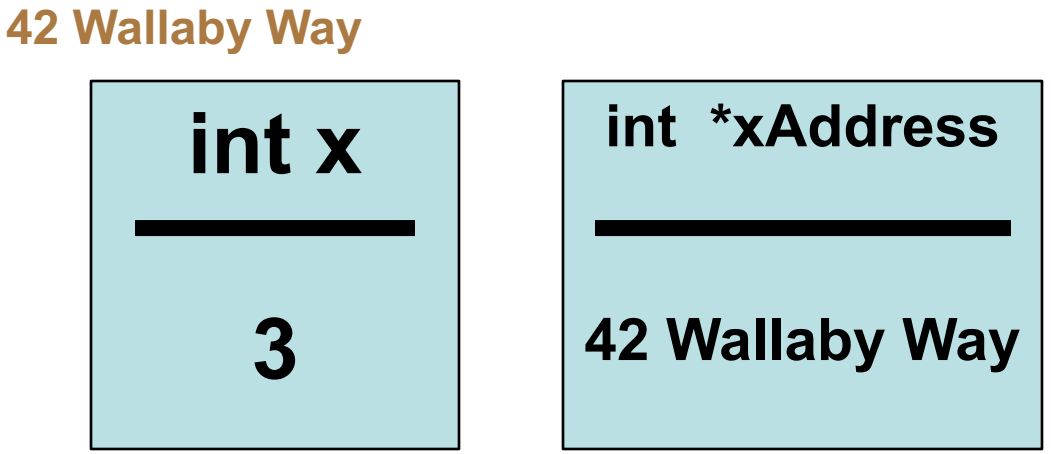
42 Wallaby Way




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

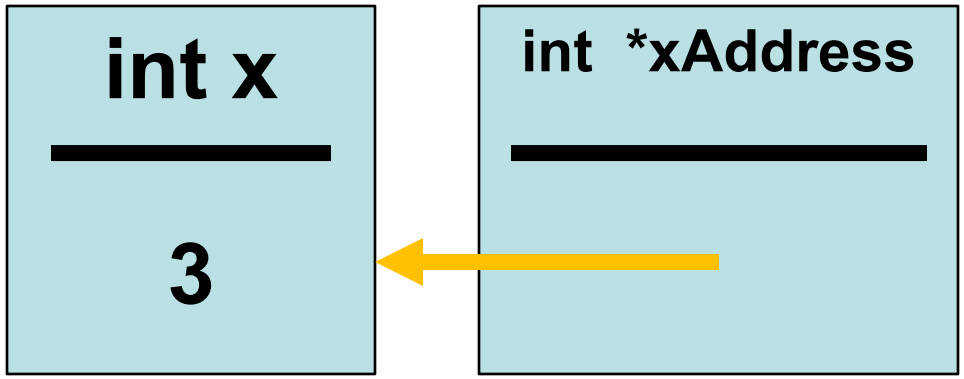
And its value should be the address of **x**.

Addresses




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

Addresses




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

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;
```

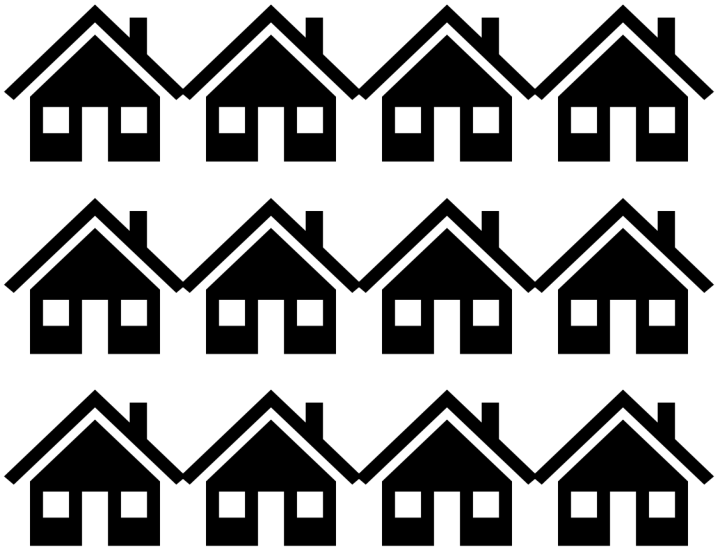
x is the **pointee** of **xAddress**. It is being pointed to by **xAddress**.

Dereferencing

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

*xAddress = 5;
```

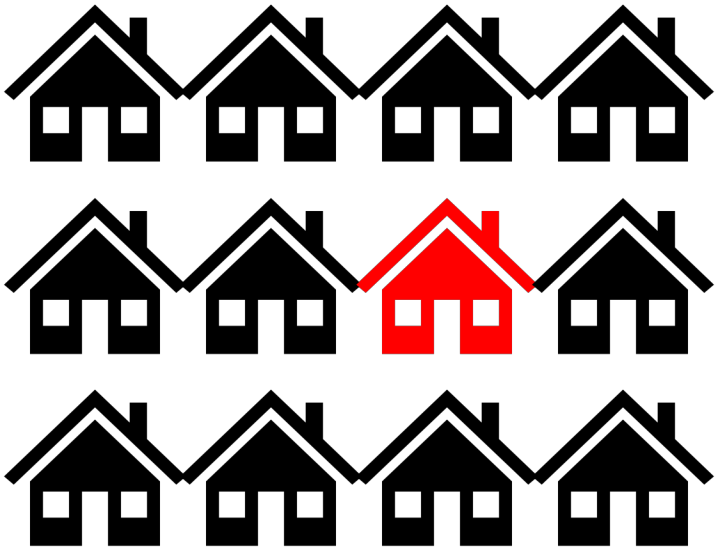
Dereferencing

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

*xAddress = 5;
```



Dereferencing

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

*xAddress = 5;
```

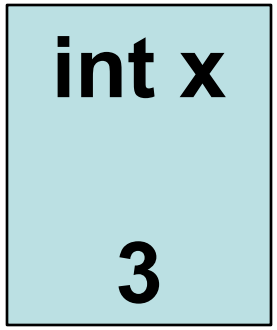


Dereferencing

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

*xAddress = 5;
```

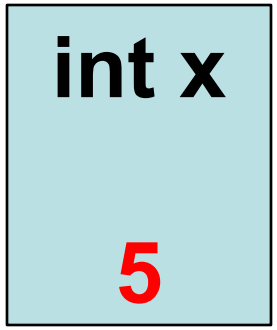
Dereferencing

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

*xAddress = 5;
```



Dereferencing

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

*xAddress = 5;
```



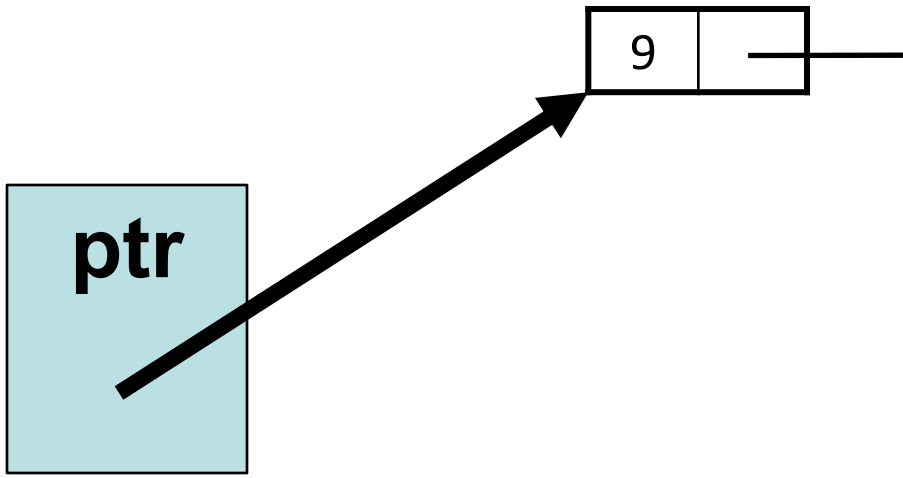
Dereferencing

```
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.

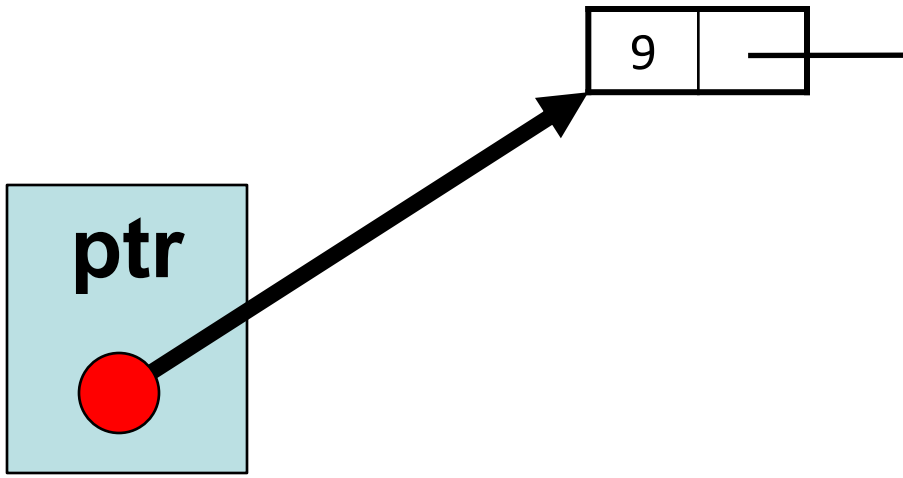
Dereferencing Structs

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

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



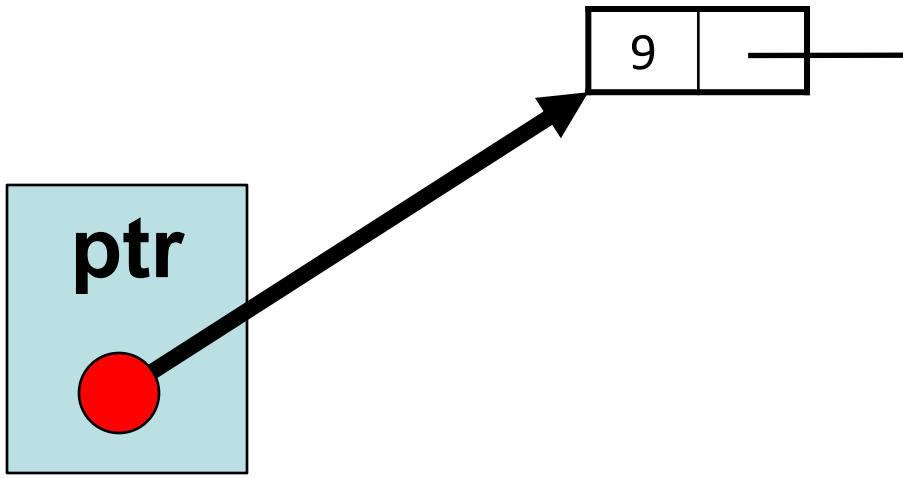
Dereferencing Structs

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

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



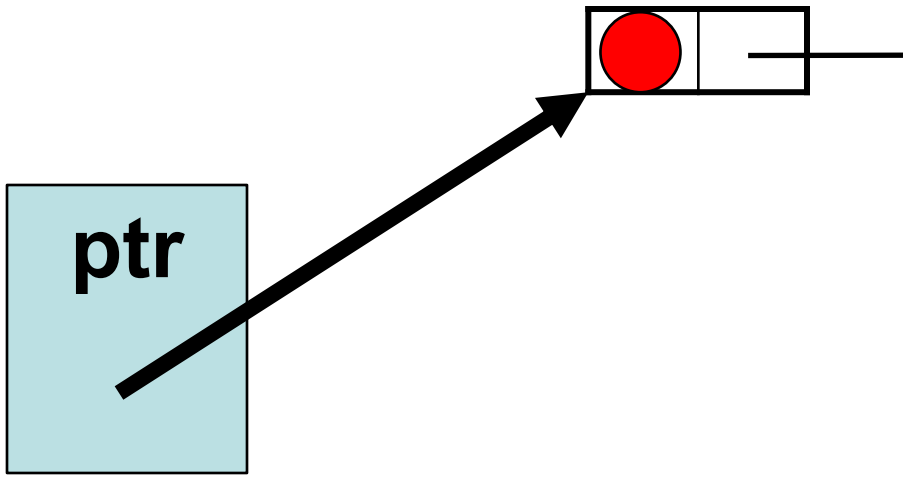
Dereferencing Structs

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

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



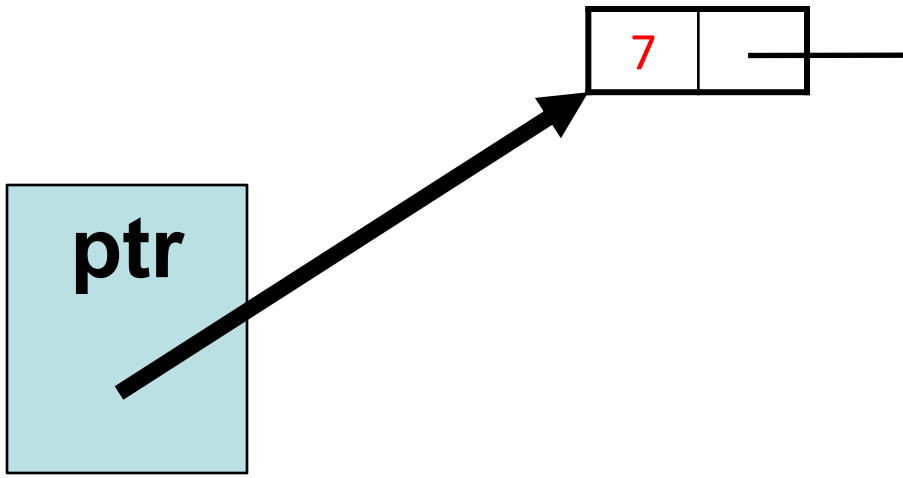
Dereferencing Structs

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

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



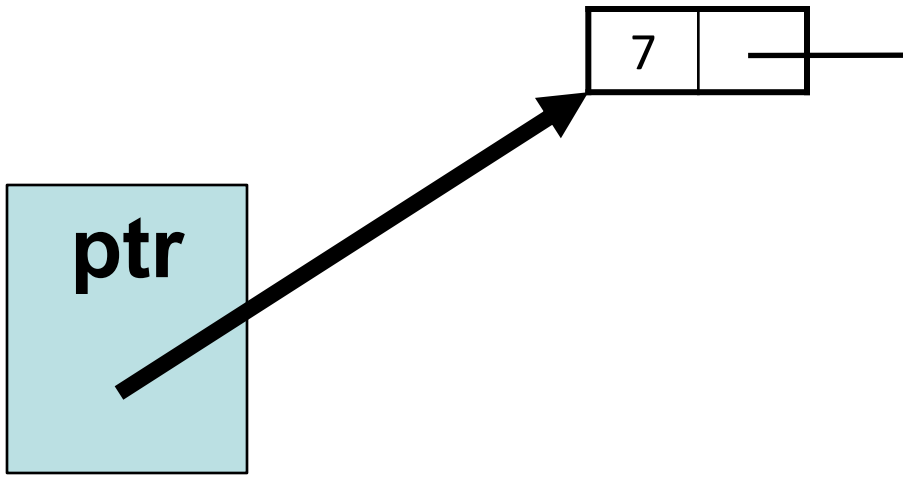
Dereferencing Structs

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

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

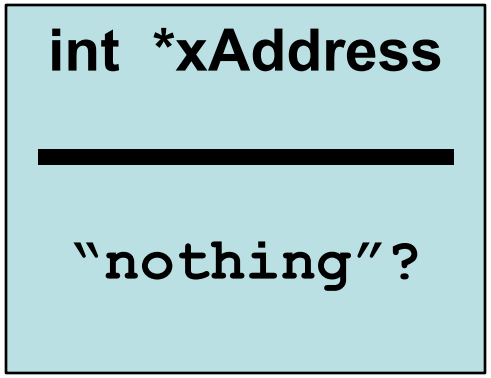


Dereferencing Structs

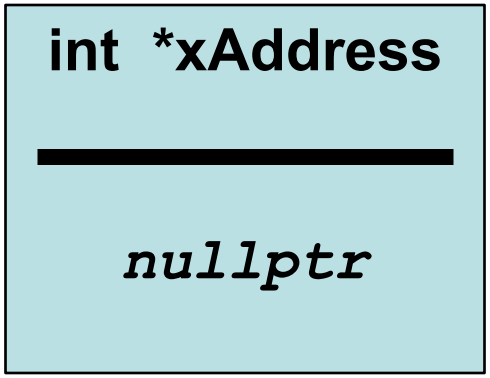
```
Node n = ...
Node *ptr = &n;

ptr->value = 7;
```



Much Ado About Nothing

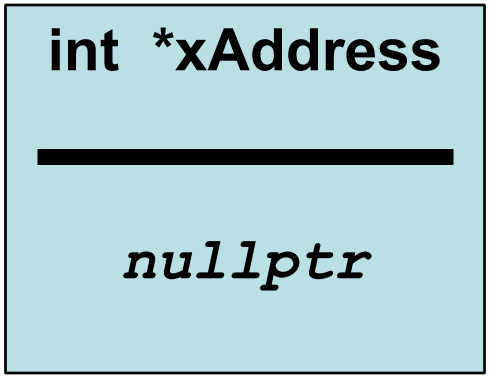


Dereferencing nullptr




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

Dereferencing nullptr

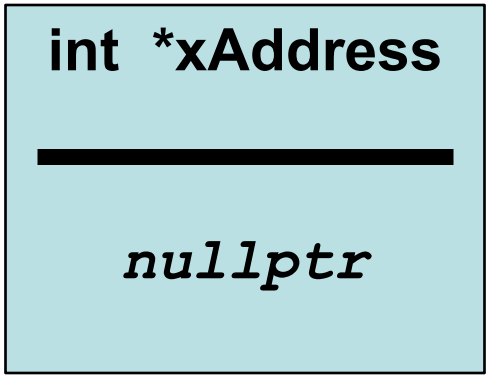



```
int *xAddress = nullptr;
cout << *xAddress << endl;
```

Console

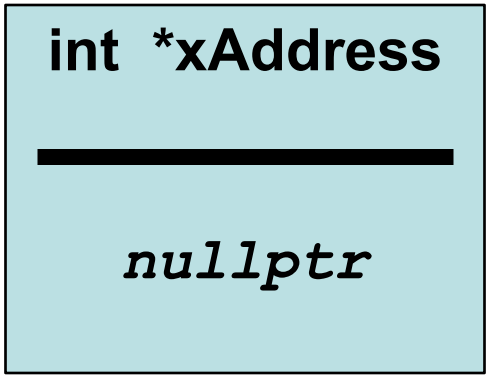
```
...

***
*** 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()
```

Dereferencing nullptr




```
int *xAddress = nullptr;
if (xAddress != nullptr) {
    cout << *xAddress << endl;
} else {
    cout << "nullptr!" << endl;
}
```

Dereferencing nullptr




```
int *xAddress = nullptr;
if (xAddress) {
    cout << *xAddress << endl;
} else {
    cout << “nullptr!” << endl;
}
```

Garbage Pointers 


```
int *xAddress; // initially garbage
```

Garbage Pointers


```
int *xAddress; // initially garbage
cout << xAddress << endl; // ???
```

Garbage Pointers 

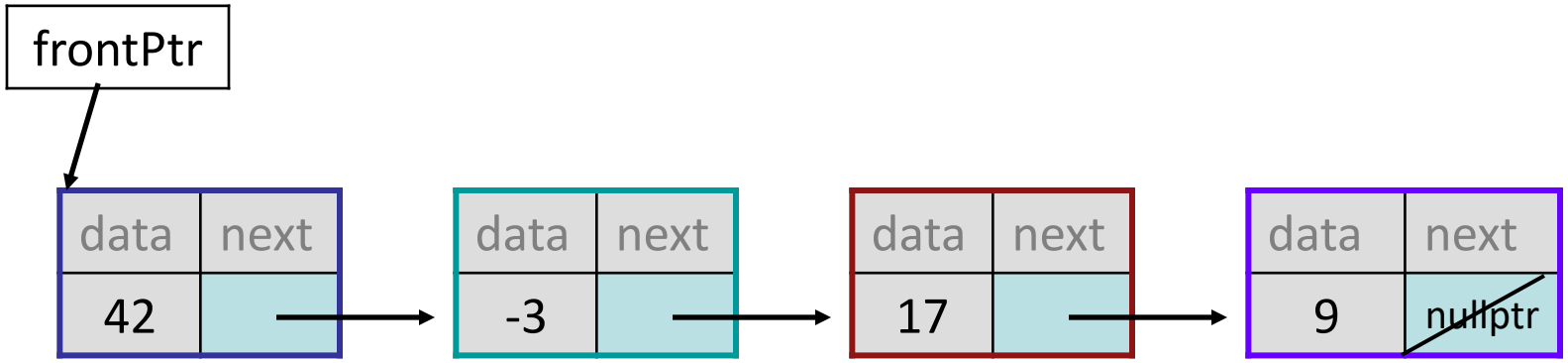

```
int *xAddress; // initially garbage
cout << xAddress << endl; // ???
cout << *xAddress << endl; // likely crash!
```

Garbage Pointers


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


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

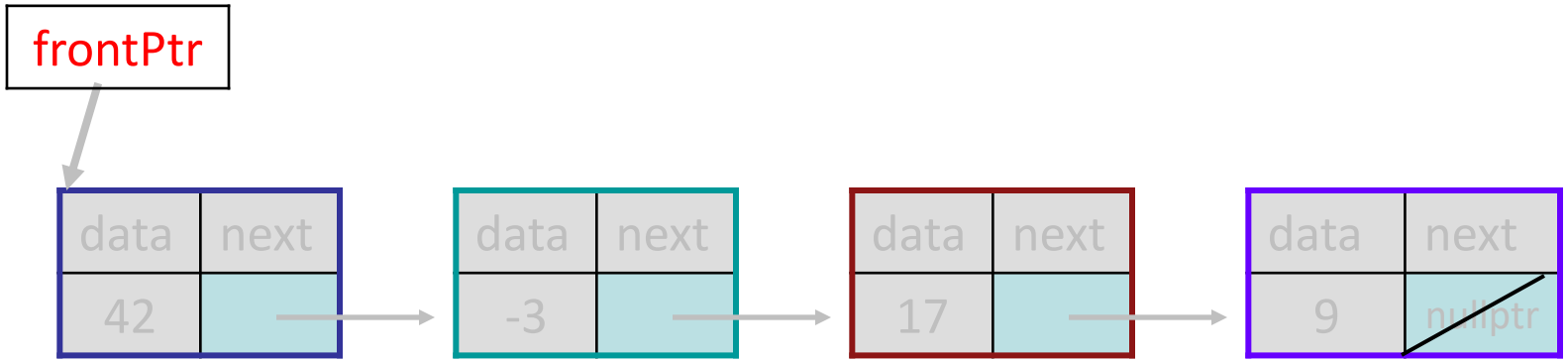
Using a Linked List

```
ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?

frontPtr->next->next->next->data = 2;
```



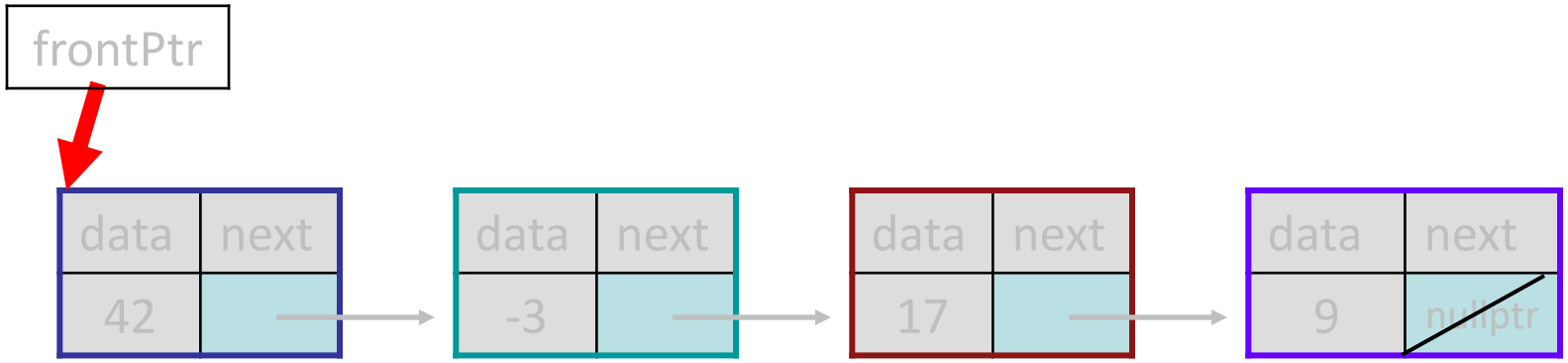
Using a Linked List

```
ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?

frontPtr->next->next->next->data = 2;
```



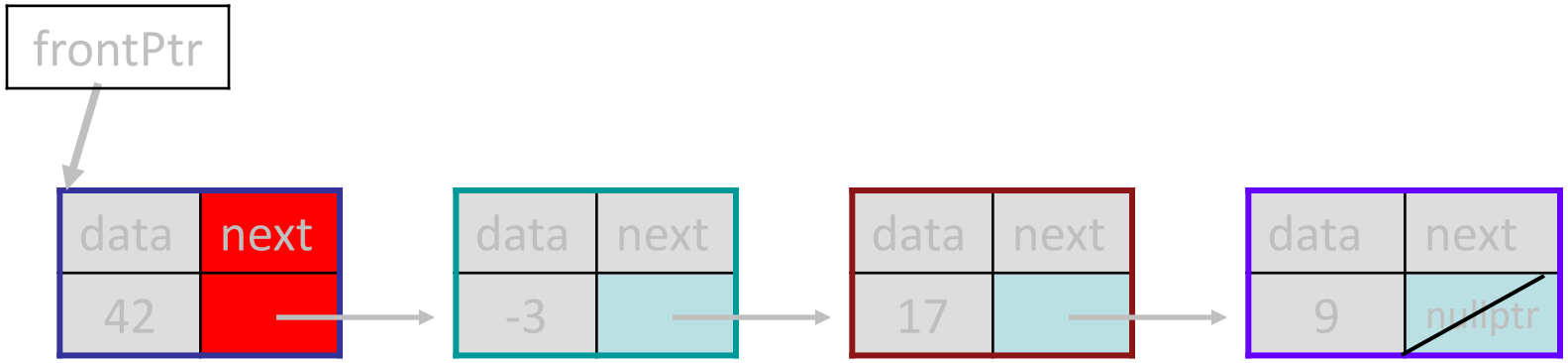
Using a Linked List

```
ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?

frontPtr->next->next->next->data = 2;
```



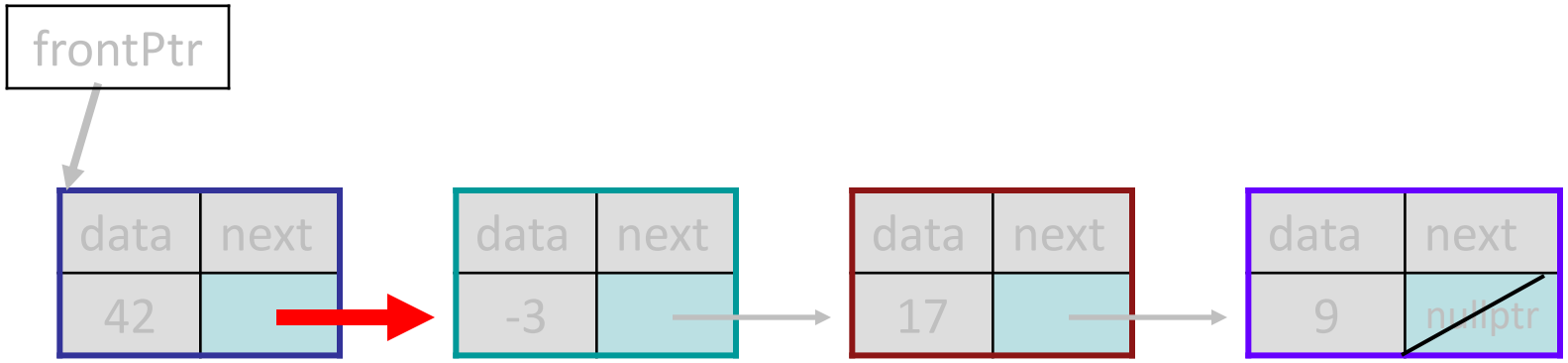
Using a Linked List

```
ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?

frontPtr->next->next->next->data = 2;
```



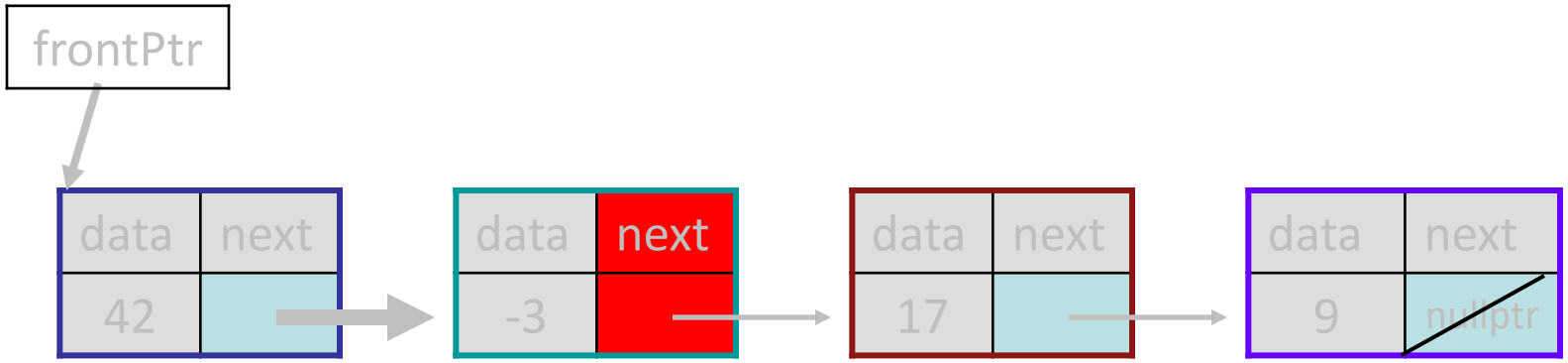
Using a Linked List

```
ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?

frontPtr->next->next->next->data = 2;
```



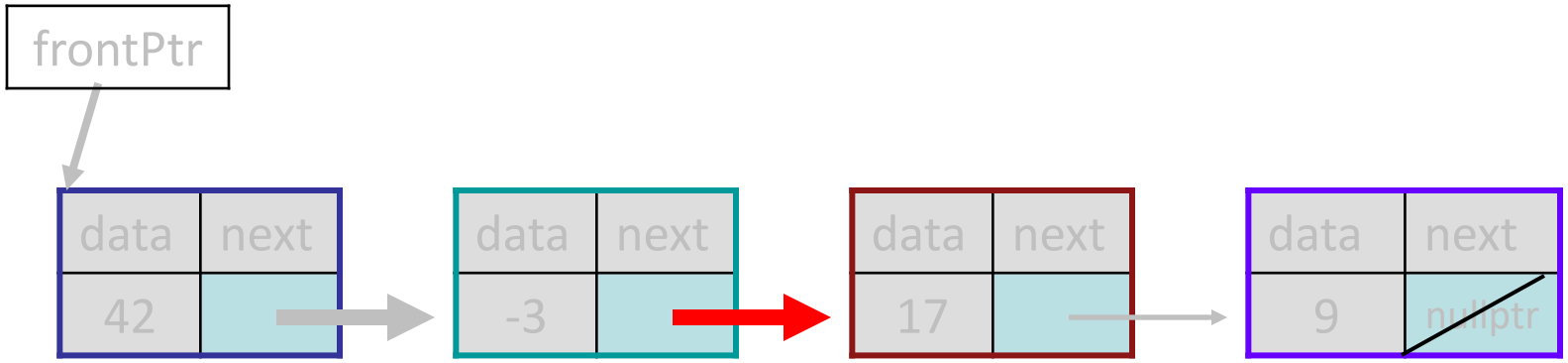
Using a Linked List

```
ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?

frontPtr->next->next->next->data = 2;
```



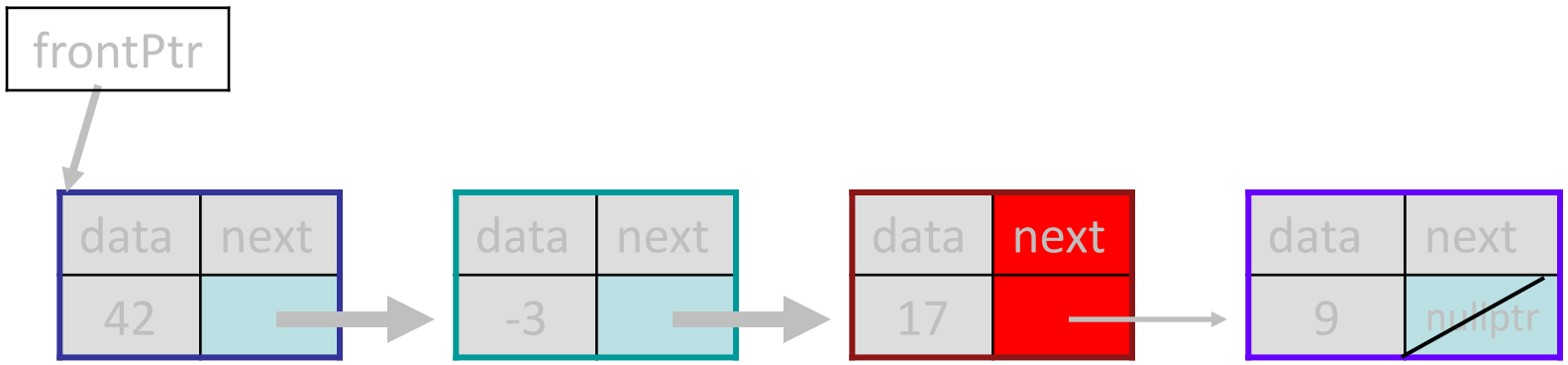
Using a Linked List

```
ListNode *frontPtr = ...t;
// How do we e.g. modify the data in the fourth node?

frontPtr->next->next->next->data = 2;
```



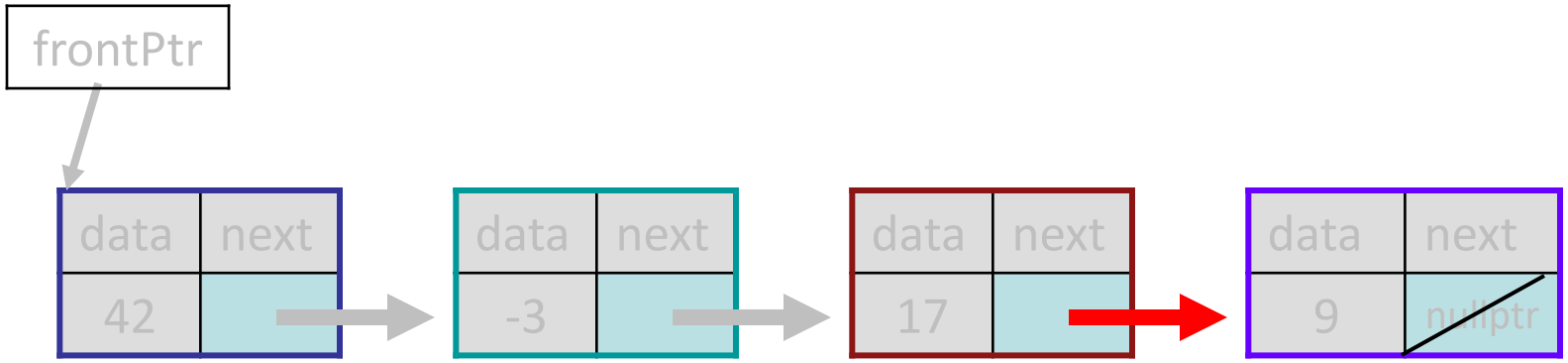
Using a Linked List

```
ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?

frontPtr->next->next->next->data = 2;
```



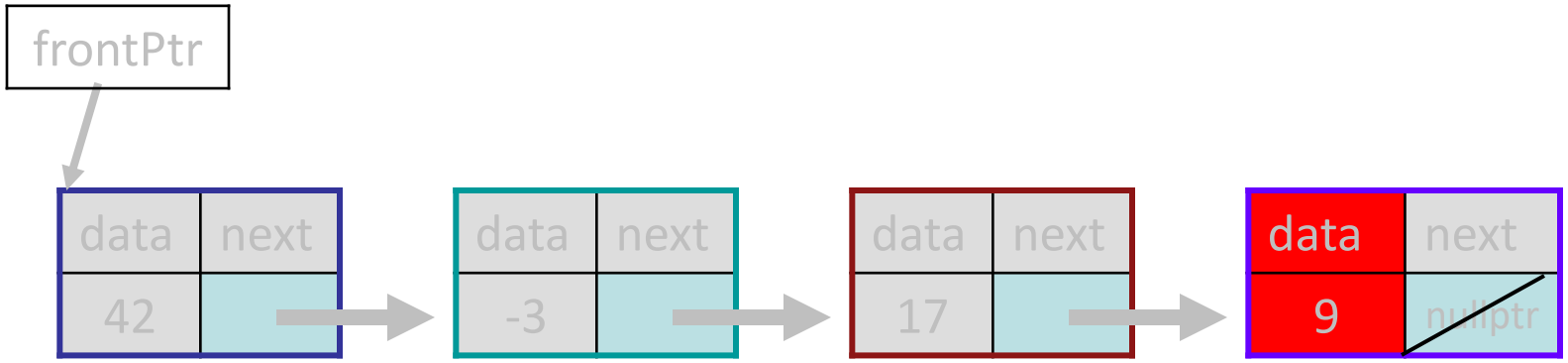
Using a Linked List

```
ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?

frontPtr->next->next->next->data = 2;
```



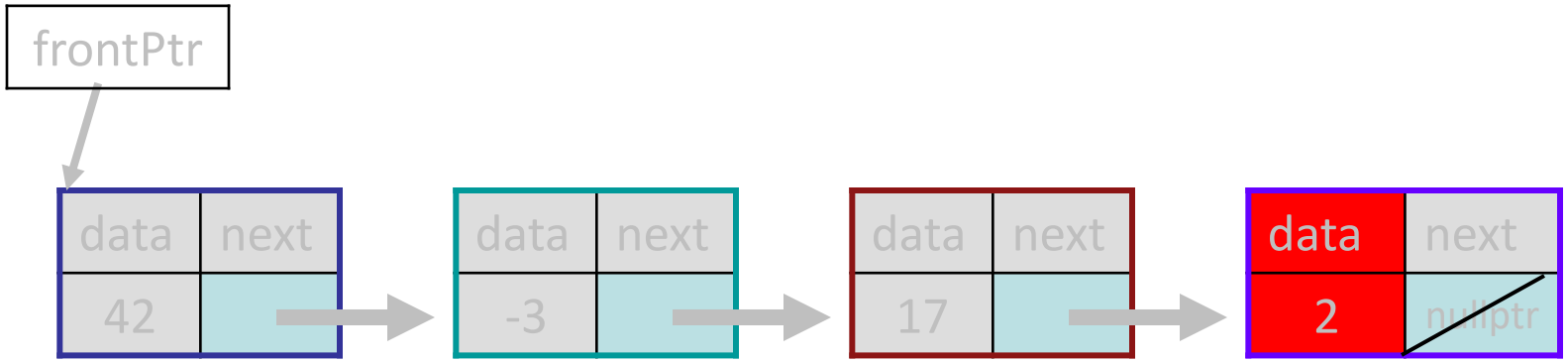
Using a Linked List

```
ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?

frontPtr->next->next->next->data = 2;
```

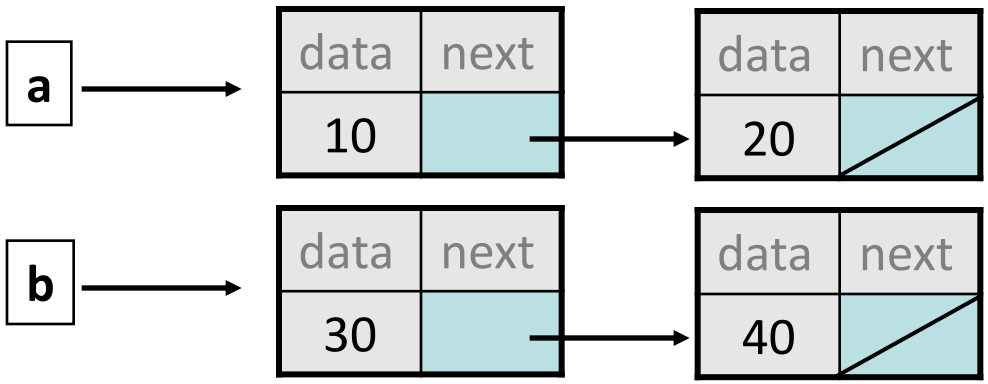


Using a Linked List

```
ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?

frontPtr->next->next->next->data = 2;
```

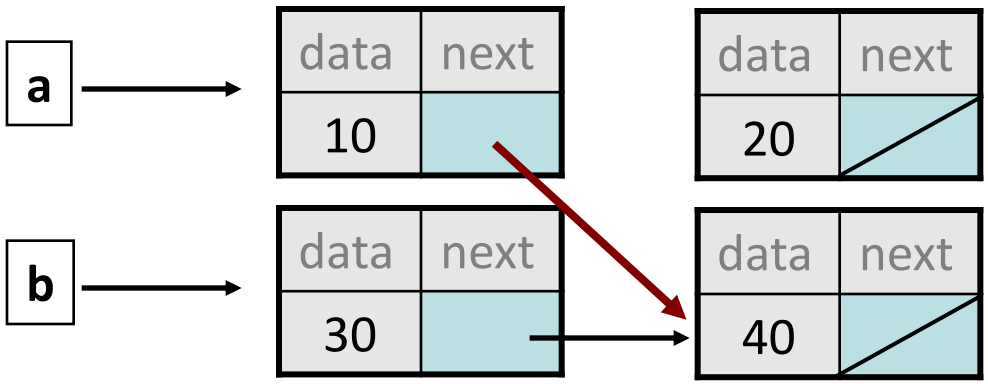


Reassigning Pointers




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

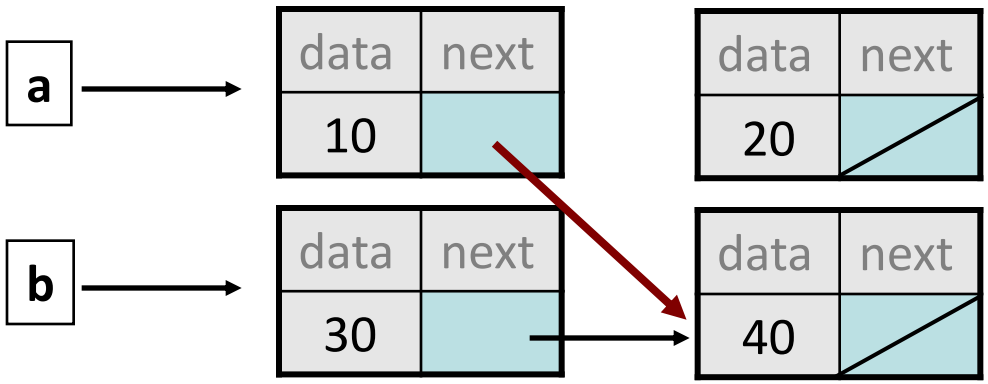
Reassigning Pointers




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

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

Reassigning Pointers

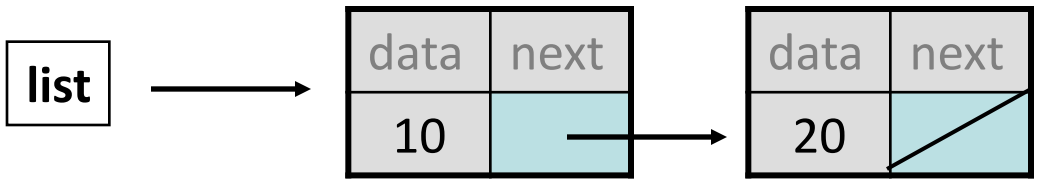


~~a->next = firstNode;~~

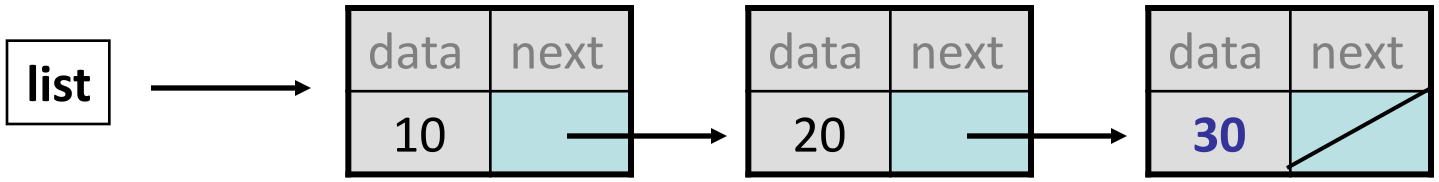
Tip: the types on the left- and right-hand sides must always match!

Linked node problem 1

- Which statement turns this picture:



- Into this?




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

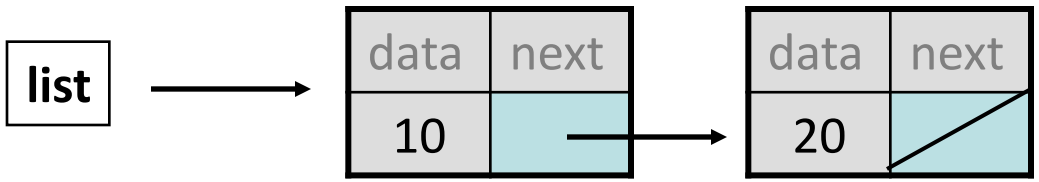
A. `list->next = node;`

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

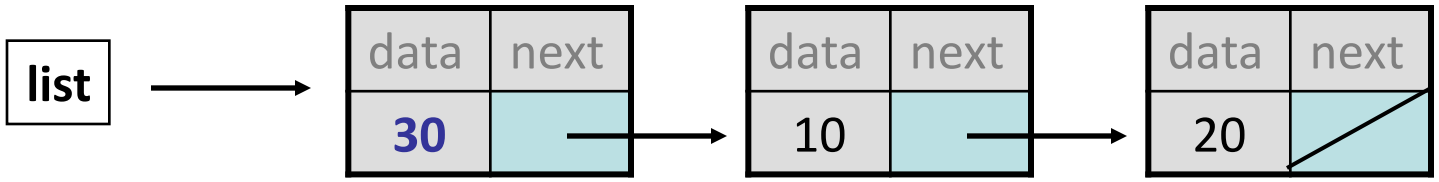
C. `list->next->next->next = node;`

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;`

Pass By Reference

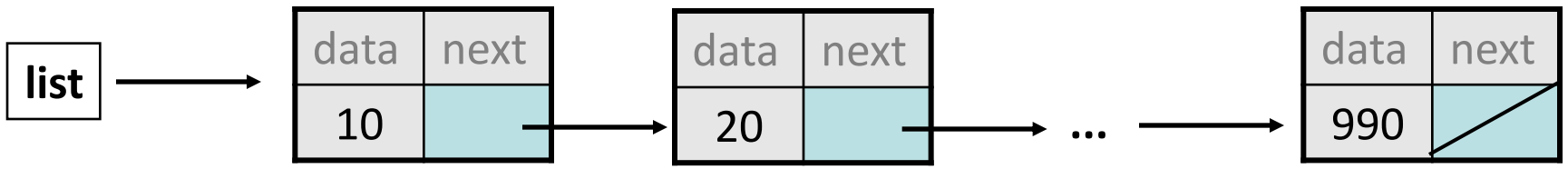
```
int main() {
    int x = 0;
    addTwo(x);
    cout << x << endl;    // 2
}
void addTwo(int& x) {
    x += 2;
}
```

Pass By Reference

```
int main() {
    int x = 0;
    addTwo(&x);
    cout << x << endl;    // 2
}
void addTwo(int *x) {
    *x += 2;
}
```

Pass-by-reference is implemented using pointers! It is an "automatically-dereferenced" pointer.

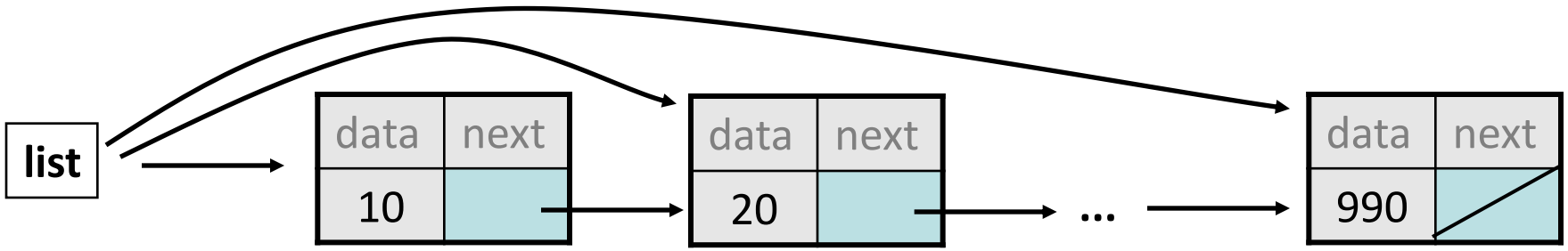
Traversing a Linked List



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

Traversing a list?

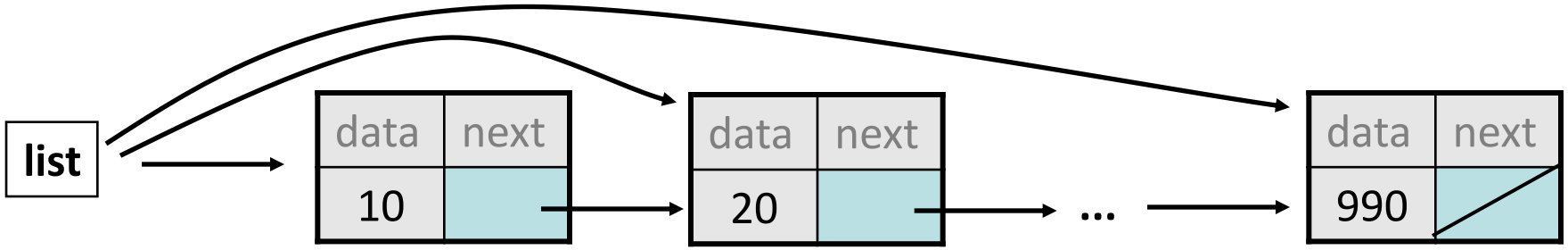
```
while (list != nullptr) {
    cout << list->data << endl;
    list = list->next;    // move to next node
}
```



Traversing a list?

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

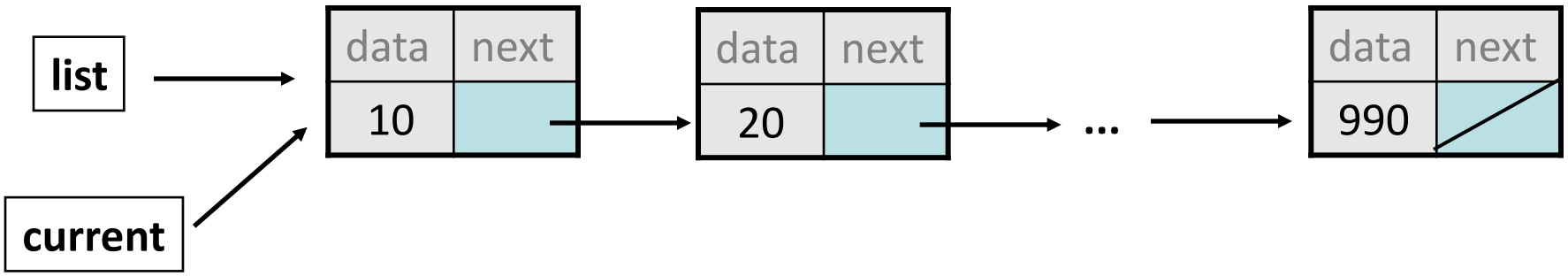
This modifies our only reference to the head of the list!



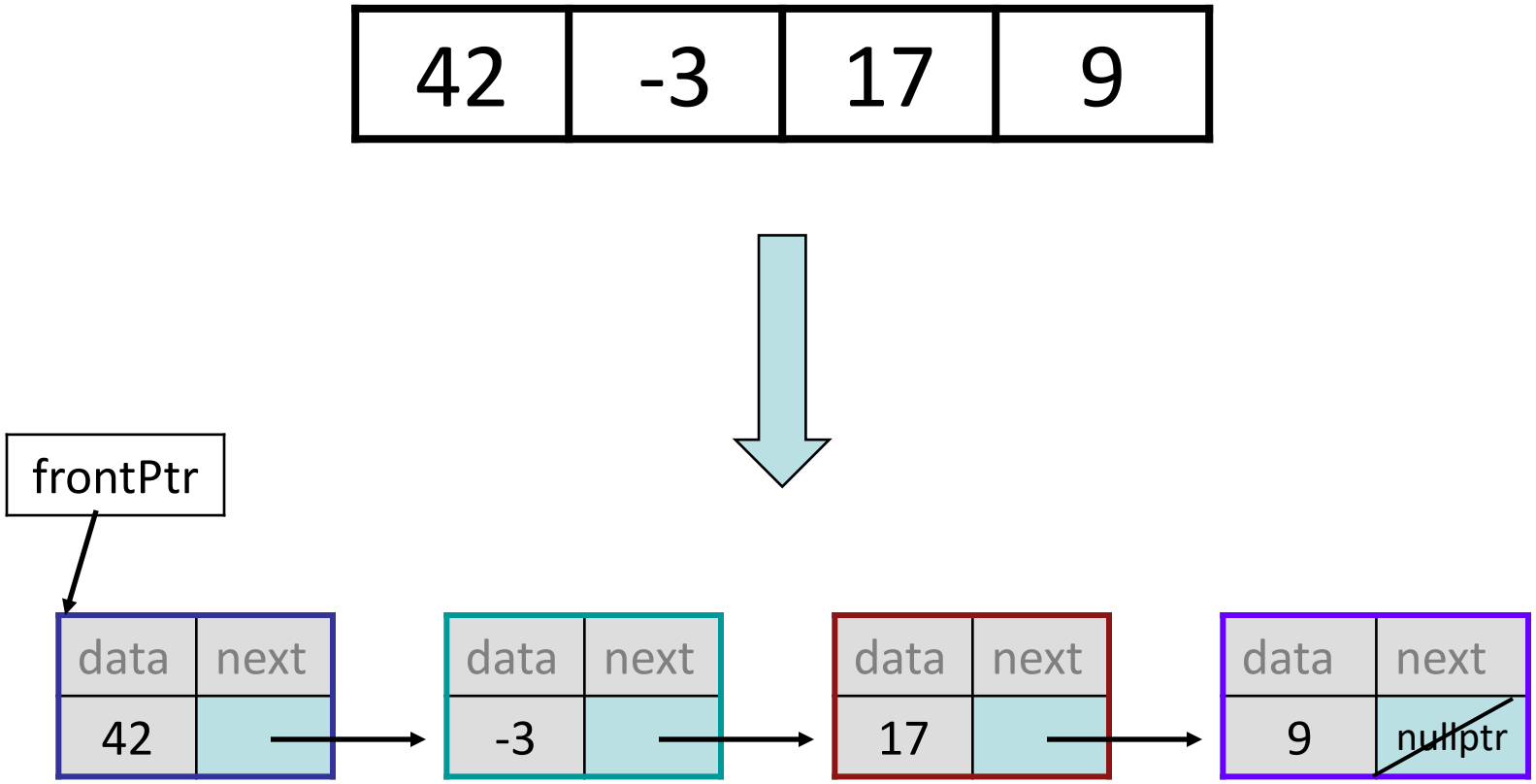
Traversing a list (12.2)

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

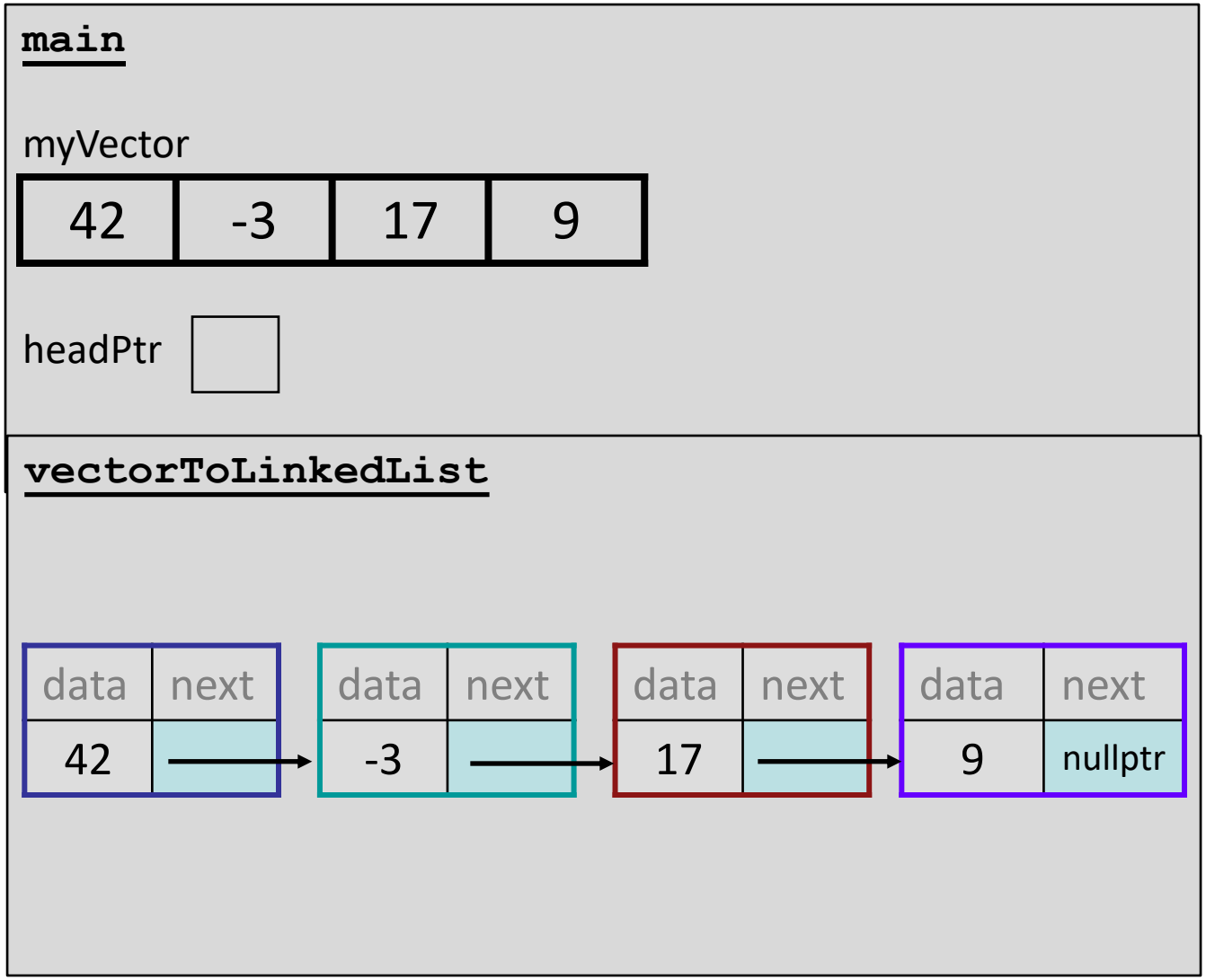
```
ListNode *vectorToLinkedList(const Vector<int>& v) {
    if (v.size() == 0) return nullptr;
    ListNode head = {v[0], nullptr};
    ListNode *currPtr = &head;
    for (int i = 1; i < v.size(); i++) {
        ListNode node = {v[i], nullptr};
        currPtr->next = &node;
        currPtr = &node;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

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

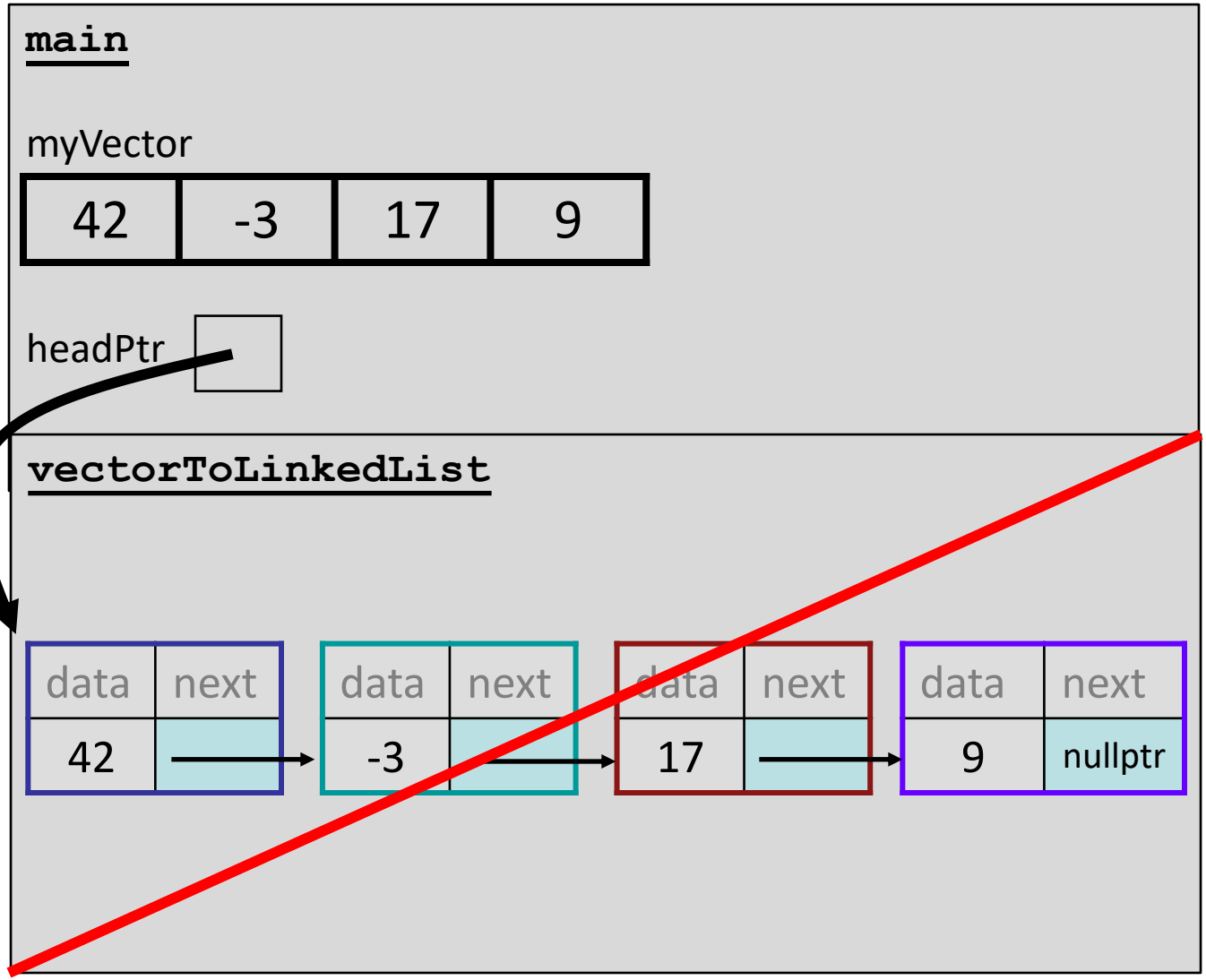
Creating a List

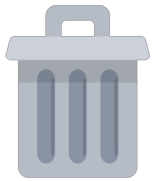
```
int main() {
    Vector<int> v = {42, -3, 17, 9};
    ListNode *headPtr = vectorToLinkedList(v);
    if (headPtr) {
        cout << *headPtr << endl;
    }
}
```

Creating a List



Creating a List





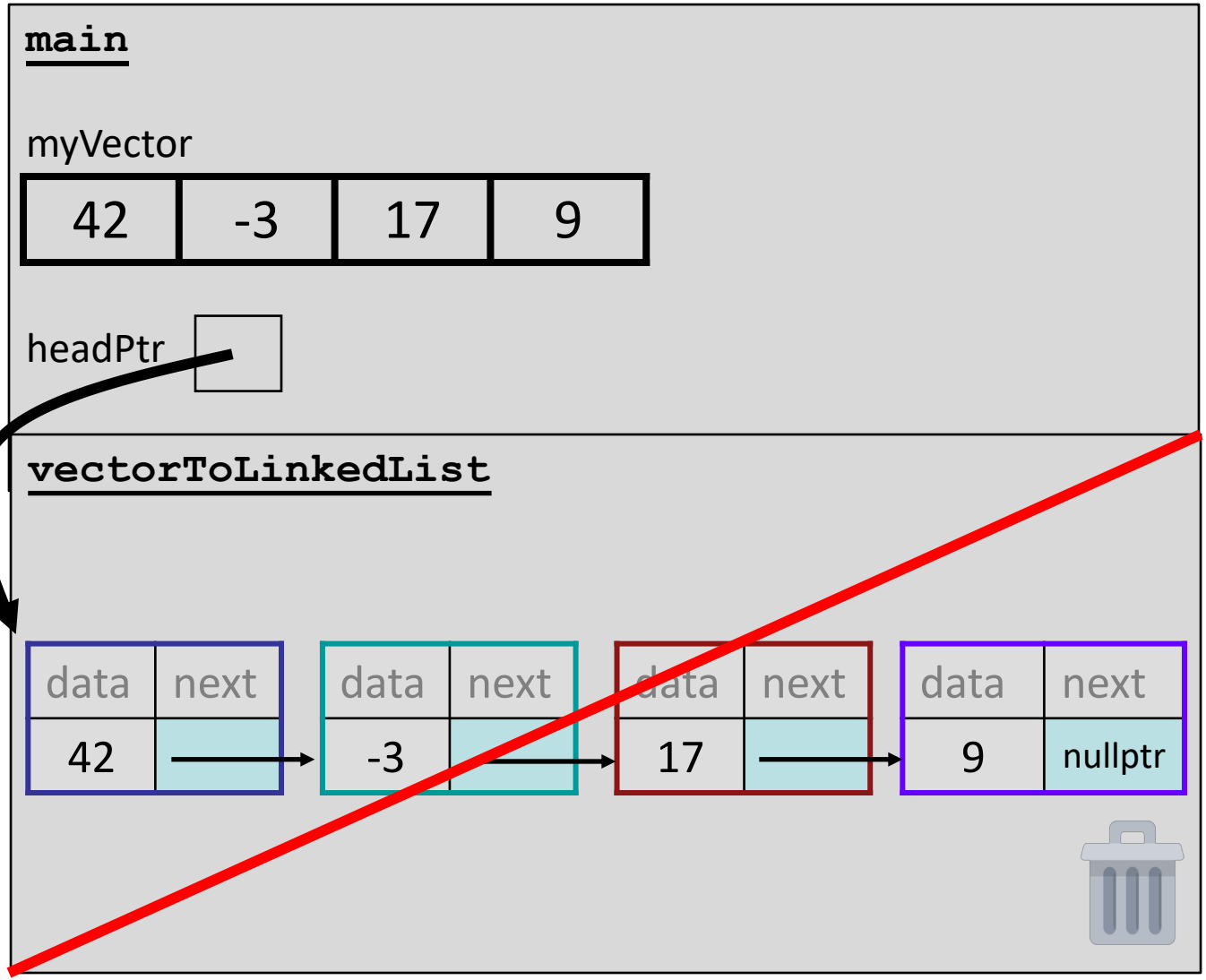
Creating a List

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

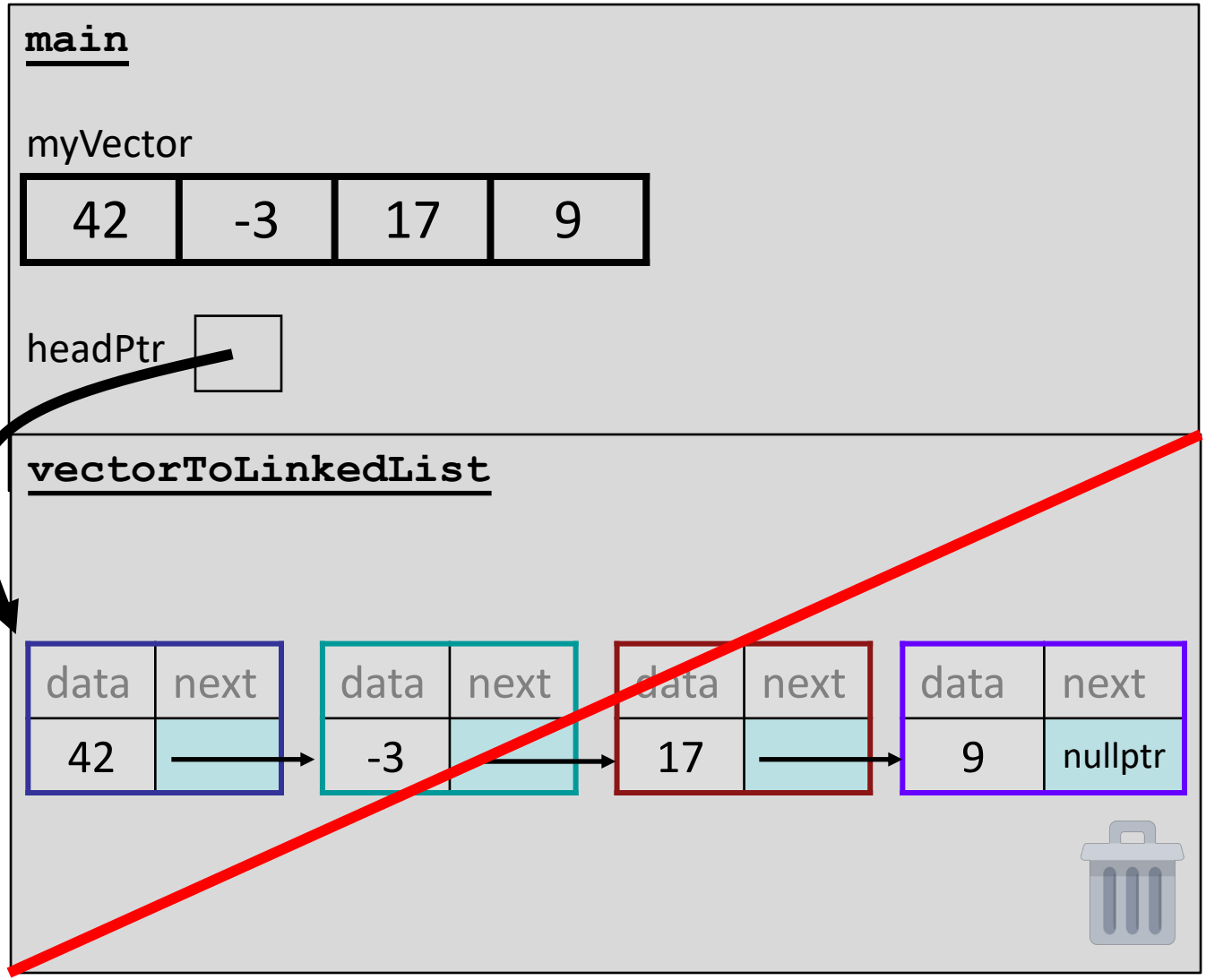
Plan For Today and Friday

- Classes
- Announcements
- 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