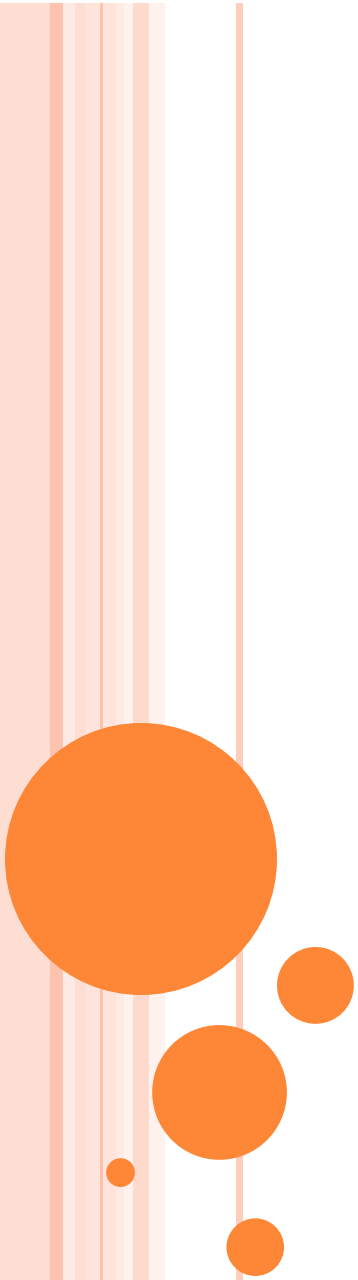


Singly Linked Lists (Part A)

CS A250 – C++ Programming II

Introduction

- **Singly-linked list**
  - Constructed using **pointers**
  - ***Grows*** and ***shrinks*** during **runtime**
  - **Doubly-linked lists**:
    - A variation with pointers in both directions
- **Pointers** are the backbone of such structures
  - Use ***dynamic*** variables
- **Standard Template Library**
  - Has predefined versions of linked lists

APPROACHES

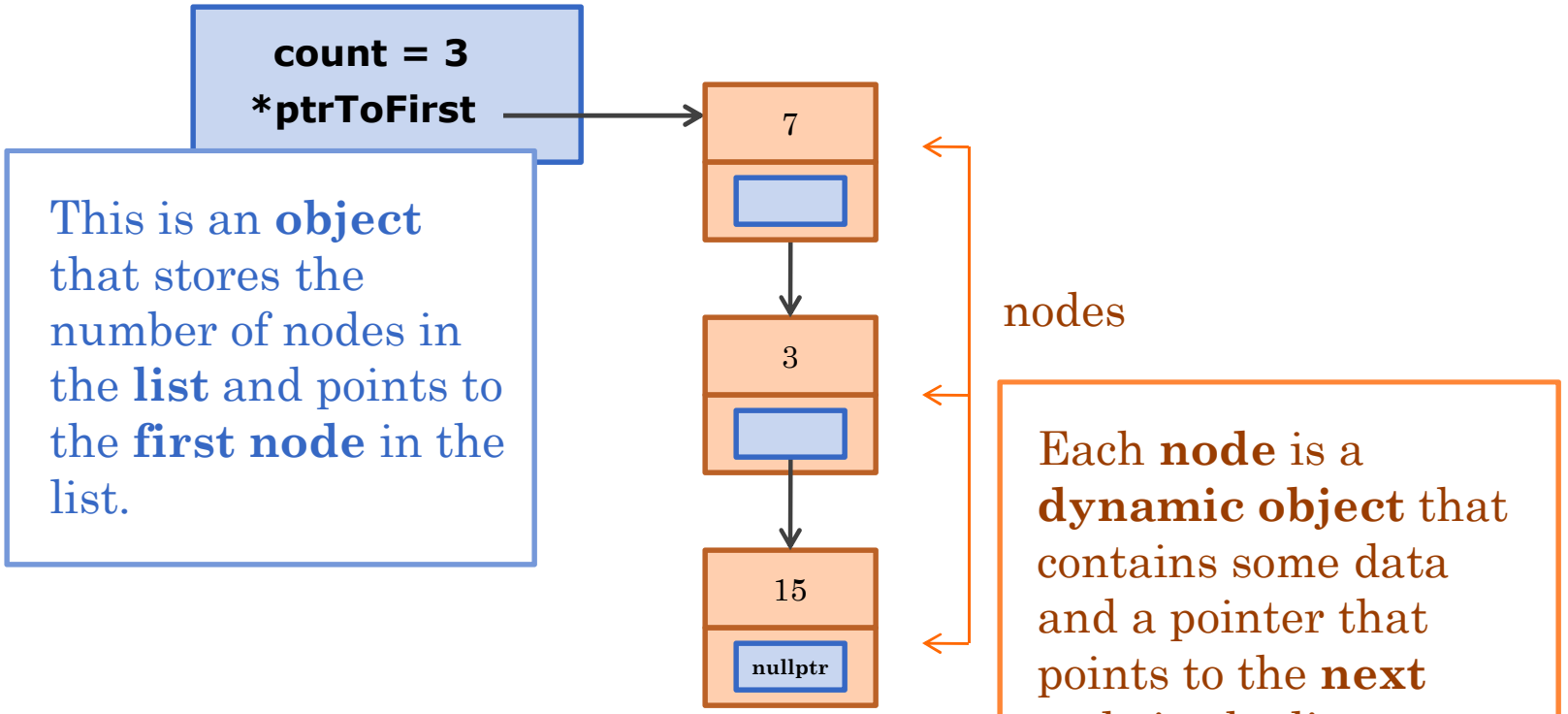
- Three ways to handle such data structures:
  1. **C-style approach:** global functions and **structures** with everything **public**
  2. **Classes** with **private** **member variables** and **accessor** and **mutator** functions
  3. **Friend classes**
- We will use approach 2

Linked Lists and Nodes

- **Linked list**
  - Simple example of "**dynamic data structure**"
  - Composed of **nodes**

- Each "**node**" is an **object** that is ***dynamically*** created with **`new`**
  - Nodes contain **pointers** to other nodes.

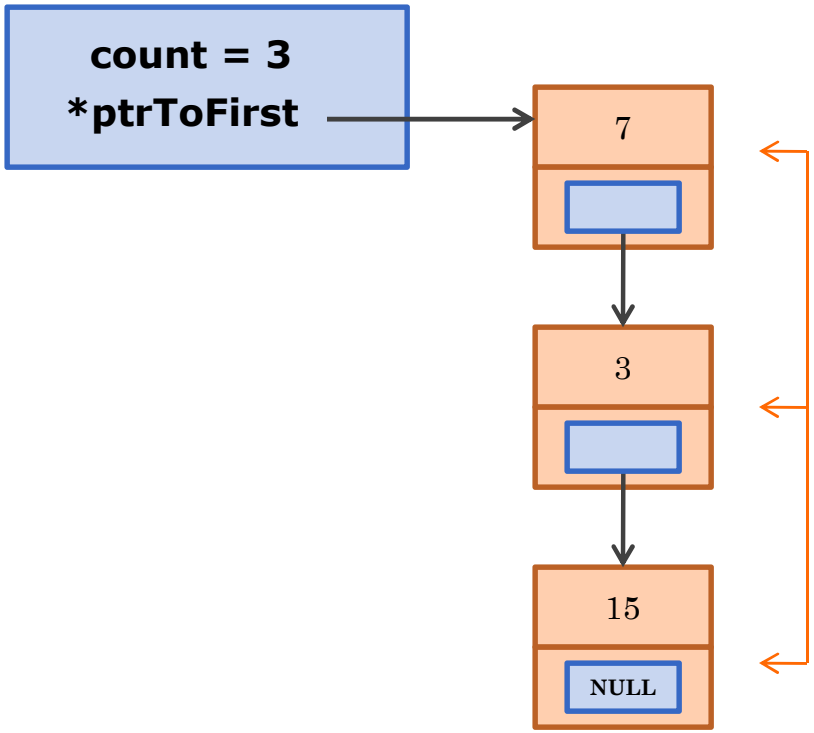
Nodes and Pointers



This is an **object** that stores the number of nodes in the **list** and points to the **first node** in the list.

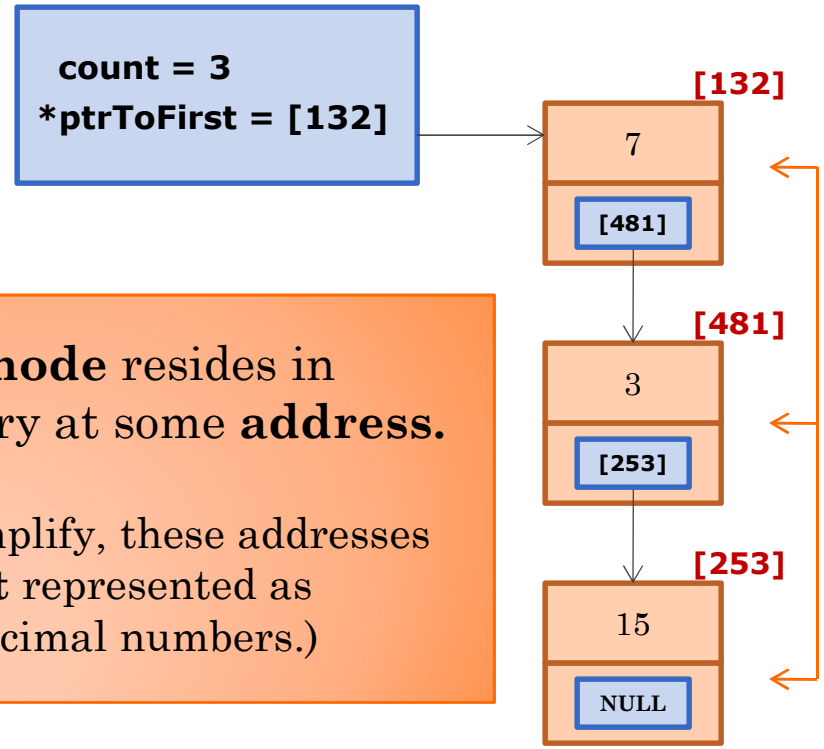
Each **node** is a **dynamic object** that contains some data and a pointer that points to the **next** node in the list.

Nodes and Pointers



In this example, each node stores an **int** and a **pointer** that points to the next node.

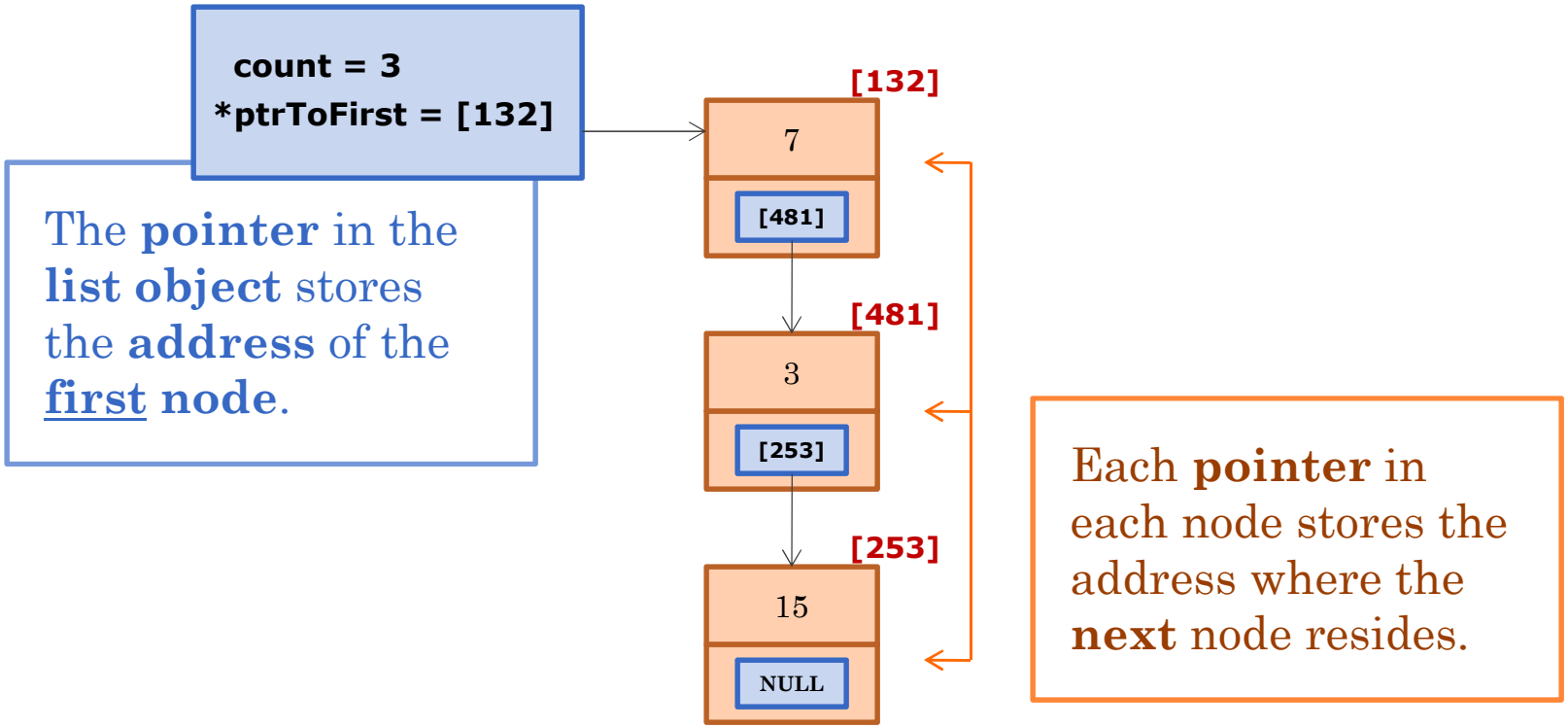
Nodes and Pointers



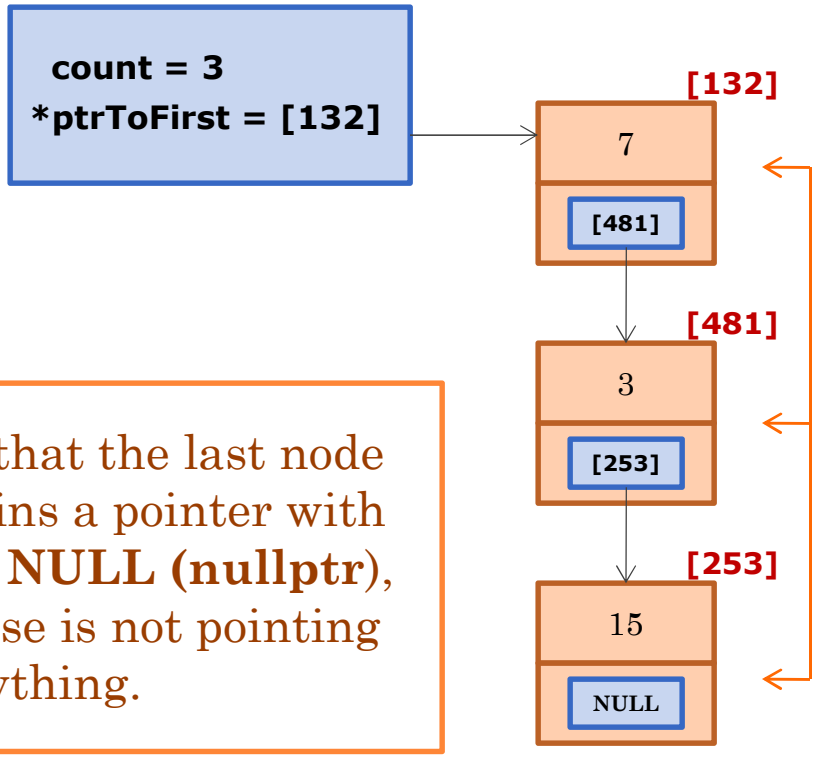
Each **node** resides in memory at some **address.**

(To simplify, these addresses are **not** represented as hexadecimal numbers.)

Nodes and Pointers



Nodes and Pointers



Note that the last node contains a pointer with value **NULL (nullptr)**, because is not pointing to anything.

Linked List

- These type of lists are called **linked lists**

- The **list object** contains a **pointer**, usually named **head** or **first**, that **points** to the **first node** in the list
  - We will call this **pointer** **ptrToFirst** for now (this is a pointer, **NOT** a node)

- The **last node** will contain a **pointer** that is set to **`NULL (nullptr)`**
  - Considered it a "sentinel" value, because it indicates there are no other nodes after this one.

Implementation

Linked List Implementation

- To implement a **linked list** we need **2 classes**:
  - A **class** to create the **list object**
  - A **class** to create the **node object**

- For our example, we will have **nodes** that have ***only*** two pieces of data (to simplify):
  - An **integer**
  - A **pointer** that links to next node

Linked List Implementation (cont.)

- **Node class**
  - Creates a **node** that has two member variables (can have more):
    - An **integer** name **data** storing some number
    - A **pointer** named **ptrToNext** that we set to point to the next node
      - This **pointer** is usually named **next** or **link**

- We will have all **member functions definitions *inline***
  - Because the class is short and simple enough.

Node Class Definition

```
class Node
{
public:
    Node() : data(0), ptrToNext(nullptr){}
    Node(int newData, Node *newPtrToNext)
        : data(newData), ptrToNext(newPtrToNext){}
    Node* getPtrToNext( ) const { return ptrToNext; }
    int getData( ) const { return data; }
    void setPtrToNext( Node *newPtrToNext )
        { ptrToNext = newPtrToNext; }
     void setData( int newData ) { data = newData; }
     ~Node(){}
private:
    int data;
    Node *ptrToNext;
};
```

AnyList Class

- Once we have the **Node class**, we need a **class** that creates an **object** to point to the **first node**.

- In our example, we implement a class named **AnyList**
  - Creates a **list** object with the following attributes:
    - A **pointer ptrToFirst**
      - points to the ***first*** **node** of the **list**
    - An **int count**
      - keeps track of how many **nodes** are in the **list**.

EXAMPLE

- **Project:** 01a_singly_linked_lists
  - AnyList.h

Arrow Operator

- Operator **->**
  - Called **arrow operator**
  - Shorthand notation that combines " * " and " . "

- These two statements are equivalent:

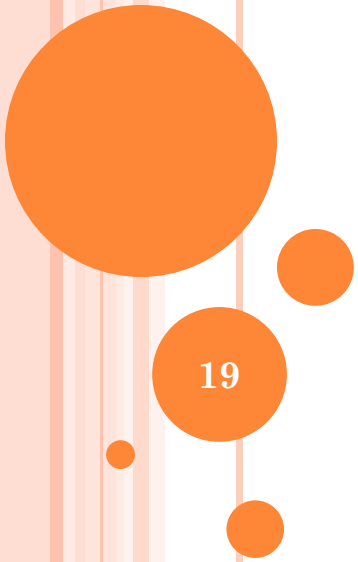
```
ptrToNode->setData(3);

(*ptrToNode).setData(3);
```

EXAMPLE

- **Project:** 01a_singly_linked_lists
  - **Constructor**
  - How to create a node

- **NOTE:** *Since one of the member variables of the* ***SinglyLinked List*** *class is dynamic, the class should include a* ***copy constructor*** *and an* ***overloaded assignment operator****. For practical purposes, we will omit these until we address these topics later in the semester.*



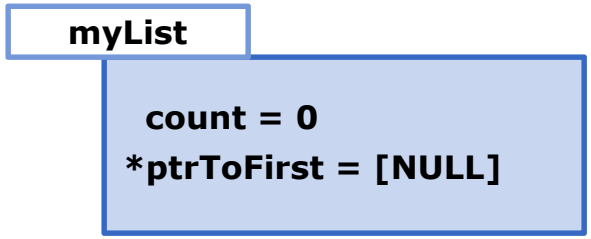
Creating a List

How Do We Create a Linked List?

- First, we need to create the **list object**.
- In our example, we would use the **AnyList class**

```
AnyList myList;
```

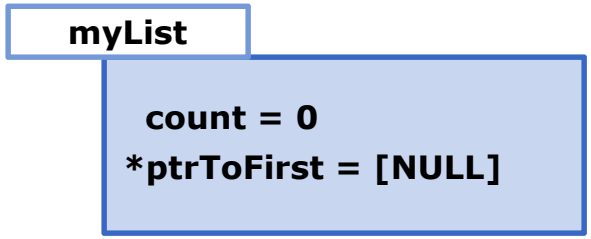
- This statement will call the **constructor** in the **AnyList class** to create the object.



How Do We Create a Linked List? (cont.)

- Once you have the **list object**, we create a **dynamic node** by using one of the **constructors** of the **Node class**.

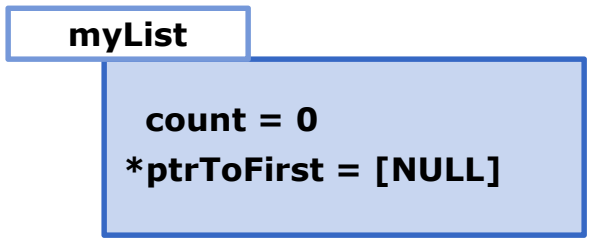
```
Node *ptr...              // Create a pointer.
```

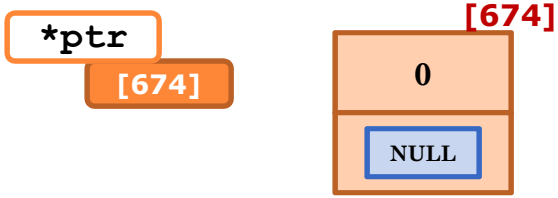


How Do We Create a Linked List? (cont.)

- Once you have the **list object**, we create a **dynamic node** by using one of the **constructors** of the **Node class**.

```
Node *ptr = new Node;  // Create a pointer.
                       // Point the pointer to a new node
                       // by using the default constructor.
```



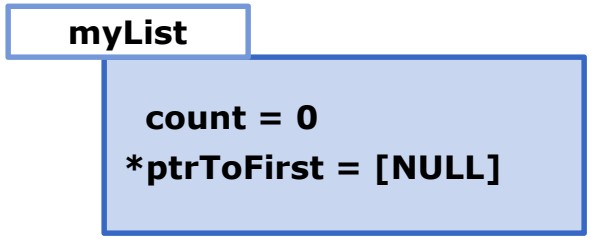


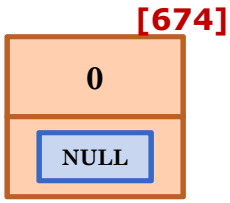
Note that the default constructor will give a default value of 0 to the variable stored in the node.

How Do We Create a Linked List? (cont.)

- Once you have the **list object**, we create a **dynamic node** by using one of the **constructors** of the **Node class**.

```
Node *ptr = new Node;  // Create a pointer.
                       // Point the pointer to a new node
                       // by using the default constructor.
```





This node is in **dynamic memory** and has **no identifier**. The only way to get there is by knowing the **address** were the node is located.

How Do We Create a Linked List? (cont.)

- Once you have the **list object**, we create a **dynamic node** by using one of the **constructors** of the **Node class**.

```
Node *ptr = new Node;  // Create a pointer.
                       // Point the pointer to a new node
                       // by using the default constructor.
ptr->setData(7); // Give a value to store in the node.
```

How Do We Create a Linked List? (cont.)

- Once you have the **list object**, we create a **dynamic node** by using one of the **constructors** of the **Node class**.

```
Node *ptr = new Node;  // Create a pointer.
                       // Point the pointer to a new node
                       // by using the default constructor.
ptr->setData(7); // Give a value to store in the node.
++count;      // Increment the count
```

How Do We Create a Linked List? (cont.)

- Let's simplify…

How Do We Create a Linked List? (cont.)

- We could simplify even further by using the **overloaded constructor**:

```
Node *ptr = new Node(4, nullptr);
// Create a pointer and make it point to a new node by
// using the overloaded constructor and storing a
// value right away.
```

How Do We Create a Linked List? (cont.)

- We could simplify even further by using the **overloaded constructor**:

```
Node *ptr = new Node(4, nullptr);
// Create a pointer and make it point to a new node by
// using the overloaded constructor and storing a
// value right away.
ptrToFirst = ptr; // ptrToFirst will point to the node
                  // pointed by ptr.
```

How Do We Create a Linked List? (cont.)

- We could simplify even further by using the **overloaded constructor**:

```
Node *ptr = new Node(4, nullptr);
// Create a pointer and make it point to a new node by
// using the overloaded constructor and storing a
// value right away.
ptrToFirst = ptr; // ptrToFirst will point to the node
                  // pointed by ptr.
++count; // Increment count.
```

How Do We Create a Linked List? (cont.)

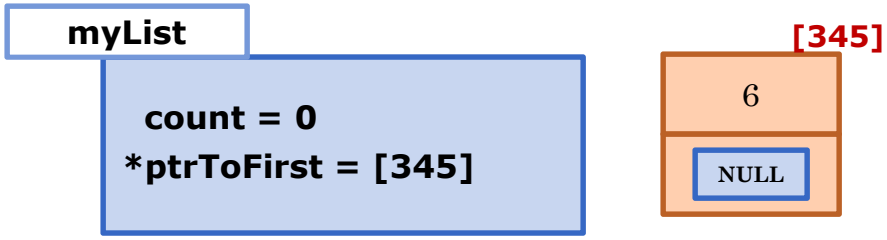
- Let's simplify even more…

How Do We Create a Linked List? (cont.)

- The MOST efficient way to create the first node is done in one single statement.

```
ptrToFirst = new Node(6, nullptr);
```

We do NOT create a pointer at all.
We use pointer **ptrToFirst** to create the new node.

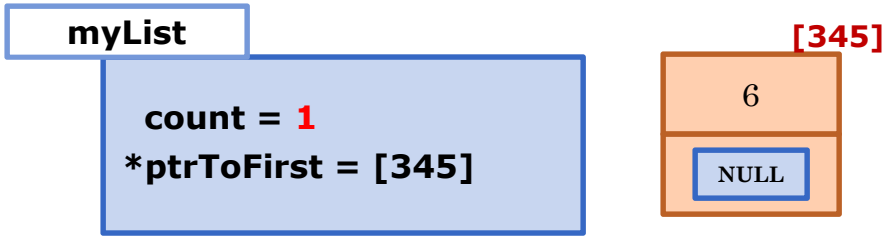


How Do We Create a Linked List? (cont.)

- The MOST efficient way to create the first node is done in one single statement.

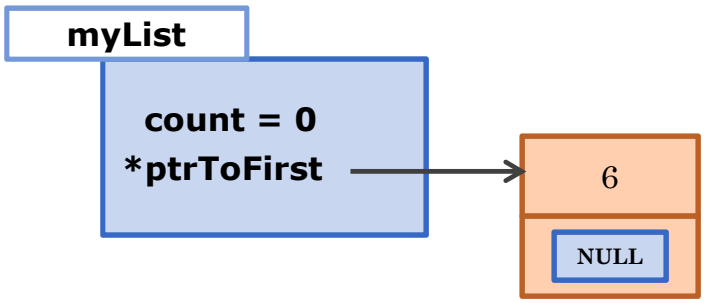
```
ptrToFirst = new Node(6, nullptr);
++count;
```

Don't forget to increment the count!

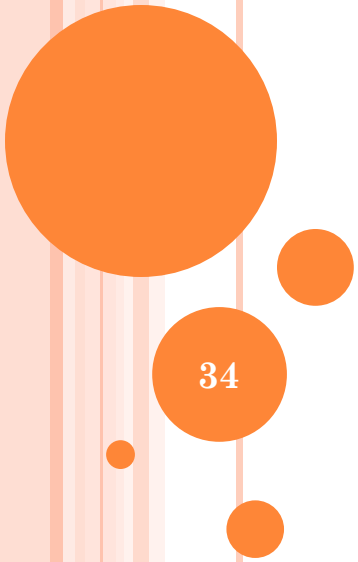


Arrows to Visualize the List

- From now on, we will simplify our representation by using **arrows** instead of addresses.



- The next section will show you how to add more nodes to the list.



Inserting Nodes

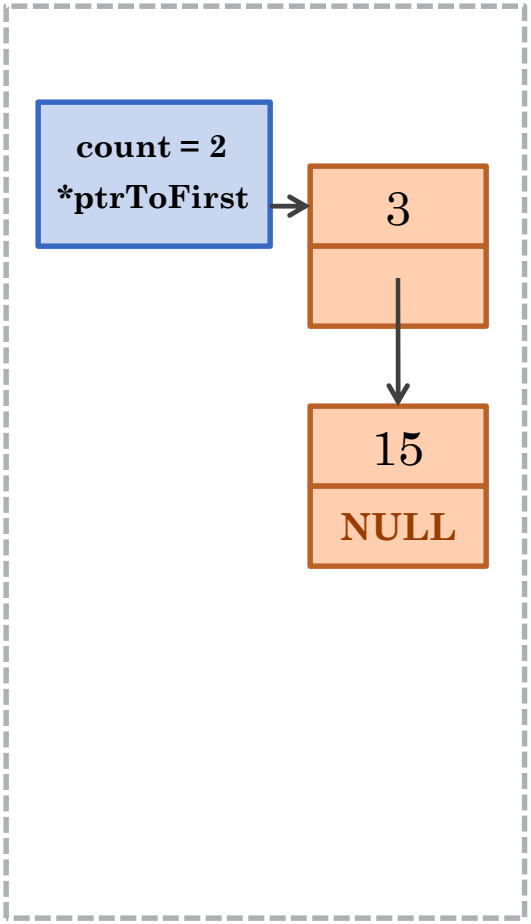
Inserting to the Front of the List

- To **insert a node** to the *front* of the list,
  you need to:
  1. Create a pointer to point to a **new** node
     (this is ***dynamic***)
  2. Create a new node
  3. Store data in the new node
  4. Set **new node's pointer** to point to the **first node**
  5. Make the new node be the "first" node
  6. Increment the count

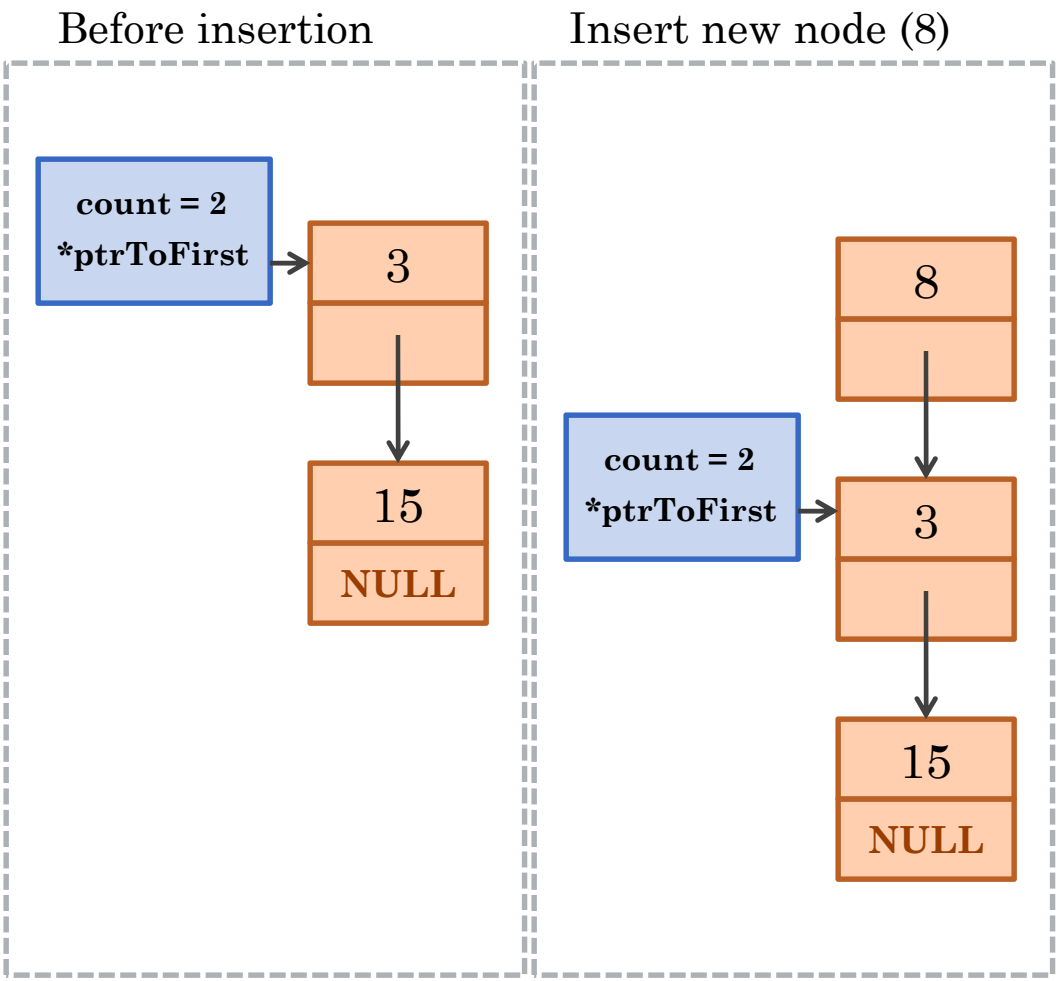
- **Note:** If the **list** is **empty**
  - Then the new node is the **first** and **only** node.

Inserting to the Front of the List

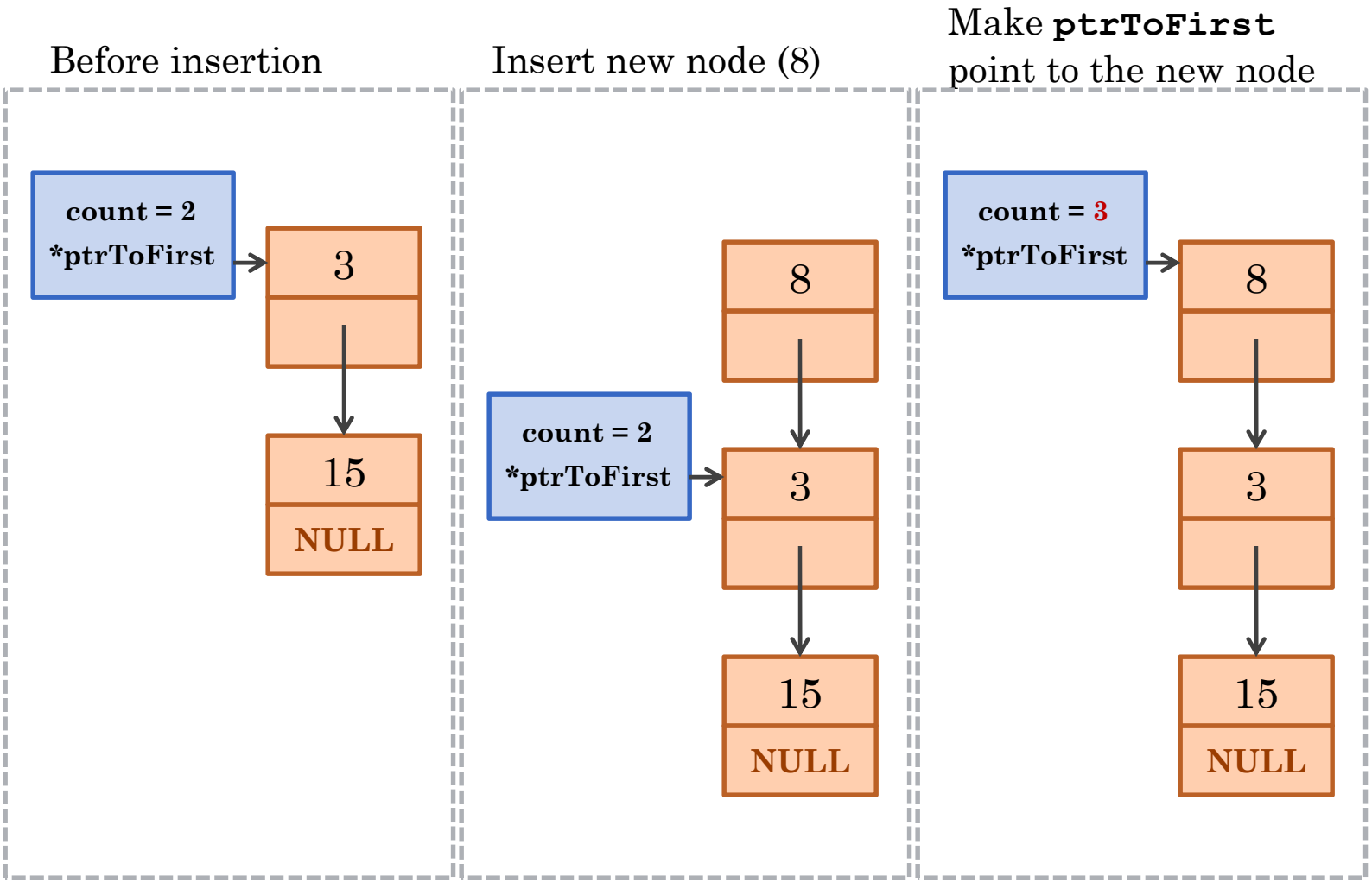
Before insertion



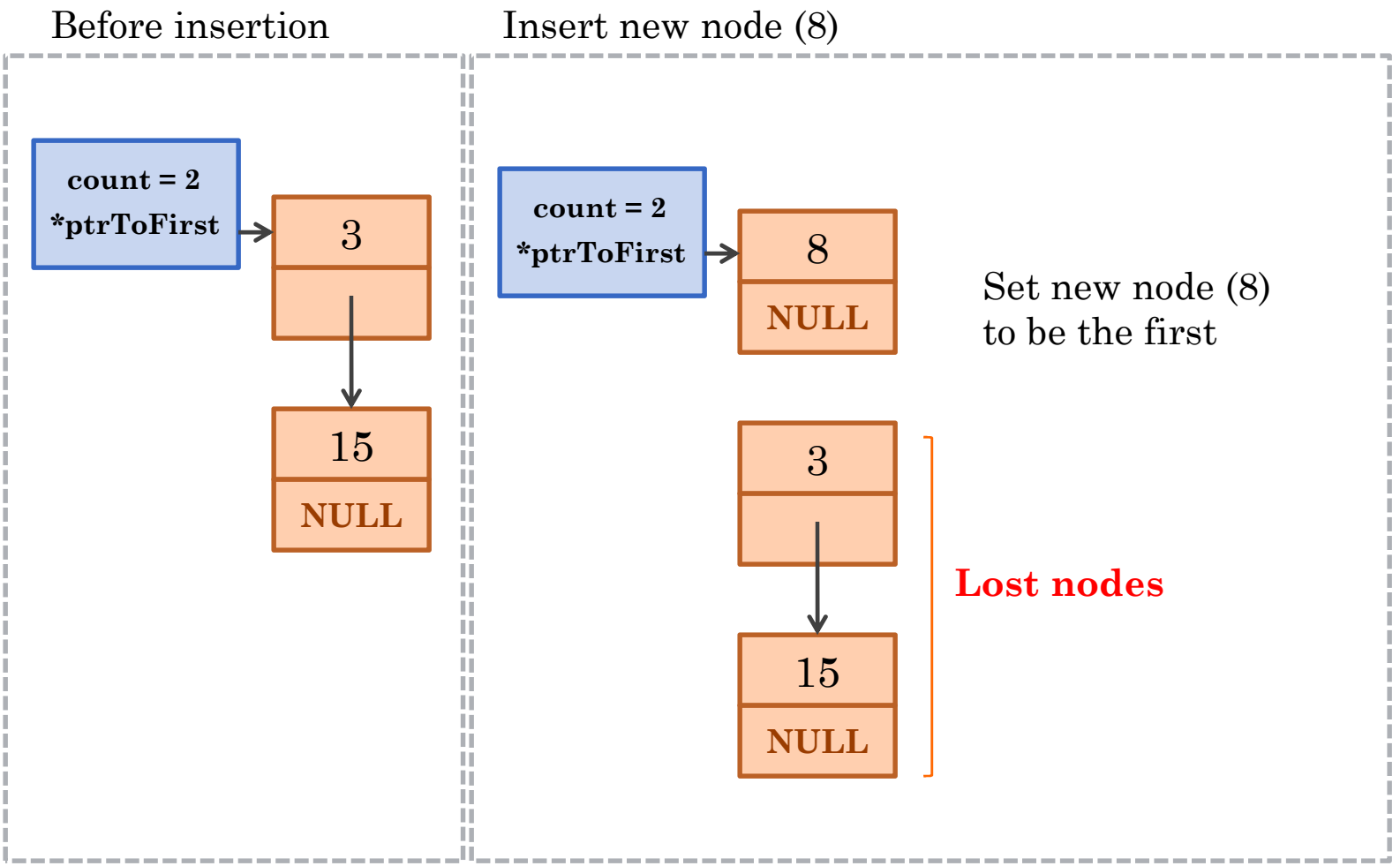
Inserting to the Front of the List



Inserting to the Front of the List

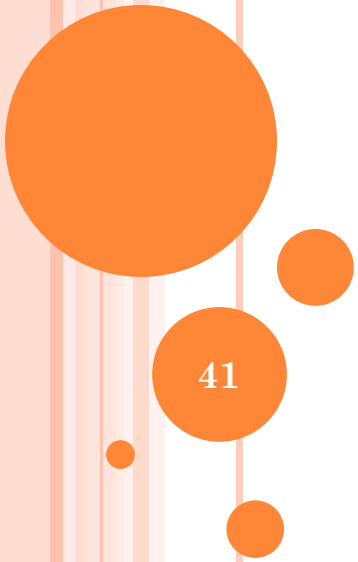


Pitfall: Lost Nodes



EXAMPLE

- **Project:** 01a_singly_linked_lists
  - Function: **insertFront( )**
  - Inserting to the front of the list



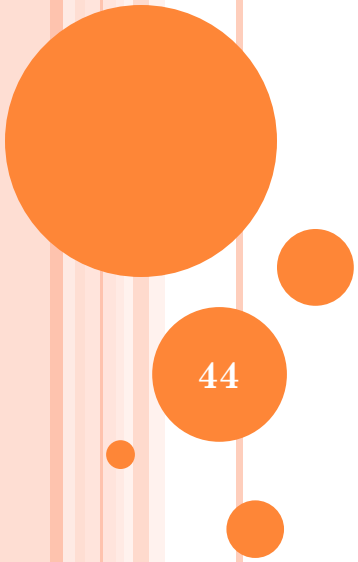
Printing the List

Printing the List

- How do you **print** the list?
  1. Create a **pointer** to traverse the list → **`current`**
  2. Set the **`current`** pointer to point to the **`first`** node
  3. While the **`current`** pointer is not **`NULL`** (that is, has not reached the end of the list)
     a) Output the data the **`current`** pointer is pointing to
     b) Move the **`current`** pointer forward

EXAMPLE

- **Project:** 01a_singly_linked_lists
  - Function: **print()**

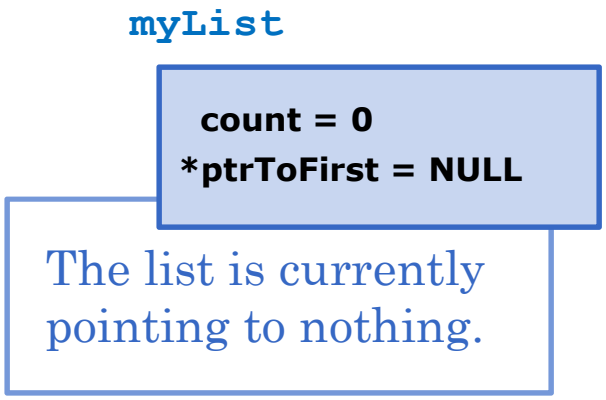


Syntax and More

A Short List

```
AnyList myList;
```

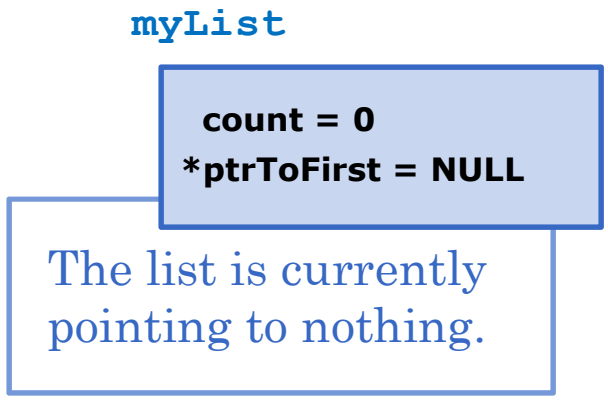
- Creates an **object** of the class **AnyList**.
- Uses the **default constructor** to set the **member variables** to **default values**.



A Short List (cont.)

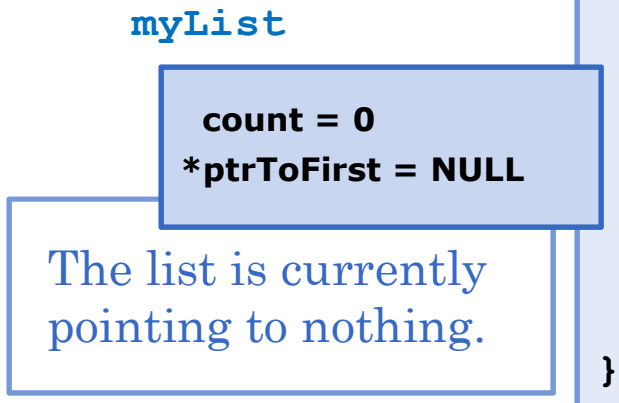
```
myList.createShortList();
```

- The **object myList** calls **createShortList** , **member function** of the class **AnyList**.



A Short List (cont.)

- The next set of slides will show the implementation of the **member function createShortList** of the class **AnyList**.

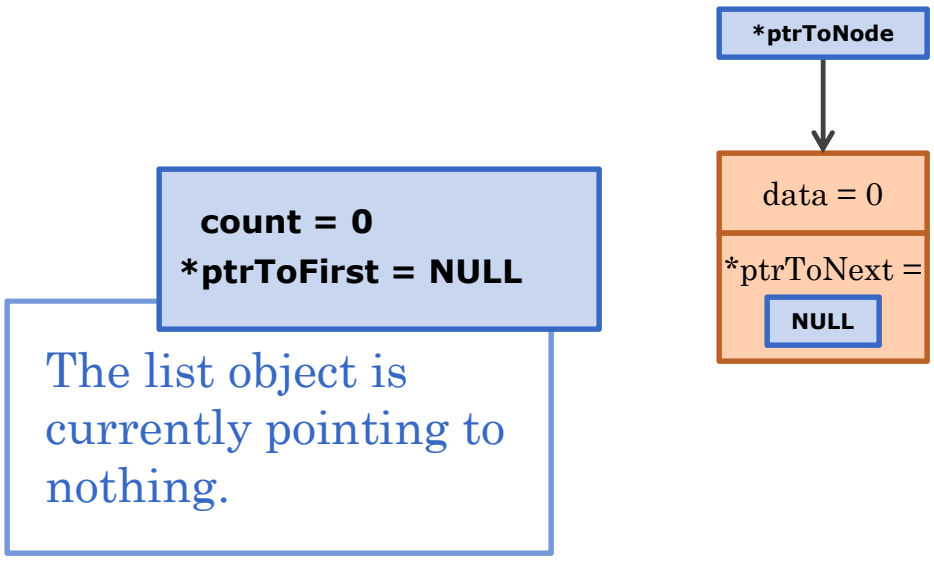



```
void AnyList::createShortList()
{
   // create a node
   // make this node to be the first node
   // update the count
   // create another node, using same pointer
   // connect the first node with this node
   // update the count
   // change the data in the first node
   // print the value stored in the first node
}
```

A Short List (cont.)

```
Node *ptrToNode = new Node;
```

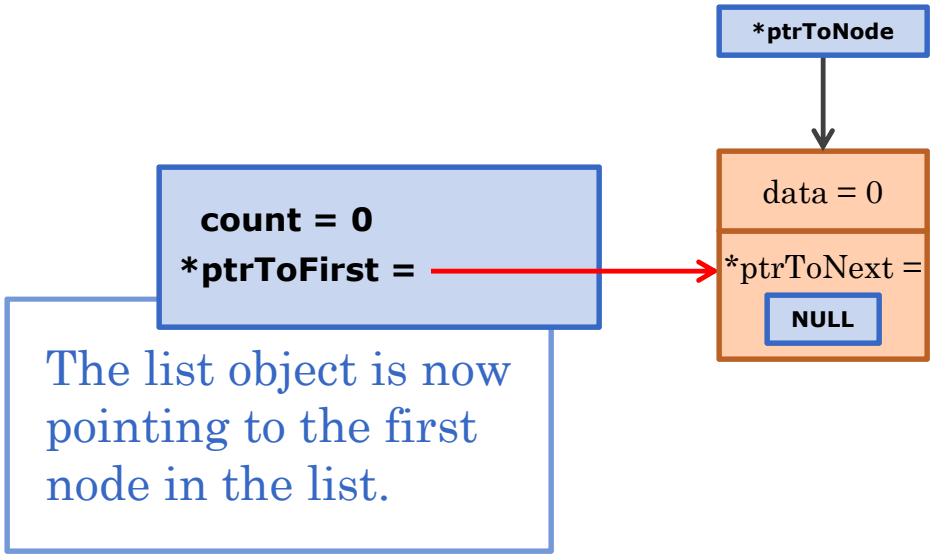
- Creates a ***pointer*** (**ptrToNode**) that points to a **new** node that has no name.
- Uses the **default constructor** to set **default values** for the node (0 and NULL)



A Short List (cont.)

```
ptrToFirst = ptrToNode;
```

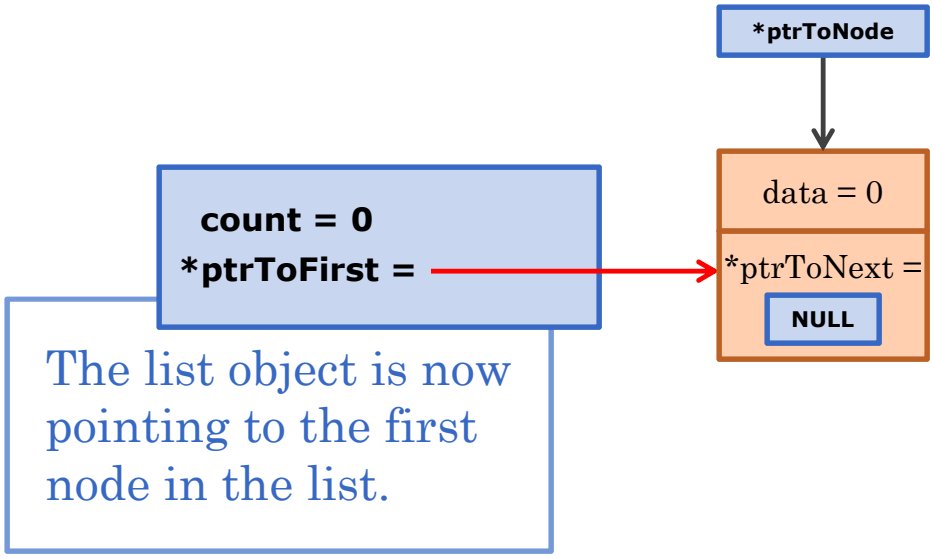
- The pointer **ptrToFirst**, which is inside the **list object**, will point to the node pointed by pointer **ptrToNode**.



A Short List (cont.)

```
ptrToFirst = ptrToNode;
```

- The pointer **ptrToFirst**, which is inside the **list object**, will point to the node pointed by pointer **ptrToNode**.



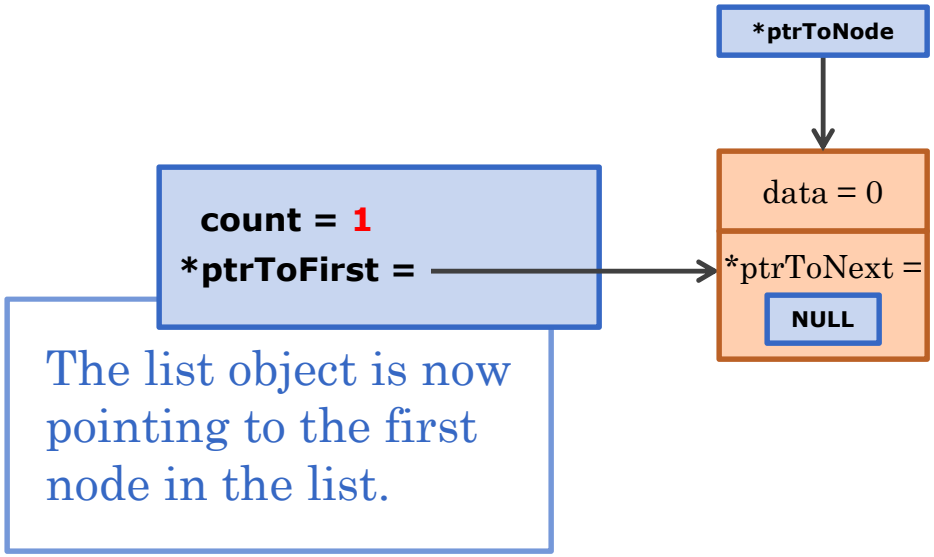
Of course, we could have done this in one easy step:

```
ptrToFirst = new Node;
```

A Short List (cont.)

```
++count;
```

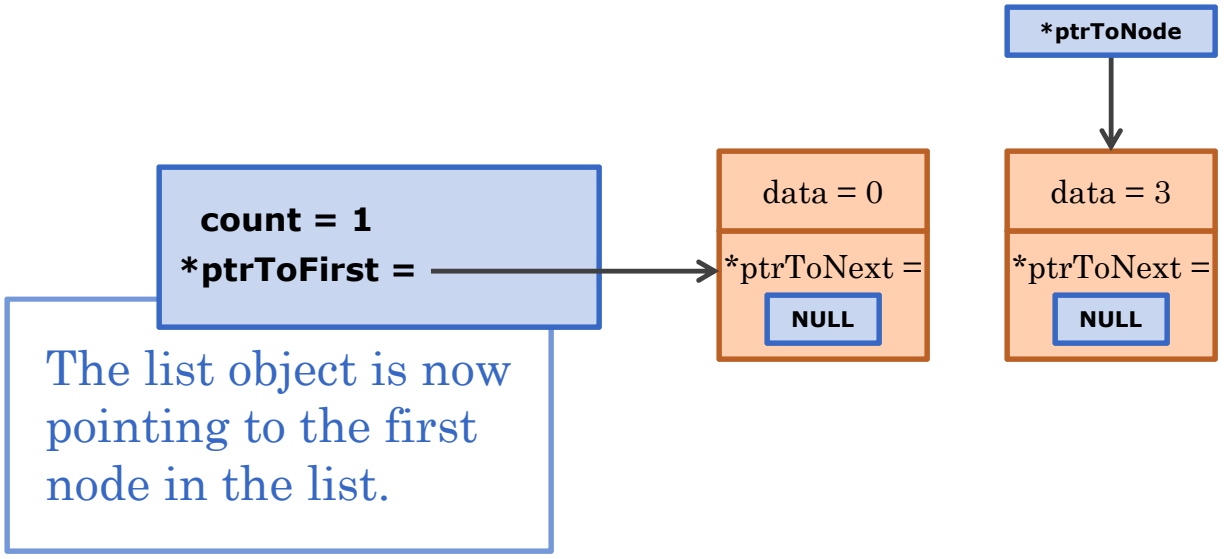
- Updates the **count**, because now the list has 1 node.



A Short List (cont.)

```
ptrToNode = new Node (3, nullptr);
```

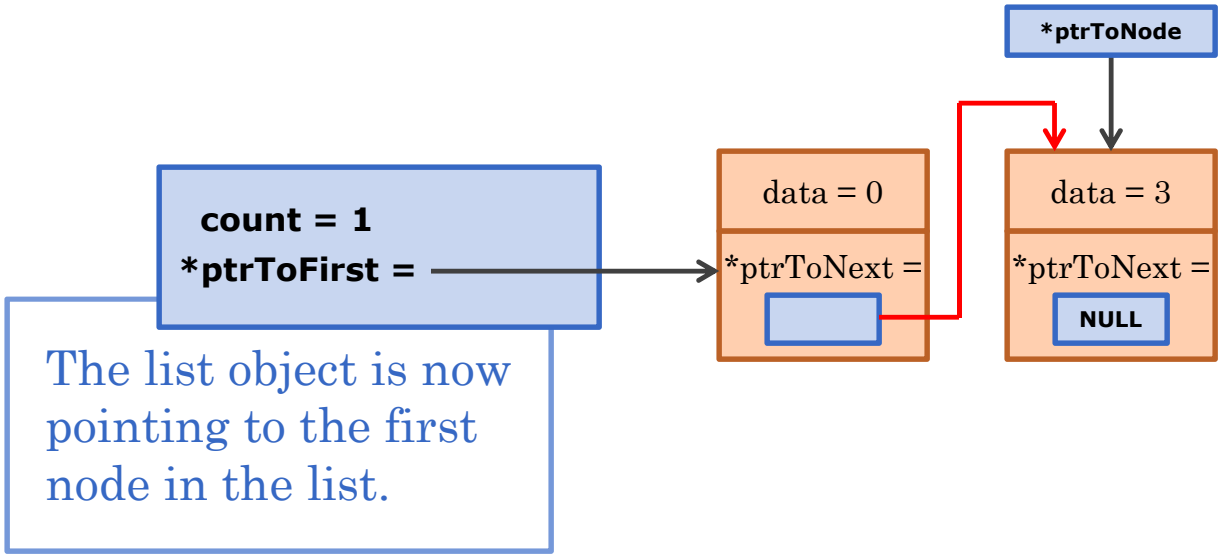
- Using the same pointer, creates a **new** node that has no name.
- Uses the **overloaded constructor** to set **specific values** for the node (3 and NULL)



A Short List (cont.)

```
ptrToFirst->setPtrToNext(ptrToNode);
```

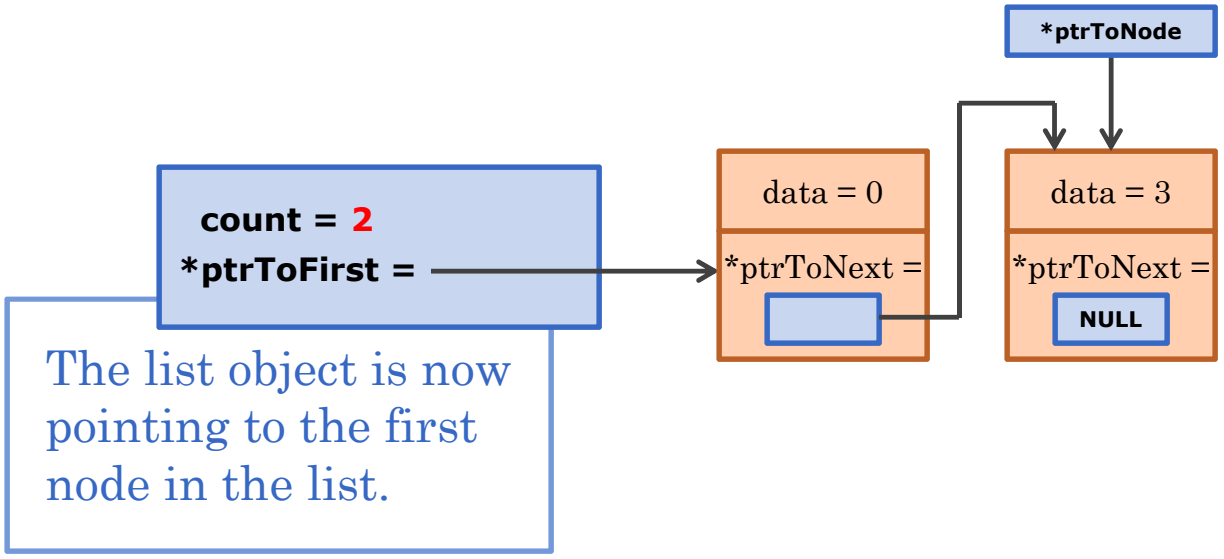
- The pointer **ptrToNext** nside the **first node** will point to the node pointed by pointer **ptrToNode**.



A Short List (cont.)

```
++count;
```

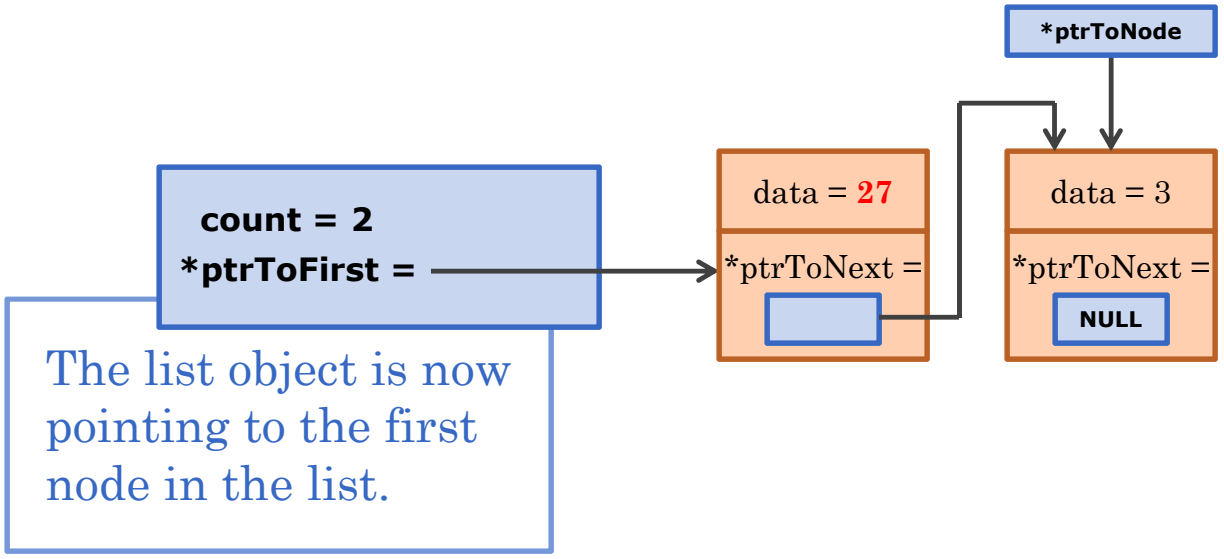
- Updates the count, because now the list has 2 nodes.



A Short List (cont.)

```
ptrToFirst->setData(27);
```

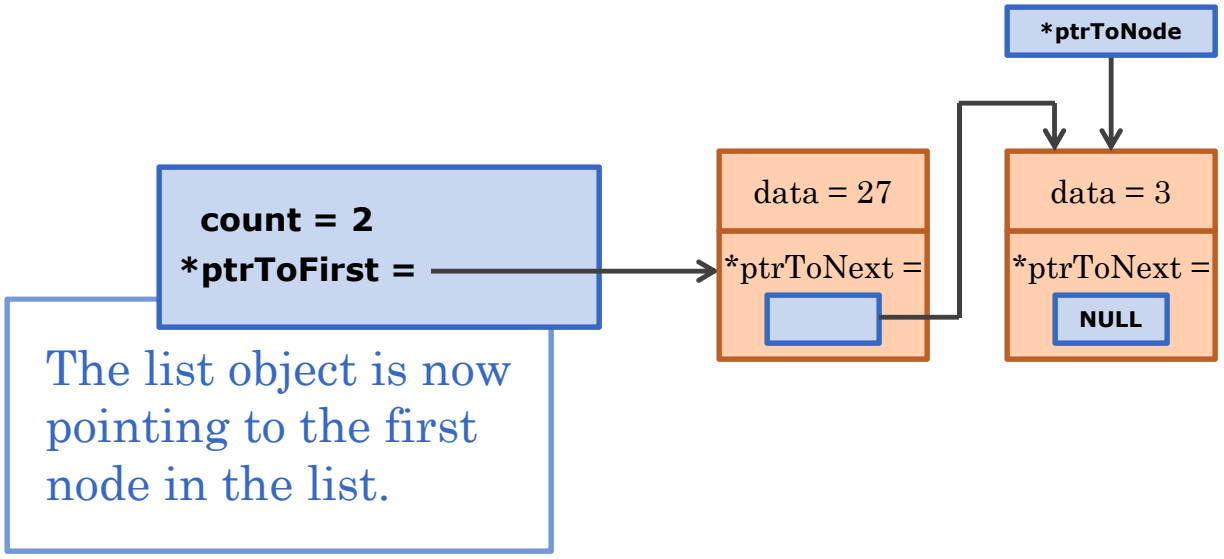
- Overwrites the data stored in the first pointer with 27.



A Short List (cont.)

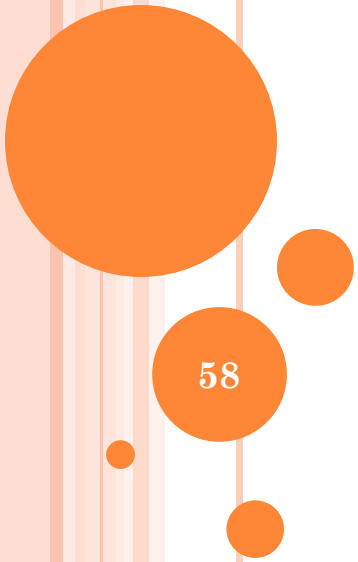
```
cout << ptrToFirst->getData();
```

- Prints out the data stored in the first node.



A Short List (cont.)

- To add more nodes, you will need a **loop**.
- The **project** included with the slides shows how to loop through a list.



Destroying the List

Destroying the List

- Since you are storing the nodes in **dynamic memory**, you will need to **delete all nodes**.

- The best way is to use 2 functions:
  - **destroyList**
    - Deletes each node in the list
  - **Destructor**
    - This will be automatically called.
    - You **cannot** delete the **list object**, but you can **empty it** by calling the **destroyList** function.

THE FUNCTION destroyList

- Create a pointer and set it to point to the first node.
  - Using a **while** loop until the pointer is NULL:
    - Make the second node be the new first node.
    - Delete the node that the pointer is pointing to.
    - Move the pointer to point to the new first node.
- Update the **count** to 0
  - Do **NOT** increment the count in the loop; that would be inefficient. You know that the count will eventually be zero; therefore, you can update the count at the end of the function, outside the loop.

EXAMPLE

- **Project:** 01a_singly_linked_lists
  - Destructor: **~AnyList()**
  - Function **destroyList**

Common Errors

- Forgetting to add
  - `#include <string>` in the `AnyList.h` file
    - Needed for `nullptr`
- Confusing **nodes** and **pointers**
- Forgetting to reset the pointer that points to the first node in the list, `ptrToFirst`
  - Always keep in eye on the **list object**, to avoid losing track of your list.

IMPORTANT !

- **Before executing** your program (F5) ***always*** do the following:
  - Click on **Build** → **Rebuild Solution**

IMPORTANT → COMMON IDENTIFIERS

- We have named the pointer that points to the first node **ptrToFirstNode**
  - BUT, common identifiers are: **first**, **head**
- We have named the pointer that points to the next node **ptrToNextNode**
  - BUT, common identifiers are: **link**, **next**
- We have named the pointer that points to a new node **ptrToNewNode**
  - BUT, most common identifier is: **newNode**

IMPORTANT → COMMON IDENTIFIERS

- We have named the pointer that points to the first node **ptrToFirstNode**
  - BUT, common identifiers are: **first**, **head**
- We have named the pointer that points to **ptrToNextNode**
  - BUT, common identifiers are: **link**, **next**
- We have named the pointer that points to **ptrToNewNode**
  - BUT, most common identifier is: **newNode**

IMPORTANT:

Our projects might use **any** of these identifiers.

End Singly-linked Lists (Part A)