

Lecture #4

- Resource Management, Part 2
 - Assignment Operators
- Basic Linked Lists
 - Insertion, deletion, destruction, traversals, etc.
- Advanced Linked Lists
 - Tail Pointers
 - Doubly-linked Lists

Assignment Operators



Assignment Operators...

Why should you care?

Assignment Operators are required in all nontrivial C++ programs.

If you fail to use them properly, it can result in **nasty bugs** and **crashes**.

So pay attention!

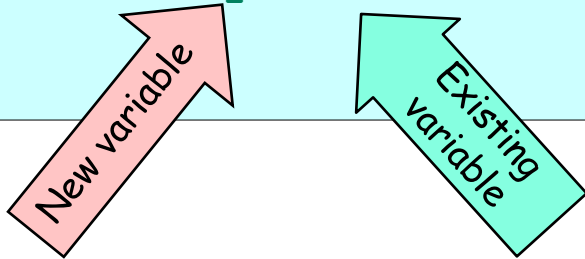
Why
should
I care?



The Assignment Operator

```
int main()
{
    Circ  x(1,2,3);

    Circ  y = x;
}
```



Last time we learned how to **construct** a **new variable** using the value of an **existing variable** (via the copy constructor).

Now let's learn how to *change* the value of an **existing variable** to the value of an **another variable**.

In this example, both **foo** and **bar** have been constructed.

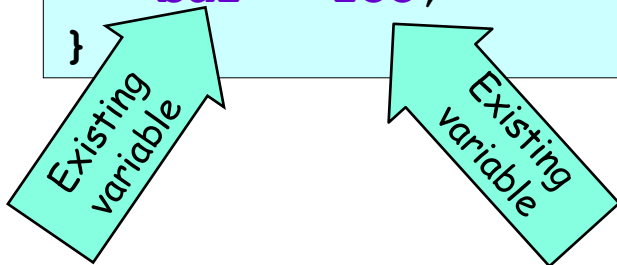
Both have had their member variables initialized.

Then we set **bar** equal to **foo**.

```
int main()
{
    Circ  foo(1,2,3);

    Circ  bar(4,5,6);

    bar = foo;
}
```



The Assignment Operator

In this case, the **copy constructor** is **NOT** used to copy values from **foo** to **bar**.

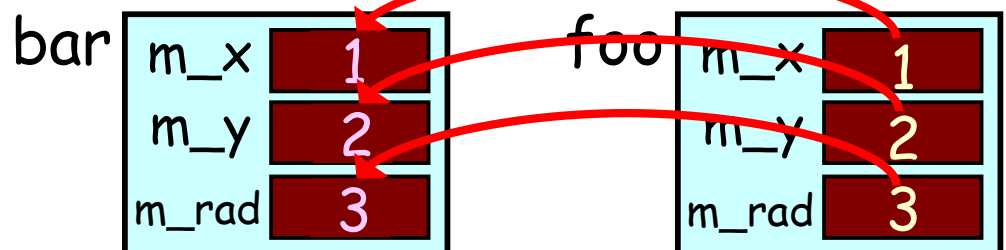
Instead, a special member function called an **assignment operator** is used to copy **foo**'s values into **bar**.

Why isn't **bar's copy constructor** called? Because **bar** was already constructed on the line above! The **bar** variable already exists and is already initialized, so it doesn't make any sense to re-construct it!

```
int main()
{
    Circ  foo(1,2,3);
    Circ  bar(4,5,6);
    bar = foo;
}
```

If you don't define your own **assignment operator**...

Then C++ provides a default version that just copies each of the members.



Lets see how to define our own **assignment operator**.

The Assignment Operator

Hmmm.. This looks familiar, doesn't it?

What does it remind you of?

foo

```
class Circ
{
public:
    Circ(float x, float y, float r)
    {
        m_x = x; m_y = y; m_rad = r;
    }
    void setMeEqualTo(const Circ &src)
    {
        m_x = src.m_x;
        m_y = src.m_y;
        m_rad = src.m_rad;
    }
    float GetArea(void)
    {
        return(3.14159*m_rad*m_rad);
    }
private:
    float m_x, m_y, m_rad;
};
```

The syntax for an assignment operator is a bit confusing.

So lets define a simpler version first...

Here's how we'd use our new function.

```
int main()
{
    Circ  foo(1,2,3);

    Circ  bar(4,5,6);

    bar.setMeEqualTo(foo);
} // same as bar = foo;
```

The Assignment Operator

The **const** keyword guarantees that the source object (src) is not modified during the copy.

Now lets see what a real assignment operator

You **MUST** pass a **reference** to the source object. This means you have to have the **&** here!!!

```
Circ(float x, float y, float r)
{
    m_x = x; m_y = y; m_rad = r;
}

Circ &operator= (const Circ &src)
{
    m_x = src.m_x;
    m_y = src.m_y;
    m_rad = src.m_rad;
    return(*this);
}

float GetArea(void)
{
    return(3.14159*m_rad*m_rad);
}

private:
    float m_x, m_y, m_rad;
};
```

1. The function name is **operator=**
2. The function return type is a **reference to the class**.
3. The function returns ***this** when its done.

I'll explain this more in a bit...

The Assignment Operator

```

foo
class Ci
{
    ...
    Circ &operator=(const Circ &src)
    {
        m_x = src.m_x;
        m_y = src.m_y;
        m_rad = src.m_rad;
        return(*this);
    }
    ...
private:
    m_x 1 m_y 2 m_rad 3
}
Circ &

```

```

int main()
{
    Circ foo(1,2,3);

    Circ bar(4,5,6);

    bar = foo;
}

```

So, to summarize...

If you've defined an **operator=** function in a class...

Then any time you use the **equal sign** to set an **existing variable** equal to another...

C++ will call the **operator=** function of your **target variable** and pass in the **source variable**!

bar

```

class Circ
{
    ...
    Circ &operator=(const Circ &src)
    {
        m_x = src.m_x;
        m_y = src.m_y;
        m_rad = src.m_rad;
        return(*this);
    }
    ...
private:
    m_x 4 m_y 5 m_rad 6
}

```


The Assignment Operator

```
class PiNerd
{
public:
    PiNerd(int n) {
        m_n = n;
        m_pi = new int[n];
        for (int j=0;j<n;j++)
            m_pi[j] = getPiDigit(j);
    }

    ~PiNerd() {delete []m_pi;}

    void showOff()
    {
        for (int j=0;j<n;j++)
            cout << m_pi[j] << endl;
    }
private:
    int *m_pi, m_n;
};
```

Ok - so when would we ever need to write our own **Assignment Operator**?

After all, C++ copies all of the fields for us automatically if we don't write our own!

Well, remember our **PiNerd** class...

Lets see what happens if we use the default **assignment operator** with it...

The Assignment Operator

```
class PiNerd
{
public:
    PiNerd(int n) {
        m_n = n;
        m_pi = new int[n];
        for (int j=0;j<n;j++)
            m_pi[j] = getPiDigit(j);
    }

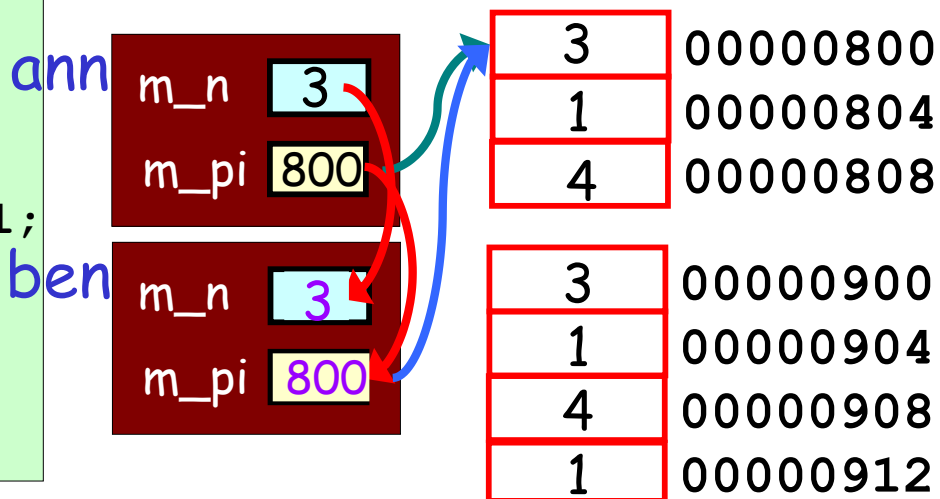
    ~PiNerd(){delete []m_pi;}

    void showOff()
    {
        for (int j=0;j<n;j++)
            cout << m_pi[j] << endl;
    }
private:
    int *m_pi, m_n;
};
```

```
int main()
{
    PiNerd ann(3);

    PiNerd ben(4);

    ben = ann;
}
```

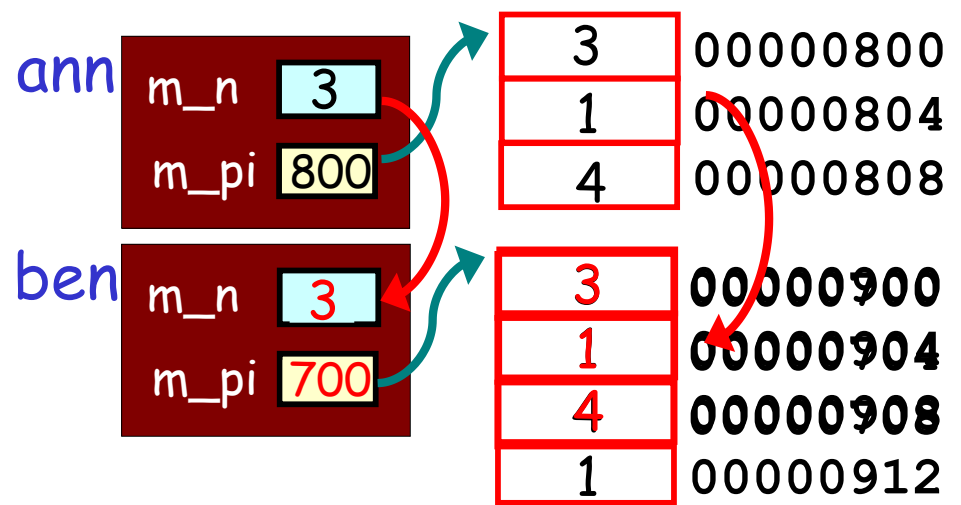


Assignment Operator

For such classes, you **must** define your own **assignment operator!**

Here's how it works for
ben = ann;

1. Free any memory currently held by the target variable (ben).
2. Determine how much memory is used by the source variable (ann).
3. Allocate the same amount of memory in the target variable.
4. Copy the contents of the source variable to the target variable.
5. Return a reference to the target variable.



The Assignment Operator

```

class PiNerd
{
public:
    PiNerd(int n) { ... }
    ~PiNerd() { delete[] m_pi; }

    // assignment operator:
    PiNerd& operator=(const PiNerd &src)
    {
        delete [] m_pi;
        m_n = src.m_n;
        m_pi = new int[m_n];
        for (int j=0; j<m_n; j++)
            m_pi[j] = src.m_pi[j];
        return(*this);
    }

    void showOff() { ... }

private:
    int *m_pi, m_n;
};

```

OK, first let's add a line to **free** the memory used by the **target object**.

```

int main()
{
    → PiNerd ann(3);
    → PiNerd ben(4);
}

```

Next let's determine **how much memory is required** to hold the **source object's data**.

Next we'll add a statement to **allocate enough storage** so the **target** can hold a **copy** of the source's data.

Now we can add statement(s) to **copy over all of the data** from the source to the target variable!

Finally, we'll add a statement so the function **returns a reference to itself** when it's done.
(Don't worry, I'll explain soon)

3	000960
1	000964
4	000968
1	000912

The Assignment Operator

```
class PiNerd
{
public:
    PiNerd(int n) { ... }
    ~PiNerd(){ delete[]m_pi; }

    // assignment operator:
    PiNerd &operator=(const PiNerd &src)
    {
        delete [] m_pi;
        m_n = src.m_n;
        m_pi = new int[m_n];
        for (int j=0;j<m_n;j++)
            m_pi[j] = src.m_pi[j];
        return(*this);
    }
    void showOff() { ... }

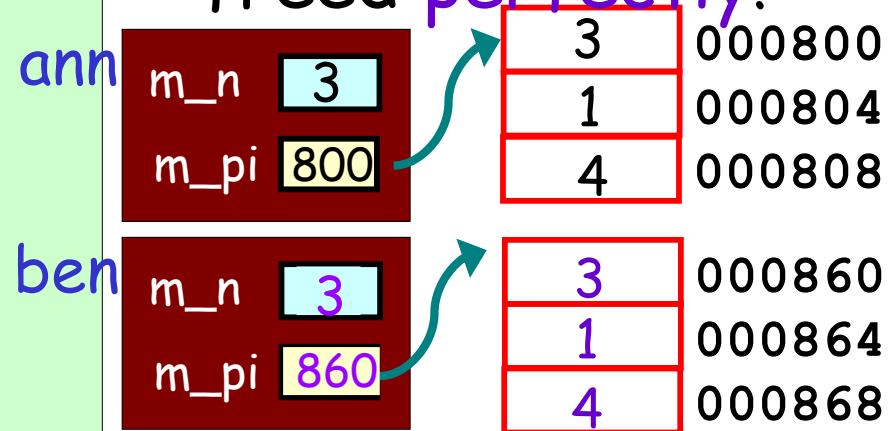
private:
    int *m_pi, m_n;
};
```

```
int main()
{
    PiNerd ann(3);
    PiNerd ben(4);

    ben = ann;

    } // ann's d'tor called, then ben's
```

... and everything is freed perfectly!



The Assignment Operator

tim

```
class Gassy
{
    Gassy &operator= (const Gassy &src)
    {
        m_age = src.m_age;
        m_ateBeans = src.m_ateBeans;
        return(*this);
    }
    m_age 5 m_ateBeans false
};
```

Question: Why do we have `return(*this)` at the end of the assignment operator function?

Answer: So we can do multiple assignments in the same statement, like this...

"this" is a special C++ pointer variable that holds the address of the current object (i.e., ted's address in RAM)

ted

```
class Gassy
{
    Gassy &operator= (const Gassy &src)
    {
        m_age = src.m_age;
        m_ateBeans = src.m_ateBeans;
        return(*this);
    }
    m_age 5 m_ateBeans false
};
```

So if "this" is a pointer to ted, then `*this` refers to the whole ted variable.

So this line returns the ted variable itself! Strange huh? A member function of a variable can return the variable itself!?!?

So the statement: `"ted = sam"` is just replaced by the ted variable!

sam

```
class Gassy
{
```

```
Gassy sam(5, false);
Gassy ted(10, false);
Gassy tim(3, true);
```

```
tim = ted = sam;
```

```
}
```

So, to sum up...
The assignment operator returns `"*this"` so that there's always a variable on the right hand side of the `=` for the next assignment.

```
= src.m_ateBeans;
s);
m_ateBeans false
```

"**Aliasing**" is when we use two different references/pointers to refer to the same variable. It can cause unintended problems!

Our assignment operator has **one more problem** with it... Can anyone guess what it is?

```
class PiNerd
{
public:
    ...

    PiNerd &operator=(const PiNerd &src)
    {
        delete [] m_pi;
        m_n = src.m_n;
        m_pi = new int[m_n];
        for (int j=0; j<m_n; j++)
            m_pi[j] = src.m_pi[j];
        return(*this);
    }

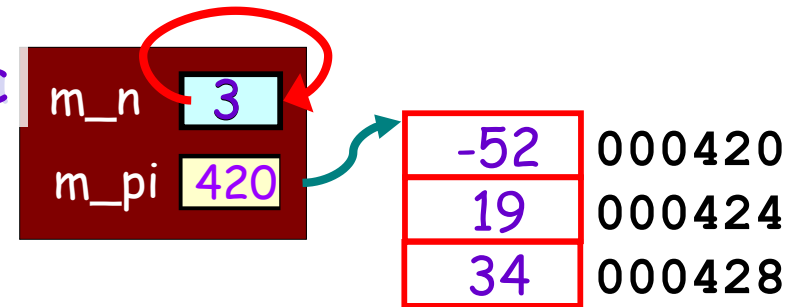
private:
    int *m_pi, m_n;
};
```

Operator

```
void f(PiNerd &x, PiNerd &y)
{
    ...
    x = y; // really ann = ann; !!!
}

int main()
{
    PiNerd ann(3);
    f(ann, ann);
}
```

src



Hmm... What happens if we set **a** to itself?

So now we copy the random values over themselves!

The Assignment Operator

The fix:

Our assignment operator function **must** check to see if a variable is being assigned to itself, and if so, **do nothing**...

If the **right-hand variable's address**...

Is the same as the **left-hand variable's address**...

```
...  
PiNerd & operator=(const PiNerd &src)  
{  
    if (&src == this)  
        return(*this); // do nothing  
    delete [] m_pi;  
    m_n = src.m_n;  
    m_pi = new int[m_n];  
    for (int j=0;j<m_n;j++)  
        m_pi[j] = src.m_pi[j];  
    return(*this);  
}  
...  
};
```

Then they're the **same variable**!
We simply **return a reference** to the variable and do nothing else!

And we're done!

Copy Constructor/ Assignment Review

Question: which of the following use the **copy constructor** and which use the **assignment operator**?

```
int main()           // #1
{
    PiNerd  a(4) , b(3) ;
    b = a;
}
```

```
int main()           // #2
{
    PiNerd  c(5) , d(c) ;
    PiNerd  e = d;
}
```

```
// #3
PiNerd func(void)
{
    PiNerd g(15) ;
    return(g) ;
}

int main()
{
    PiNerd    f = func() ;
}
```

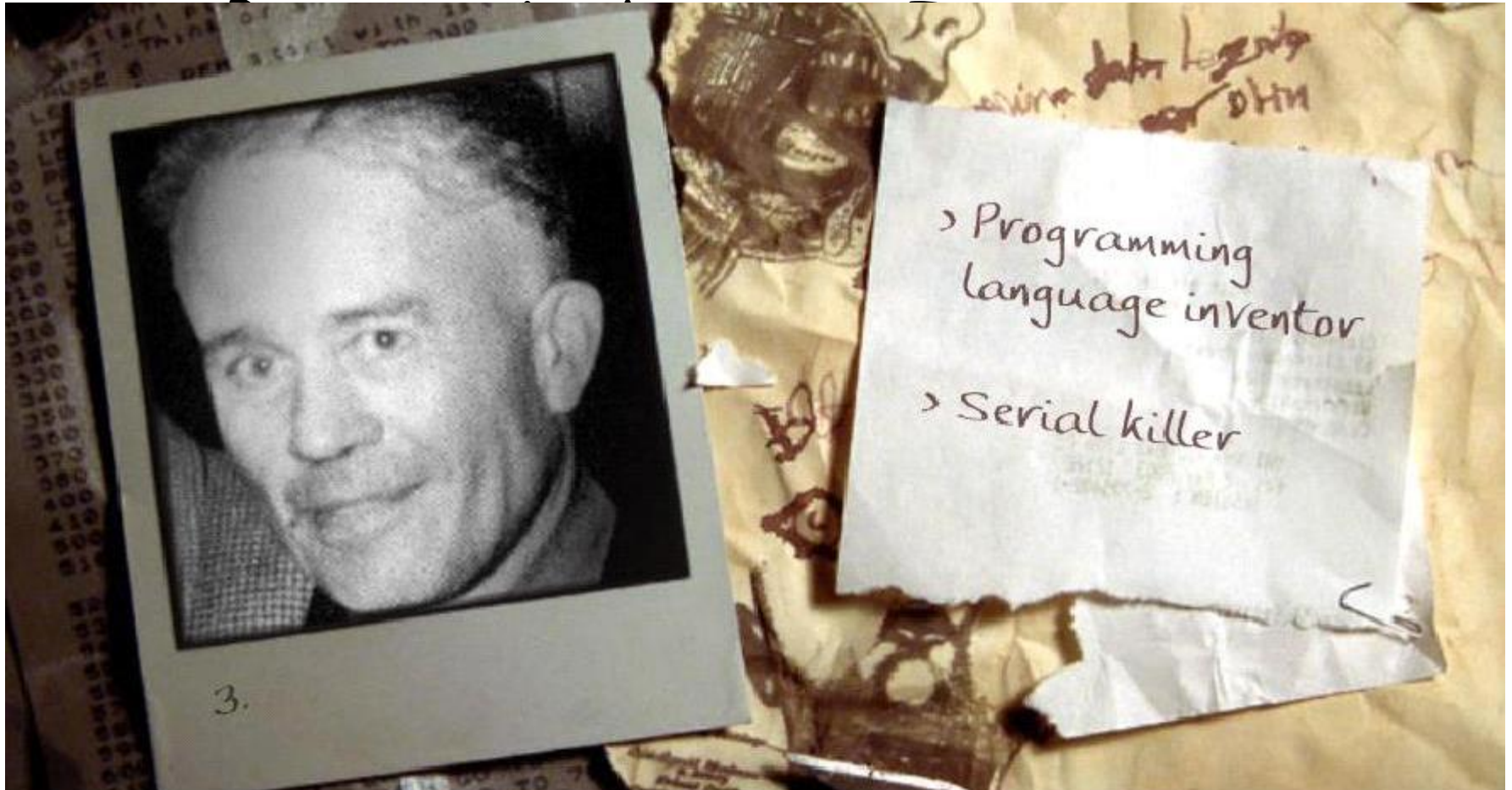
Challenge

Write an assignment operator for our CSNerd Class:

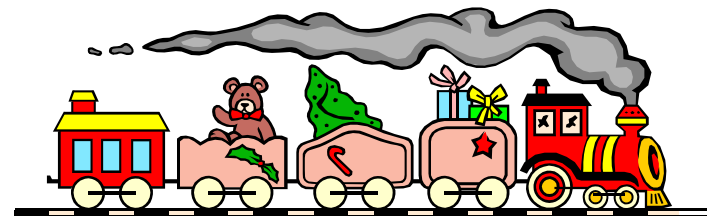
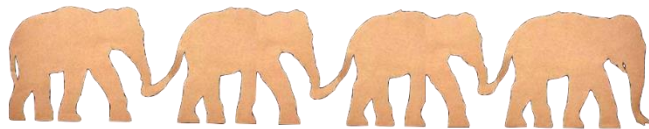
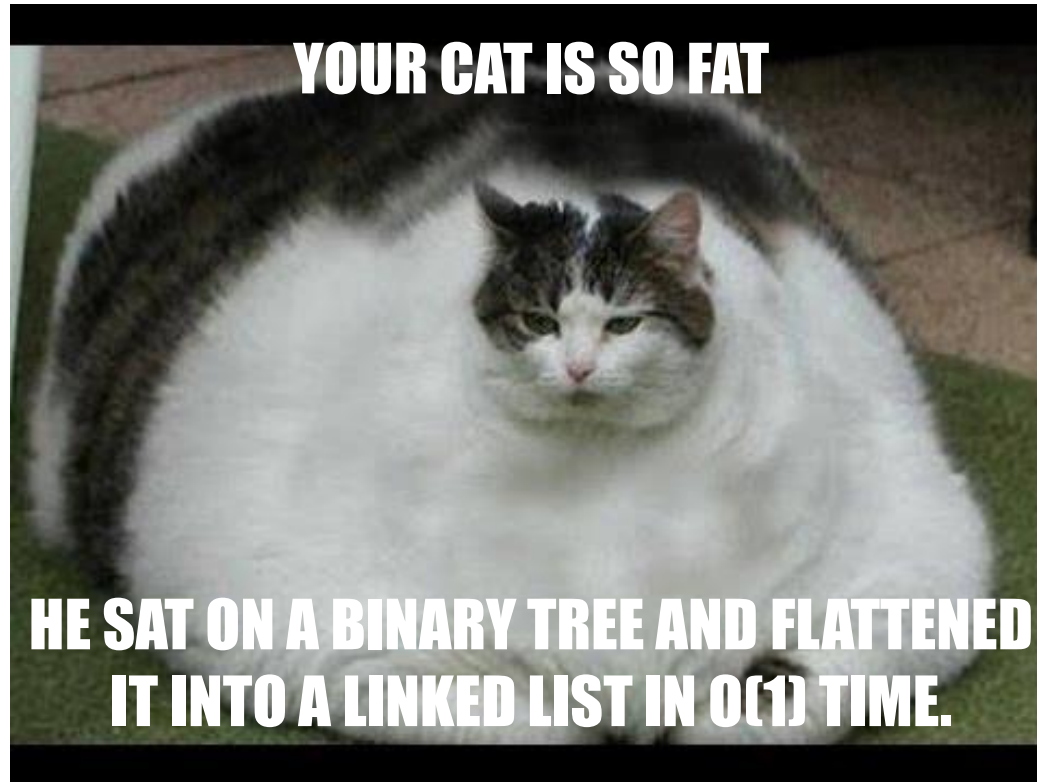
```
struct Book
{
    string title;
    string author;
};
```

```
class CSNerd
{
public:
    CSNerd(string name) {
        m_myBook = nullptr;
        m_myName = name;
    }
    void giveBook(string t, string a) {
        m_myBook = new Book;
        m_myBook->title = t;
        m_myBook->author = a;
    }
    ~CSNerd() {
        delete m_myBook;
    }
private:
    Book *m_myBook;
    string m_myName;
};
```

Time for your favorite game!



Linked Lists



Linked Lists...

Why should you care?

Linked Lists are used in everything from **video games** to **search engines**.

Any time you **don't know how many items** you'll need to store ahead of time, you use 'em.

And virtually every **job interview** will grill you on them.

So pay attention!

Why
should
I care?



Arrays are great... But...

Arrays are great when you need to store a **fixed number of items**...

But what if you **don't know how many** items you'll have ahead of time?

Then you have to **reserve enough slots** for the largest possible case.

Even **new/delete** don't really help!

```
int main()
{
    int array[100];
    ...
}
```

```
int main()
{
    // might have 10 items or 1M
    int array[1000000];
    ...
}
```

```
int main()
{
    int numItems, *ptr;

    cin >> numItems;
    ptr = new int[numItems];
}
```

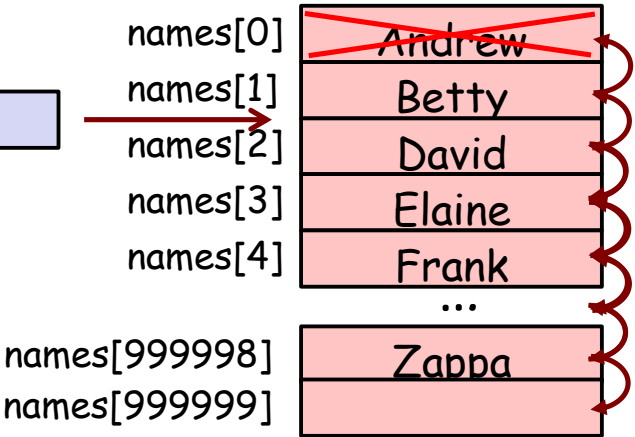
It takes nearly 1M steps to add a new item!

And what if you need to **insert** a new item in the **middle of an array**?

Carey

We have to **move every item** below the insertion spot **down by one**!

And it's just as slow if we want to **delete an item**! Yuck!



So Arrays Aren't Always Great

Hmm... Can we think of an approach from "real life" that works better than a fixed-sized array?

How about organizing the items as we would in a **Scavenger Hunt**?

What can we think of that:

allows you to **store** an arbitrary number of items

makes it fast to **insert** a new item in the middle

makes it fast to **delete** an item from the middle

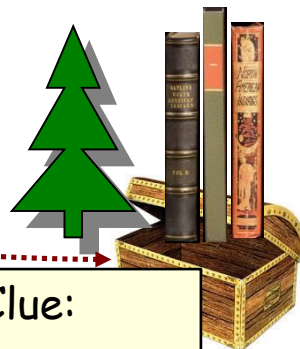
Using this approach we can **store** an arbitrary number of items!

There's no fixed limit to the number of chests and clues we can have!

The hunt starts with a clue to the location of the first chest...

Clue:

The first item is by the tree

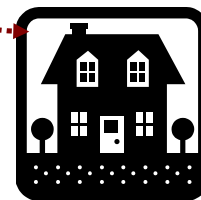


Clue:

The next item is by the house

Clue:

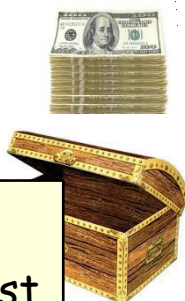
The next item is by the tower



Then each chest holds an item and a clue to the location of the next chest.

Clue:

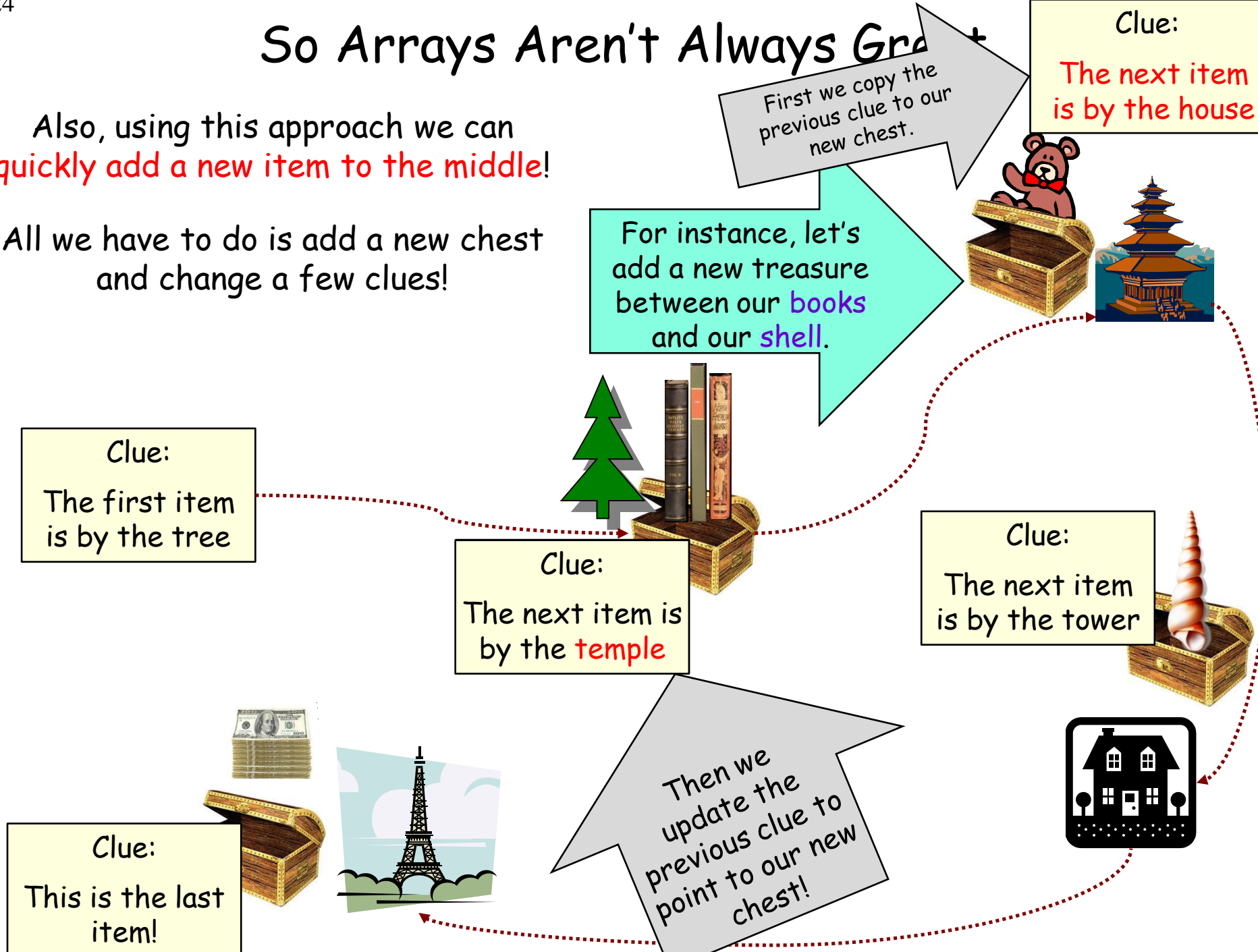
This is the last item!



So Arrays Aren't Always Great

Also, using this approach we can quickly add a new item to the middle!

All we have to do is add a new chest and change a few clues!



So Arrays Aren't Always Great

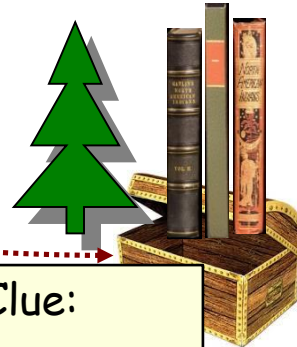
Finally, using this approach we can
quickly remove an item from the middle!

All we have to do is remove the target
chest and change a single clue!

Clue:
The next item
is by the tower



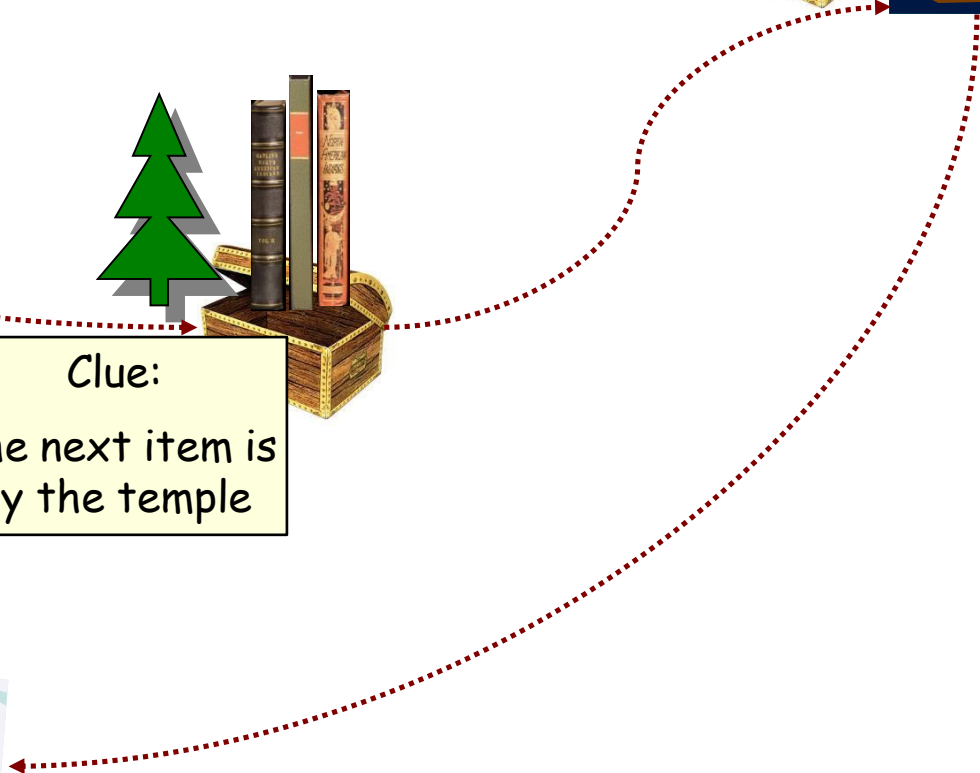
Clue:
The first item
is by the tree



Clue:
The next item is
by the temple



Clue:
This is the last
item!



A C++ Scavenger Hunt?

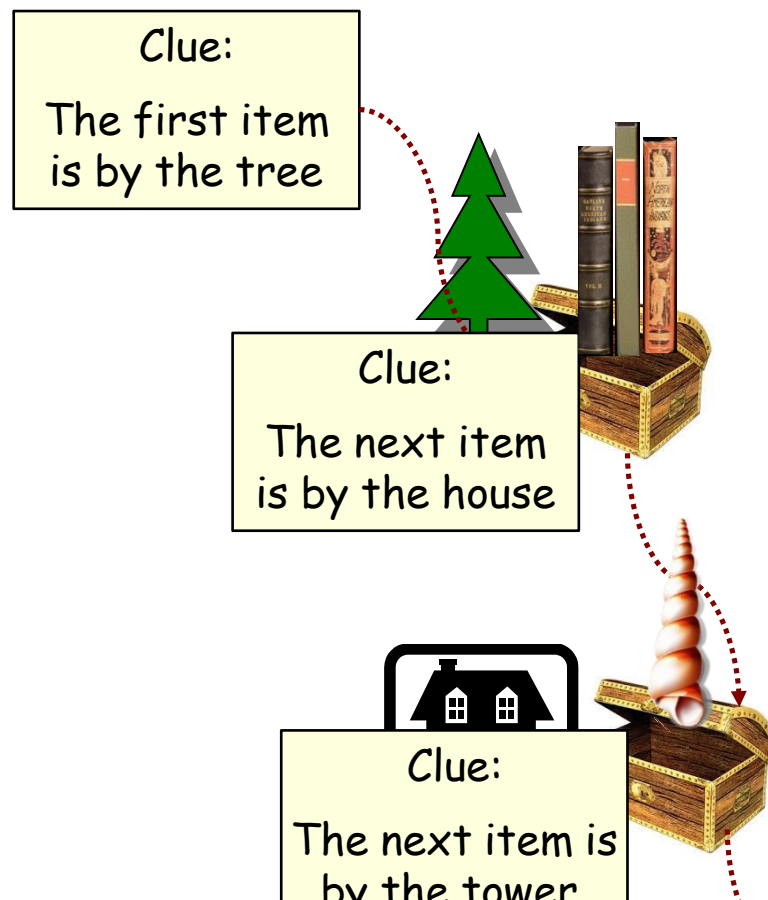
Ok, so in our **Scavenger Hunt**, we had:

A **clue** that leads us to our first treasure **chest**.

Each **chest** then holds an **item** (e.g., books) and a **clue** that leads us to the next chest.

So here's the question... can we simulate a Scavenger Hunt with a C++ **data structure**?

Why not? Let's see how.



A C++ Scavenger Hunt?

Well, we can use a C++ struct to represent a Chest.

As we know, each Chest holds two things:

A treasure - let's use a string variable to hold our treasure, e.g., "shells".

The location of the next chest - let's represent that with a pointer variable.

We can now define a Chest variable for each of the items in our scavenger hunt!

```
struct Chest
{
    string treasure;
    Chest * nextChest;
};
```

This line basically says that each Chest variable holds a pointer... to another Chest variable.

Clue:
The next item is by the house

Clue:
The next item is by the tower



A C++ Scavenger Hunt?

Well, we can use a C++ struct to represent a Chest.

OK, let's see the C++ version of a simplified scavenger hunt data structure!

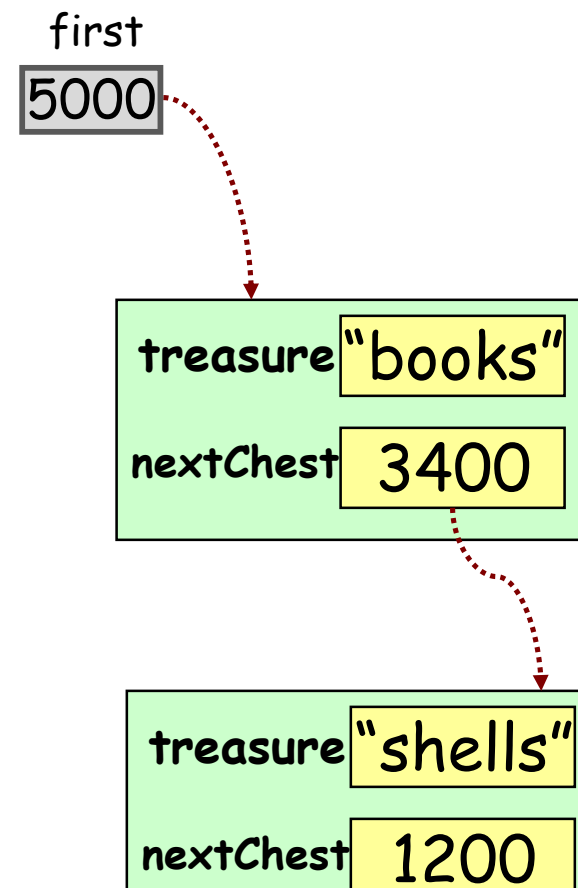
represent that with a pointer variable.

We can now define a Chest variable for each of the items in our scavenger hunt!

And we can define a pointer to point to the very first chest - **our first clue!**

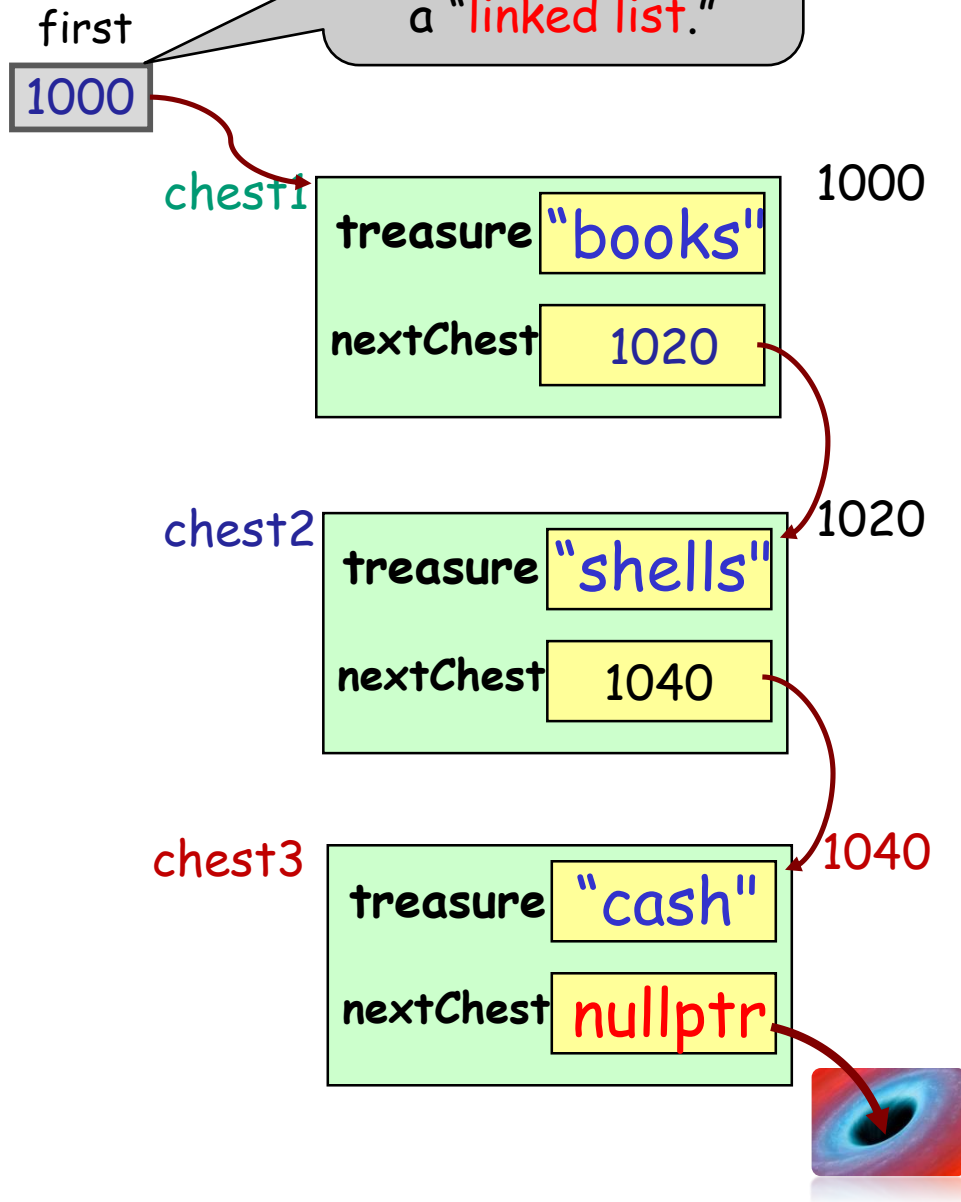
`Chest *first; // pointer to our 1st chest`

```
struct Chest
{
    string treasure;
    Chest * nextChest;
};
```



A C++ Structure? nt?

This data structure is called a "linked list."



```
struct Chest
{
    string treasure;
    Chest * nextChest;
};

int main(void)
{
    Chest *first;
    Chest chest1, chest2, chest3;

    first = &chest1;

    chest1.treasure = "books";
    chest1.nextChest = &chest2;

    chest2.treasure = "shells";
    chest2.nextChest = &chest3;

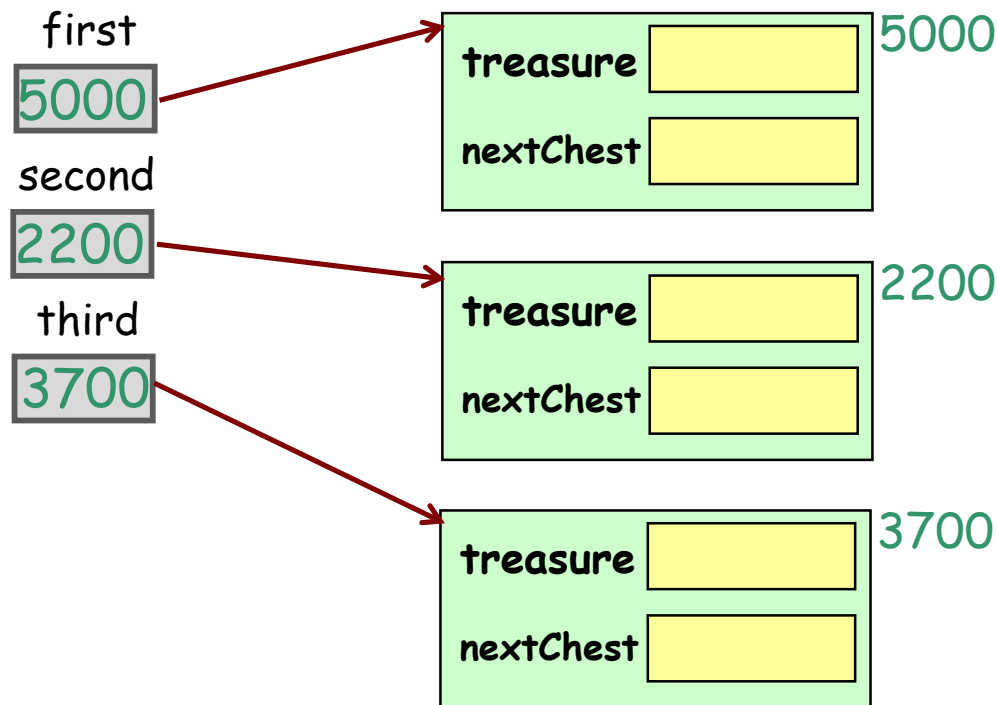
    chest3.treasure = "cash";
    chest3.nextChest = nullptr;

}
```

Linked Lists

Normally, we don't use **local variables** to create our linked list.

Instead we use dynamically-allocated variables (and pointers!).



```
struct Chest
{
    string treasure;
    Chest * nextChest;
};

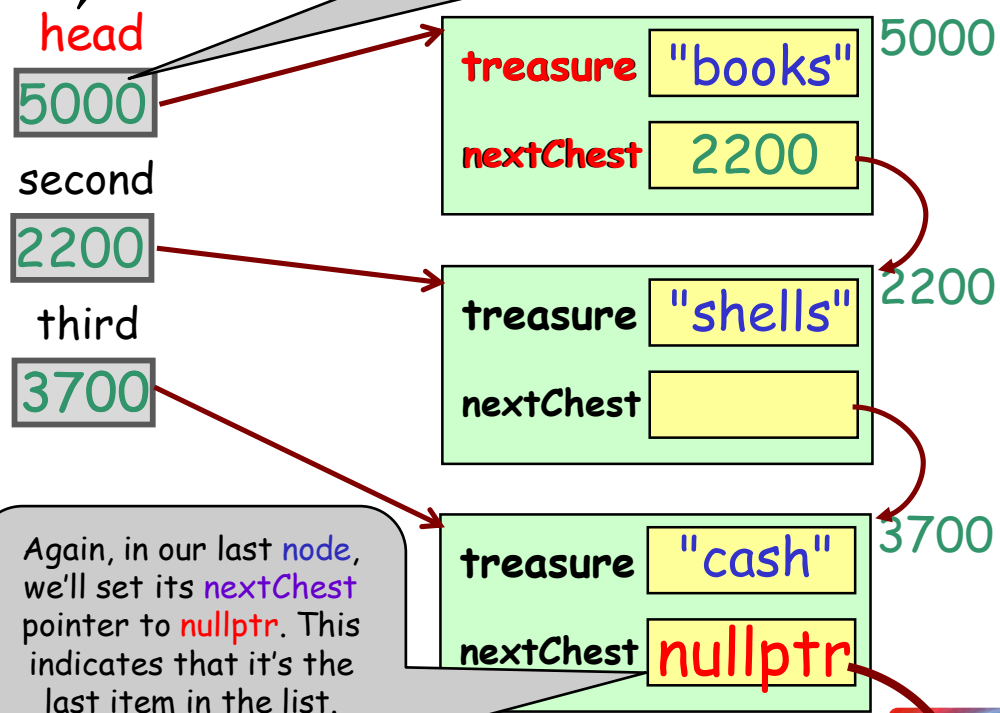
int main(void)
{
    Chest *first, *second, *third;
    first = new Chest;
    second = new Chest;
    third = new Chest;

}
```

The pointer to the top item in the linked list is traditionally called the "**head pointer**."

OK, now let's add cargo and link 'em up!

Given just the **head pointer**, you can reach every element in the list...
without using your other external pointers!



Again, in our last **node**, we'll set its **nextChest** pointer to **nullptr**. This indicates that it's the last item in the list.

struct Chest

```
{
    string treasure;
    Chest * nextChest;
};

int main(void)
{
    Chest *head, *second, *third;
    head = new Chest;
    second = new Chest;
    third = new Chest;

    first->treasure = "books";
    first->nextChest = second;

    second->treasure = "shells";
    second->nextChest = third;

    third->treasure = "cash";
    third->nextChest = nullptr;

    delete head;
    delete second;
    delete third;
}
```



Linked Lists

Ok, it's time to start using the right
Computer Science terms.

Instead of calling them "chests", let's
call each item in the linked list a "Node".

And instead of calling the value held
in a node *treasure*, let's call it "value".

And, instead of calling the linking pointer
nextChest, let's call it "next".

Finally, there's no reason a Node only
needs to hold a single value!

```
struct Node // student node
{
    int studentID;
    string name;
    int phoneNumber;
    float gpa;

    Node *next;
};
```

```
struct Node
{
    string value;
    Node * next;
};
```

```
int main(void)
{
    Node *head, *second, *third;

    head = new Node;
    second = new Node;
    third = new Node;

    head->value = "books";
    head->next = second;

    second->value = "shells";
    second->next = third;

    third->value = "cash";
    third->next = nullptr;

    delete head;
    delete second;
    delete third;
}
```


Note: The delete command
doesn't kill the pointer...

To allocate new nodes:

```
Node *p = new Node;
Node *q = new Node;
```

To change/access a node p's value:

```
p->value = "blah";
cout << p->value;
```

To make node p link to another node
that's at address q:

```
p->next = q;
```

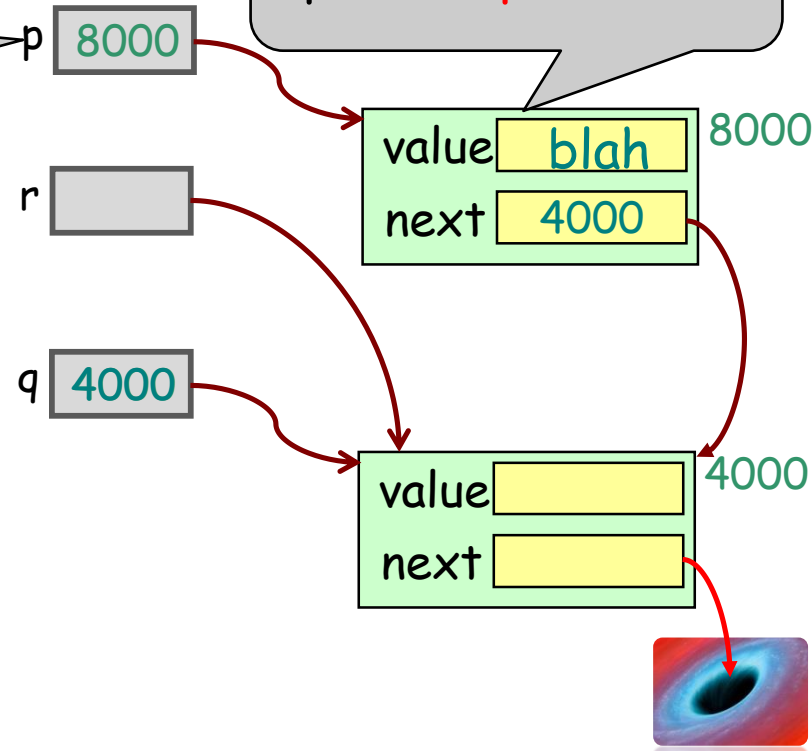
To get the address of the node after p:

```
Node *r = p->next;
```

To make node q a "terminal" node:

```
q->next = nullptr;
```

it kills what the
pointer points to!



To free your nodes:

```
delete p;
delete q;
```

Before we continue, here's a
short recap on what we've
learned:

Linked Lists

Normally, we don't create our linked list all at once in a single function.

After all, some linked lists hold **millions** of items! That wouldn't fit!

Instead, we create a **dedicated class** (an ADT) to hold our linked list...

And then add a bunch of member functions to **add new items (one at a time), process the items, delete items**, etc.

OK, so let's see our new class.

```
struct Node
{
    string value;
    Node * next;
};

int main(void)
{
    Node *head, *second, *third;
    head = new Node;
    second = new Node;
    third = new Node;
    head->value = "books";
    head->next = second;
    second->value = "shells";
    second->next = third;
    third->value = "cash";
    third->next = nullptr;

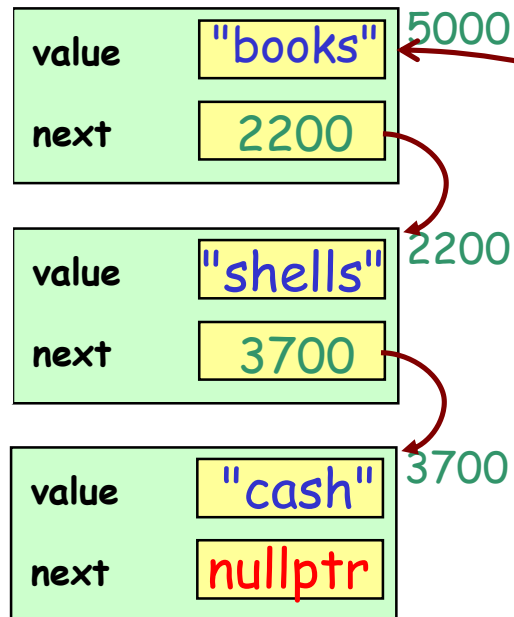
    delete head;
    delete second;
    delete third;
}
```

A Linked List Class!

First, in the simplest type of linked list class, the **only member variable** we need is a **head pointer**.

Why? Given just the head pointer, we can **follow the links** to every node in the list.

And since we can find all the nodes, we can also **link in new ones**, **delete them**, etc..



```
struct Node
```

```
{
```

```
    string value;
```

```
    class LinkedList
```

```
{
```

```
    public:
```

```
private:
```

```
    Node *head;
```

```
};
```

Ok, so let's add a **head pointer** to our class.

5000

A Linked List Class!

Alright, now what **methods** should our linked list class have?

We need a **constructor** to create an empty list...

And methods to **add new items**...

And a method to **delete items**...

And a method to find if an item is in the list...

And a method to **print all the items**...

And finally, we need a **destructor** to free all of our nodes!

Let's consider these one at a time!

```
struct Node
{
    string value;
    Node *next;
};
```

```
class LinkedList
```

```
{
```

```
public:
```

```
    LinkedList() { ... }
```

```
    void addToFront(string v) { ... }
```

```
    void addToRear(string v) { ... }
```

```
    void deleteItem(string v) { ... }
```

```
    bool findItem(string v) { ... }
```

```
    void printItems() { ... }
```

```
    ~LinkedList() { ... }
```

```
private:
```

```
    Node *head;
```

```
};
```

Linked List Constructor

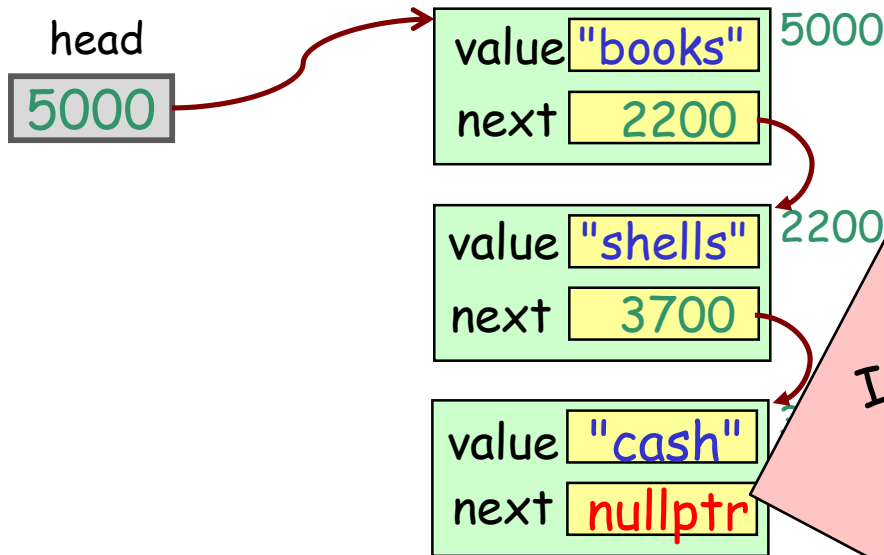
OK, so what should our **constructor** do?

Well, we'll want it to create an "empty" linked list - one with no items.

But how do we create an empty list?

Well, earlier I showed you how we marked the **last node** in a linked list...

We set its **next** value to **nullptr**.



```
struct Node
{
    string value;
    Node *next;
};

class LinkedList
{
public:
    LinkedList()
    {
        ...
    }
    ...
};
```

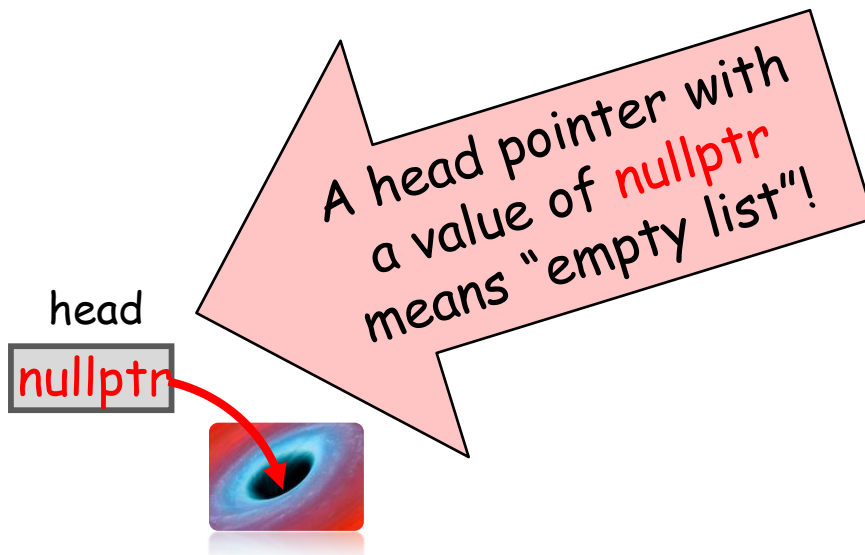
Indicates that there aren't any nodes following this one...

Linked List Constructor

So, following this logic...

We can create an empty linked list by setting our **head** pointer to **nullptr**!

OK, next let's learn how to **print the items** in our list!



```
struct Node
{
    string value;
    Node *next;
};

class LinkedList
{
public:
    LinkedList()
    {
        head = nullptr;
    }

    ...

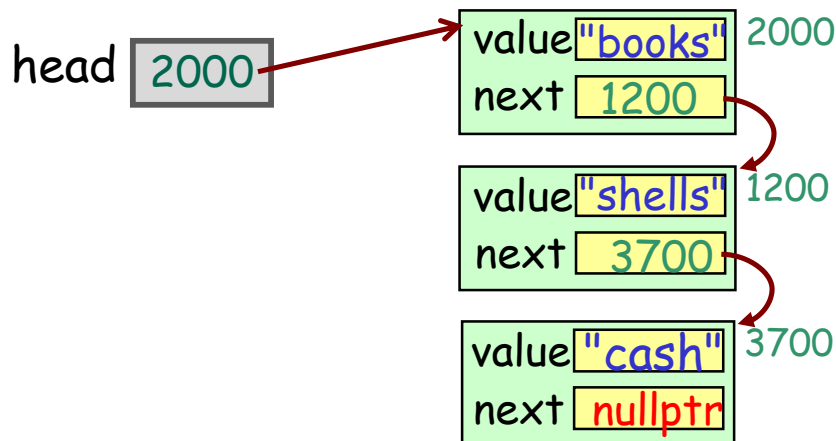
private:
    Node *head;
};
```

Printing the Items in a Linked List

So let's assume we've used our class to create a linked list and add some items...

How do we go about
printing the items in the list?

```
int main()
{
    LinkedList myList;
    // code to add nodes
    myList.printItems();
}
```



```
struct Node
{
    string value;
    Node *next;
};
```

```
class LinkedList
{
public:
```

```
    LinkedList() { ... }
    void addToFront(string v) { ... }
    void addToRear(string v) { ... }
    void deleteItem(string v) { ... }
    bool findItem(string v) { ... }
    void printItems() { ... }
    ~LinkedList() { ... }
```

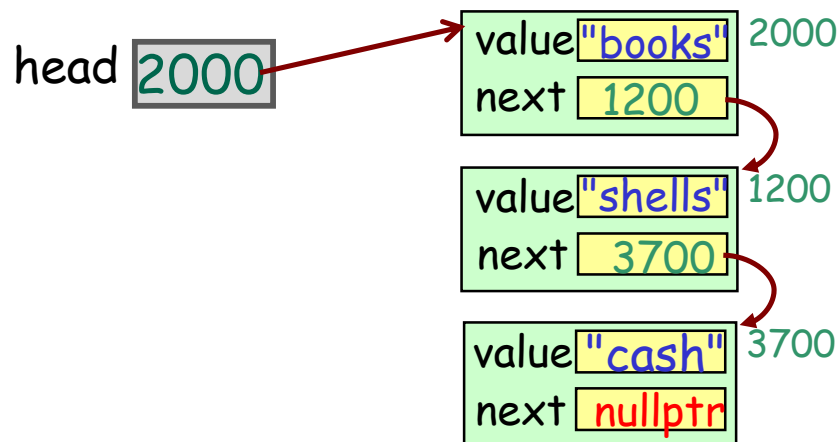
```
private:
    Node *head;
};
```

Printing the Items in a Linked List

So let's assume we've used our class to create a linked list and add some items...

How do we go about
printing the items in the list?

```
int main()
{
    LinkedList myList;
    // code to add nodes
    myList.printItems();
}
```



```
struct Node
{
    string value;
    Node *next;
};

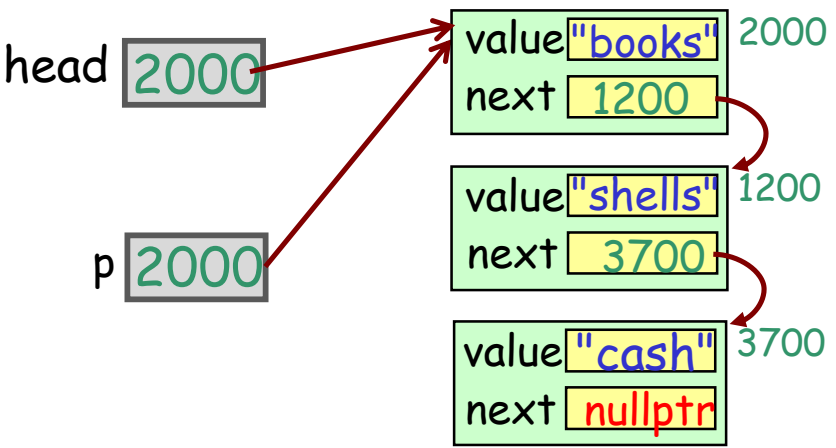
class LinkedList
{
public:
    void printItems()
    {

    }

private:
    Node *head;
};
```


Printing the Items in a Linked List

OK, so our goal is to loop through each of the nodes and print out their values, starting with the node pointed to by "head"...



```

struct Node
{
    string value;
    Node *next;
};

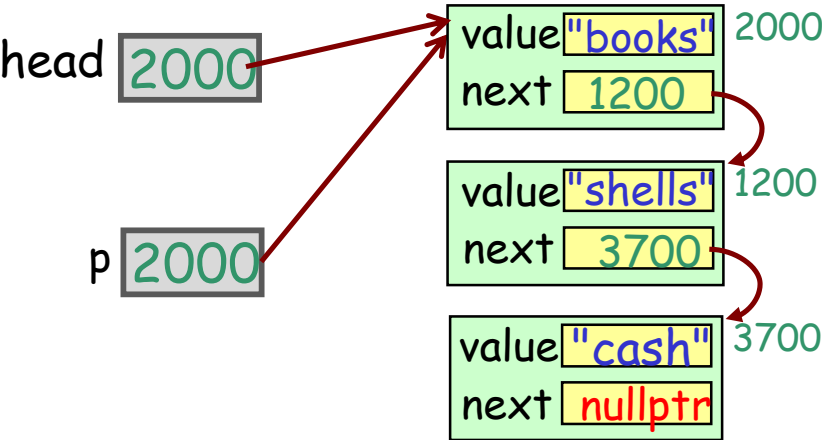
class LinkedList
{
public:
    void printItems()
    {
        Node *p;
        p = head; // p points to 1st node
        while (p points to a valid node )
        {
            print the value in the node
            p = address of the next node
        }
    }

private:
    Node *head;
};
    
```

Printing the Items in a Linked List

OK, so our goal is to loop through each of the nodes and print out their values, starting with the node pointed to by "head"...

Be careful!
 You can't use `p++` to move forward in a linked list!
 You must use the **next pointer**!



```

struct Node
{
    string value;
    Node *next;
};

class LinkedList
{
public:
    void printItems()
    {
        Node *p;
        p = head; // p points to 1st node
        while (p points to a valid node )
        {
            cout << p->value << endl;
            p = p->next;
        }
    }

private:
    Node *head;
};
    
```

Printing the Items in a Linked List

And there's our complete printing loop!

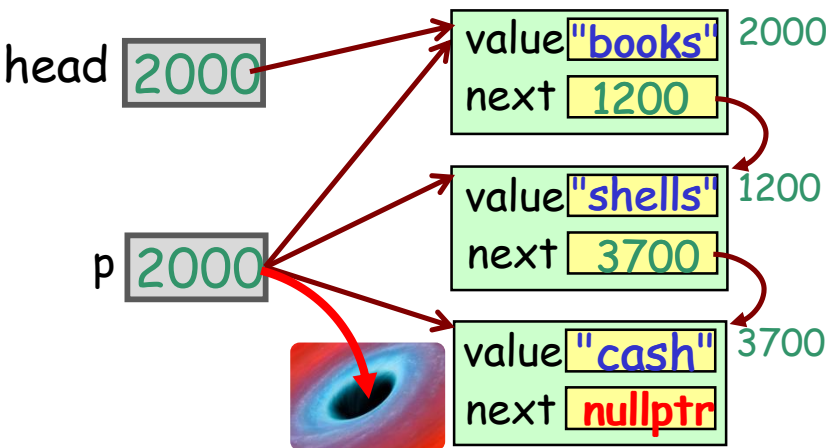
So this answers our question!

If p's value is **nullptr**, it does NOT point to a valid node. Otherwise it does.

Any time we **iterate through one or more nodes** like this, it's called a **"traversal"**.

Alright, now let's learn how to **add nodes** to our list!

This is a linked list traversal!



When we use the condition:
while (p != **nullptr**) { ... }
 the loop will process **EVERY** node in the list and only stop once it's gone **PAST** the end of the list.

```

struct Node
{
    int value;
    Node *next;
};

void printItems()
{
    Node *p;
    p = head; // p points to 1st node
    while ( p != nullptr )
    {
        cout << p->value << endl;
        p = p->next;
    }
}

private:
    Node *head;
};
  
```

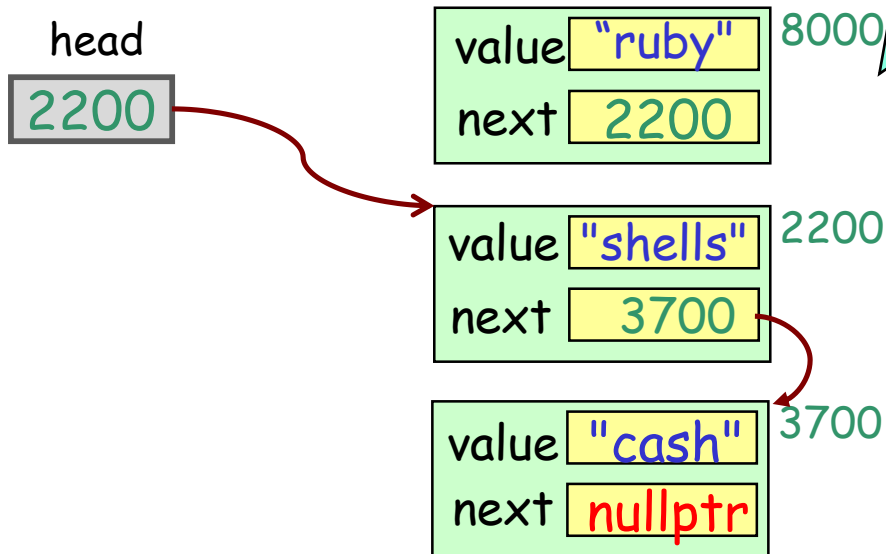
Adding an Item to a Linked List

There are three places you can insert a new item into a linked list:

- at the **top** of the list
- at the **end** of the list
- somewhere in the **middle**

The algorithm to insert at the **top** is the **easiest** to code and also **runs the fastest**.

Let's see this one first, and add a **"ruby"** to the top of our list!



These two steps **must be in this order!**

```

struct Node
{
    string value;
    Node *next;
};

class LinkedList
{
public:
    void addToFront(string v)
    {
        Allocate a new node

        Put value v in the node
        Link the new node to the
        old top node
        Link the head pointer to
        our new top node
    }

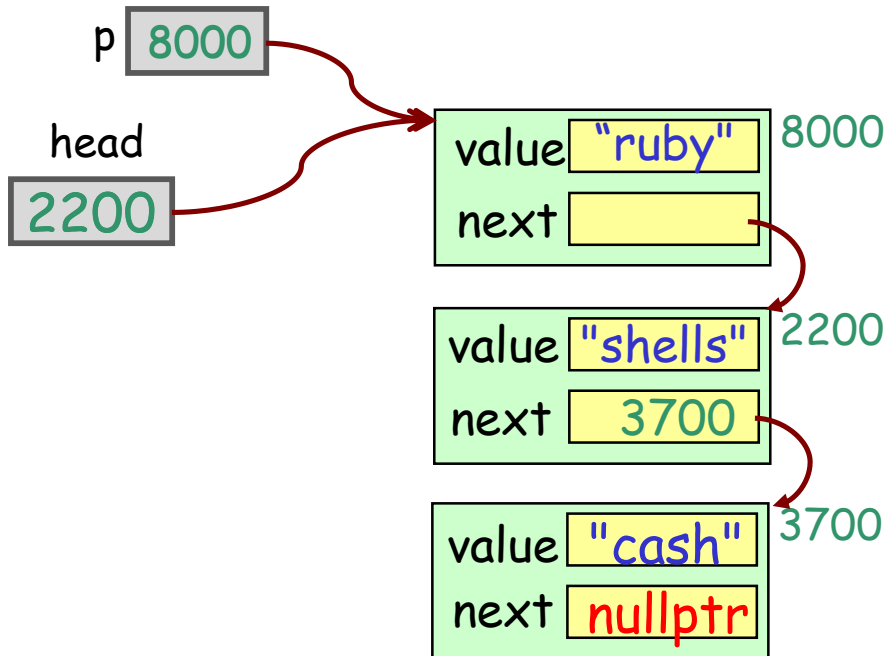
    ...

private:
    Node *head;
};
    
```

Adding an Item to the Front

OK, now let's replace our psuedo-code with valid C++ code.

And as you can see, our new node has been added at the top!



```

struct Node
{
    string value;
    Node *next;
};

class LinkedList
{
public:
    void addToFront(string v)
    {
        Node *p;
        p = new Node;
        p->value = v; // put v in node

        p->next = head;

        head = p;
    }

    ...

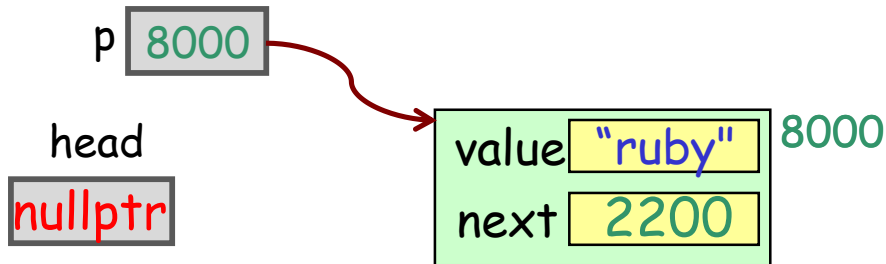
private:
    Node *head;
};
  
```

Adding an Item to the Front

OK, but will this same algorithm work if the Linked List is **empty**?

Let's see!

Pretty cool - the same algorithm works whether the list is empty or not!



Alright, now let's see how to **add a node to the rear** of a list!

```

struct Node
{
    string value;
    Node *next;
};

class LinkedList
{
public:
    void addToFront(string v)
    {
        Node *p;
        p = new Node;
        p->value = v; // put v in node

        p->next = head;

        head = p;
    }

    ...

private:
    Node *head;
};
  
```

Adding an Item to the Rear

Alright, next let's look at how to **append an item** at the **end of a list**...

There are actually **two cases** to consider:

Case #1:

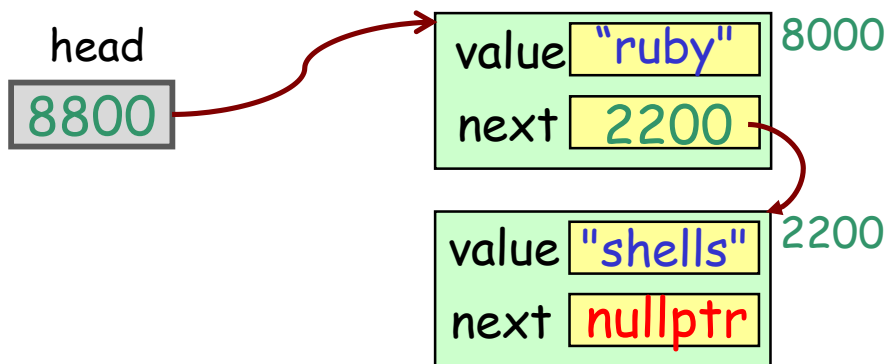
The existing list is **totally empty!**

head

nullptr

Case #2:

The existing list **has one or more nodes**...



```

struct Node
{
    string value;
    Node *next;
};

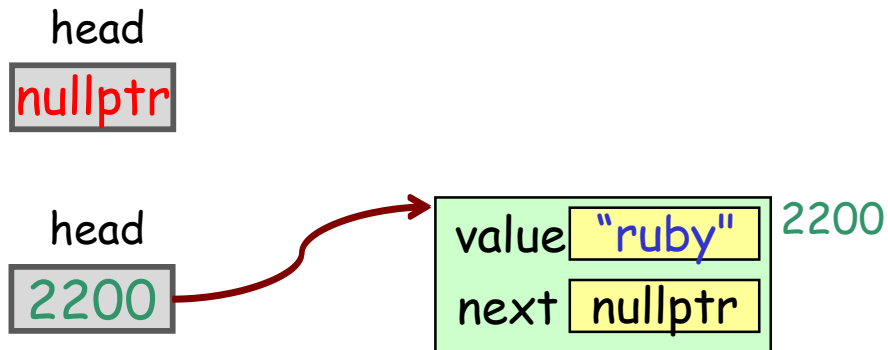
class LinkedList
{
public:
    void addToRear(string v)
    {

    }

    ...
};
  
```

Adding an Item to the Rear

Alright, let's consider Case #1 first...
It's much easier!



So how do you **add a new node** to the **end** of an empty linked list?

In fact, it's the same as **adding a new node** to the **front** of an empty linked list.

Which we just learned two minutes ago!

After all, in both cases we're adding a node right at the top of the linked list.

```
struct Node
{
    string value;
    Node *next;
};

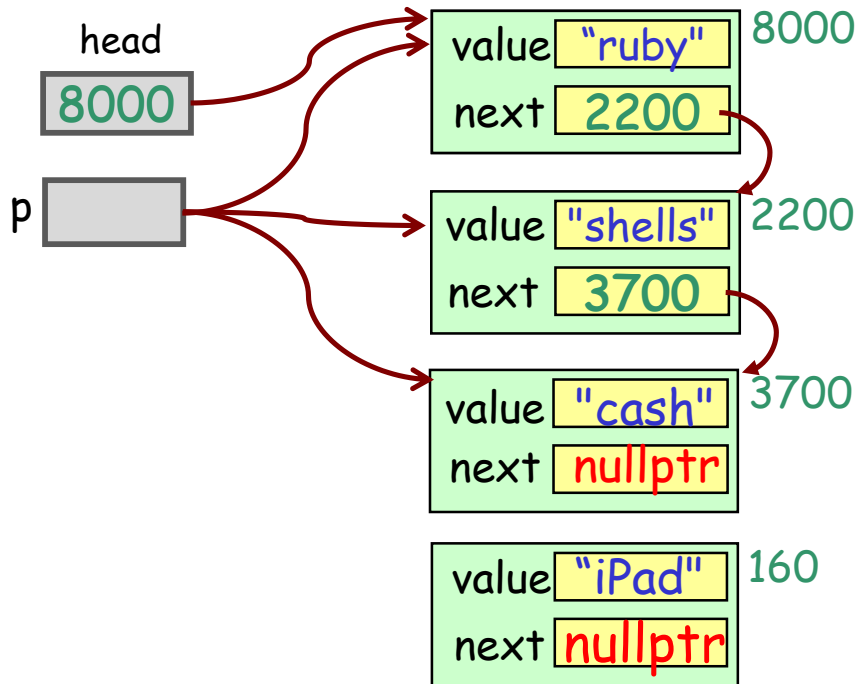
class LinkedList
{
public:
    void addToRear(string v)
    {
        if (head == nullptr)
            addToFront(v); // easy!!!
    }
};
```


Adding an Item to the Rear

Alright, let's consider Case #2 next:
It's more **complex**...

Here we want to **add an item** to the end
of a linked list that **already has nodes**.

Well that doesn't look too bad... Let's add
an "iPad" to our list.



```

struct Node
{
    string value;
    Node *next;
};

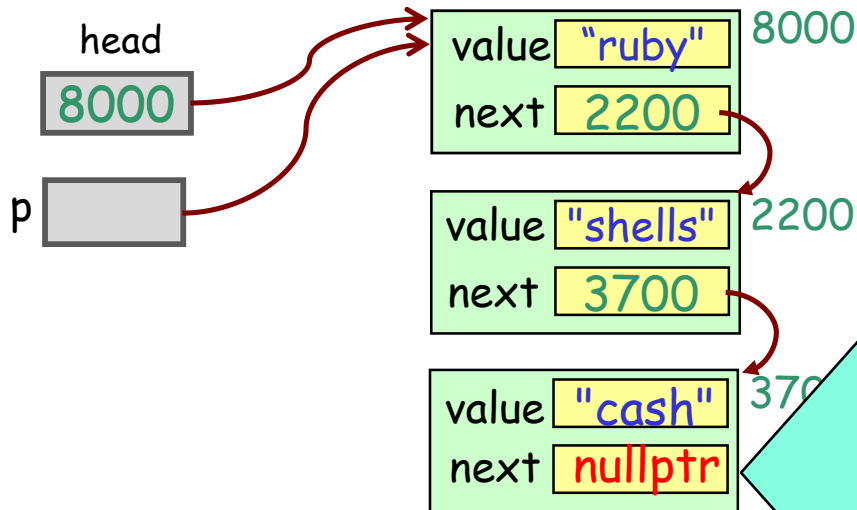
class LinkedList
{
public:
    void addToRear(string v)
    {
        if (head == nullptr)
            addToFront(v); // easy!!!
        else
        {
            Use a temp variable to
            traverse to the current
            last node of the list

            Allocate a new node
            Put value v in the node
            Link the current last node
            to our new node
            Link the last node to nullptr

        }
        ...
    };
  
```

Adding an Item to the Rear

OK, let's see the C++ code now!



We want to stop looping when p points at this node

```
struct Node
{
    string value;
    Node *next;
};

class LinkedList
{
public:
    void addToRear(string v)
    {
        if (head == nullptr)
            addToFront(v); // easy!!!
        else
        {
            Node *p;
            p = head; // start at top node
            while( p->next != nullptr )
                p = p->next;

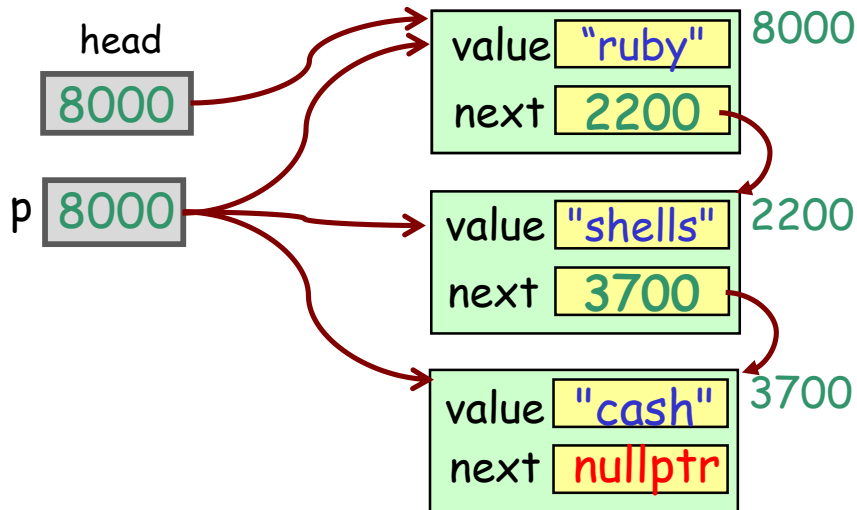
            Allocate a new node
            Put value v in the node
            Link the current last node
            to our new node
            Link the last node to nullptr

        }
    }
};
```

Adding an Item to the Rear

OK, let's see the C++ code now!

Alright, let's finish up our function!



```

struct Node
{
    string value;
    Node *next;
};

class LinkedList
{
public:
    void addToRear(string v)
    {
        if (head == nullptr)
            addToFront(v); // easy!!!
        else
        {
            Node *p;
            p = head; // start at top node
            while( p->next != nullptr )
                p = p->next;

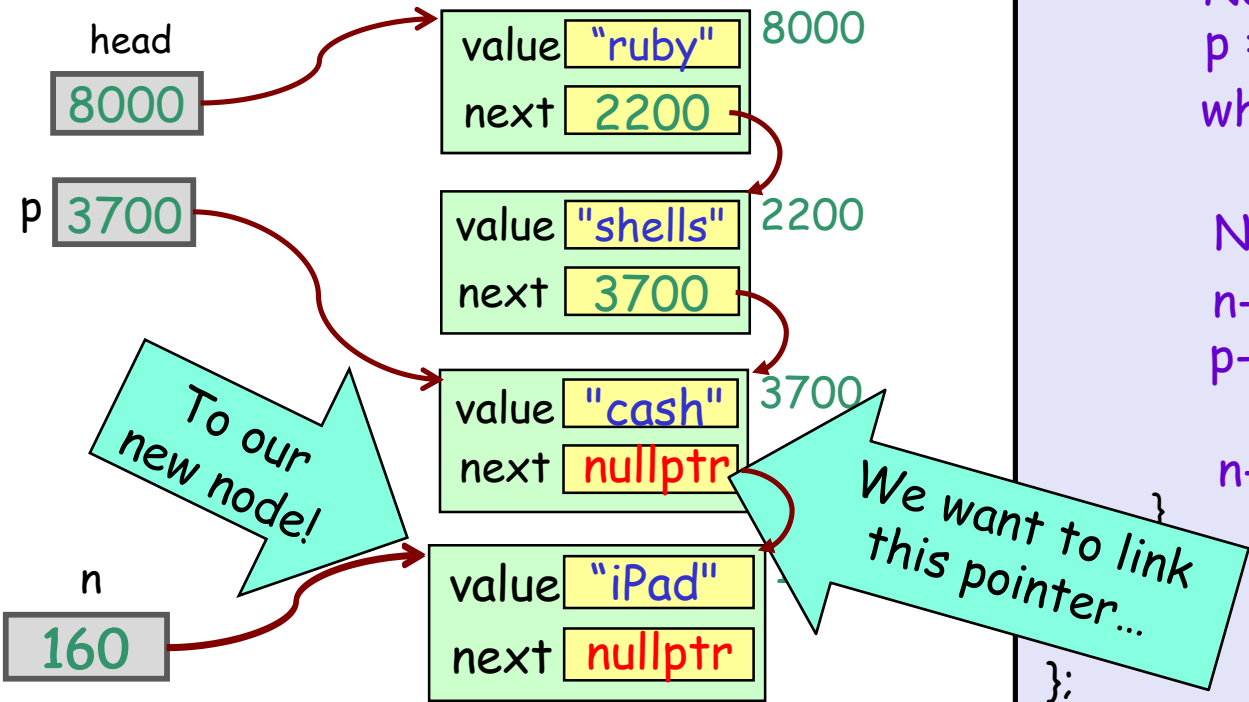
            // Allocate a new node
            // Put value v in the node
            // Link the current last node
            // to our new node
            // Link the last node to nullptr

        }
    }
};
  
```

Adding an Item to the Rear

OK, let's see the C++ code now!

Alright, let's finish up our function!



```

struct Node
{
    string value;
    Node *next;
};

When we use the condition:
while (p->next != nullptr) { ... }
the loop continues until p points at
the very last node of the list.

// ...
front(v); // easy!!!
else
{
    Node
    p = head; // start at top node
    while( p->next != nullptr )
        p = p->next;

    Node *n = new Node;
    n->value = v;
    p->next = n;

    n->next = nullptr;
}
};
    
```

Not at the top, not at the bottom...

In some cases, we won't always want to just add our node to the **top** or **bottom** of the list... Why?

Well, what if we have to maintain an **alphabetized** linked list, or we want to allow the user to **pick the spot** to put each item?

In these cases, we can't just add new items at the top or bottom...

Here's the basic algorithm:

```
void AddItem(string newItem)
{
    if (our list is totally empty)
        Just use our addToFront() method to add the new node
```

```
}
```

head **nullptr**

value	"bat"	600
next	nullptr	

Not at the top, not at the bottom...

bat belongs here, above *cat*.

value "bat" 600
next 1000

head 1000

value "cat" 1000
next 1400

value "dog" 1400
next 800

value "rat" 800
next nullptr

Here's the basic algorithm:

```
void AddItem(string newItem)
{
    if (our list is totally empty)
        Just use our addToFront() method to add the new node
    else if (our new node belongs at the very top of the list)
        Just use our addToFront() method to add the new node
}
```

Not at the top, not at the bottom...

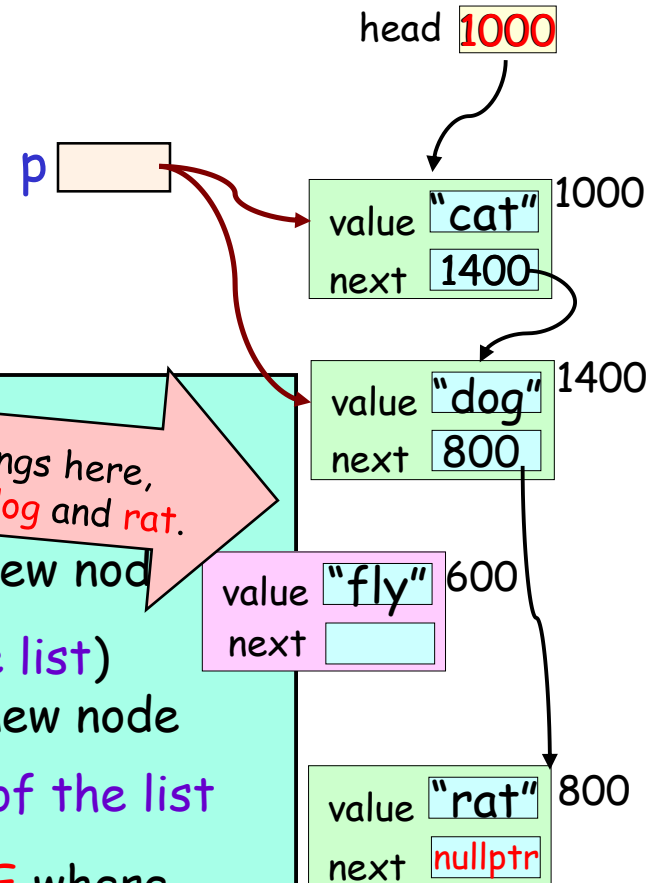
Here's the basic algorithm:

```
void AddItem(string newItem)
{
    if (our list is totally empty)
        Just use our addToFront() method to add the new node
    else if (our new node belongs at the very top of the list)
        Just use our addToFront() method to add the new node
    else // new node belongs somewhere in the middle of the list
    {
        Use a traversal loop to find the node just ABOVE where
        you want to insert our new item

        Allocate and fill our new node with the item

        Link the new node into the list right after the ABOVE node
    }
}
```

fly belongs here,
between dog and rat.



Let's Convert it to

```

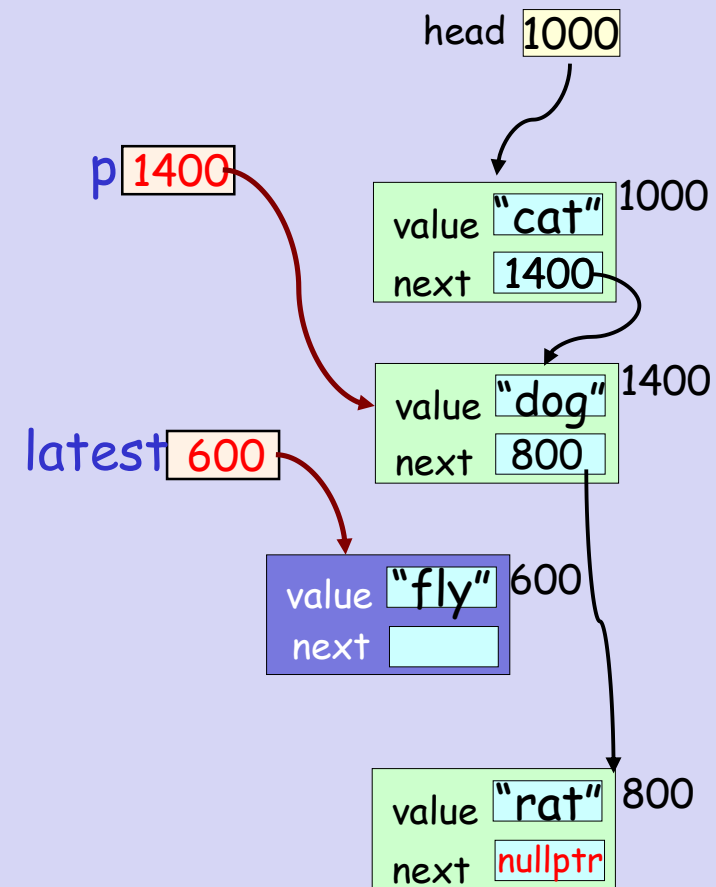
void AddItem(string newItem)
{
    if (head == nullptr)
        AddToFront(newItem);

    else if ( /* decide if the new item belongs at the top */ )
        AddToFront(newItem);

    else // new node belongs somewhere in the middle
    {
        Node *p = head; // start with top node
        while (p->next != nullptr)
        {
            if ( /* p points just above where I want to insert */ )
                break; // break out of the loop!

            p = p->next; // move down one node
        }
        Node *latest = new Node; // alloc and fill our new node
        latest->value = newItem;
        latest->next = p->next; // link new node to the node below p
        p->next = latest; // link node above to our new node
    }
}

```



These two lines
must
be in this
order!

Let's Convert i

Finally, let's fill in the blanks to convert our function into one that adds items in alphabetical order!

```
void AddItem(string newItem)
{
    if (head == nullptr)
        AddToFront(newItem);

    else if ( newItem < head->value )
        AddToFront(newItem);

    else // new node belongs somewhere in the middle of the list
    {
        Node *p = head; // start with first node
        while (p->next != nullptr)
        {
            if ( newItem >= p->value && newItem <= p->next->value )
                break;

            p = p->next; // move down one node
        }
        Node *latest = new Node; // alloc and fill our new node
        latest->value = newItem;
        latest->next = p->next; // link new node to node below
        p->next = latest; // link above node to our new node
    }
}
```

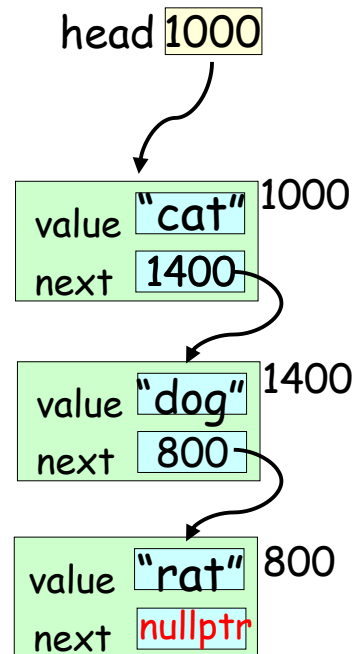
Deleting an Item in a Linked List

When deleting an item from a linked list, there are **two different cases** to consider:

Case #1: You're deleting the **first node**.

Case #2: You're deleting an **interior node** or the **last node**.

Let's consider **Case #1** first...



```

struct Node
{
    string value;
    Node *next;
};

class LinkedList
{
public:
    void deleteItem(string v)
    {

    }

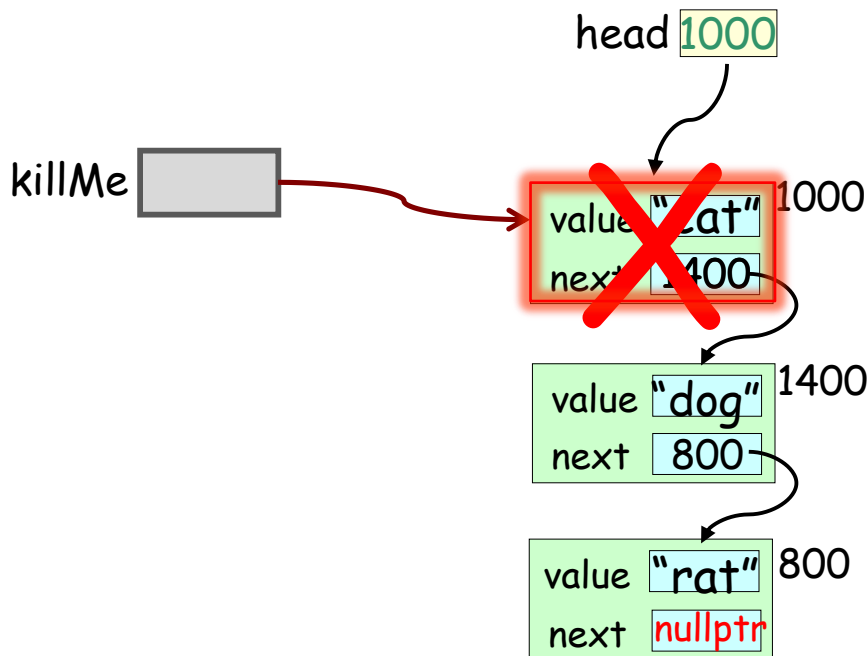
    ...

};
  
```

Deleting an Item in a Linked List

Ok, let's consider Case #1...
deleting the top item in a list.

Let's kill our **cat**.



```
struct Node
{
    string value;
    Node *next;
};
```

```
class LinkedList
{
public:
```

```
    void deleteItem(string v)
    {
```

If the list's empty then return

If the first node holds the
item we wish to delete then

```
{
    killMe = address of top node
```

Update head to point to the
second node in the list

Delete our target node

Return - we're done

```
}
```

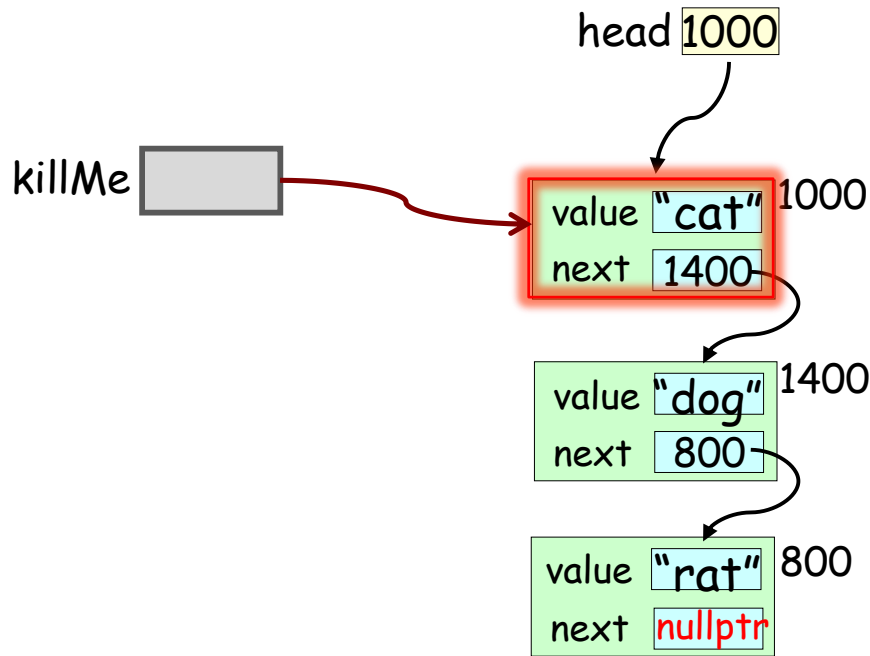
```
}
```

```
...
```

```
};
```

Deleting an Item in a Linked List

OK, let's see the C++ code now!



```

struct Node
{
    string value;
    Node *next;
};

class LinkedList
{
public:
    "cat"
    void deleteItem(string v)
    {
        if (head == nullptr) return;

        if (head->value == v)

            Node *killMe = head;

            head = killMe->next;

            delete killMe;

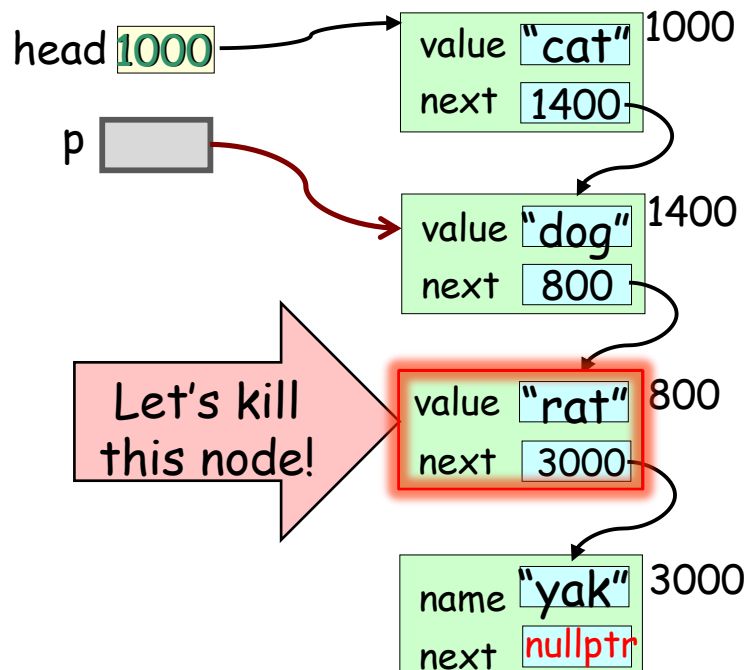
            return;

    }
    ...
};
  
```

Deleting an Item in a Linked List

Alright, let's consider Case #2 next - unfortunately it's more complex...

Let's kill our "rat" node!



```

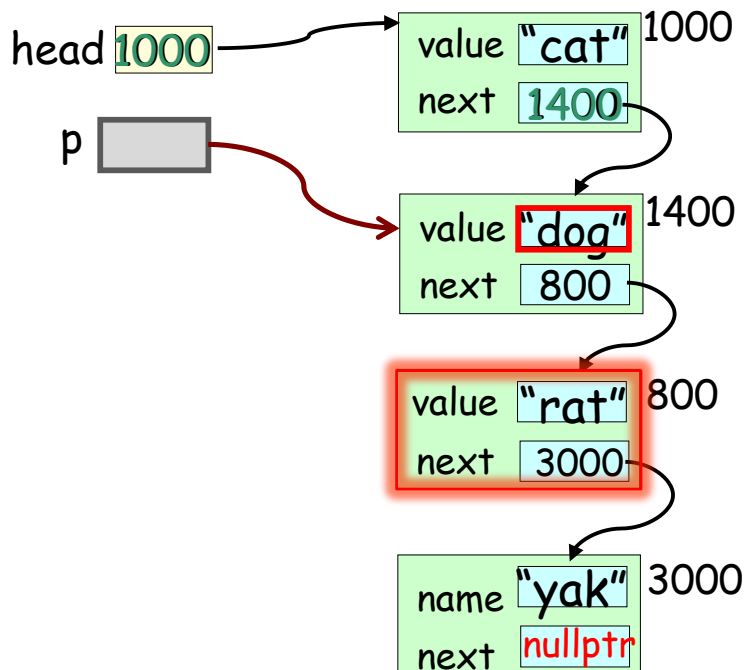
struct Node
{
    string value;
    Node *next;
};

class LinkedList
{
public:
    void deleteItem(string v)
    {
        ... // the code we just wrote
        Use a temp pointer to traverse
        down to the node above the
        one we want to delete...

        If we found our target node
        {
            killMe = addr of target node
            Link the node above to
            the node below
            Delete our target node
        }
    }
    ...
};
  
```

Deleting an Item in a Linked List

OK, let's see the C++ code now!



```

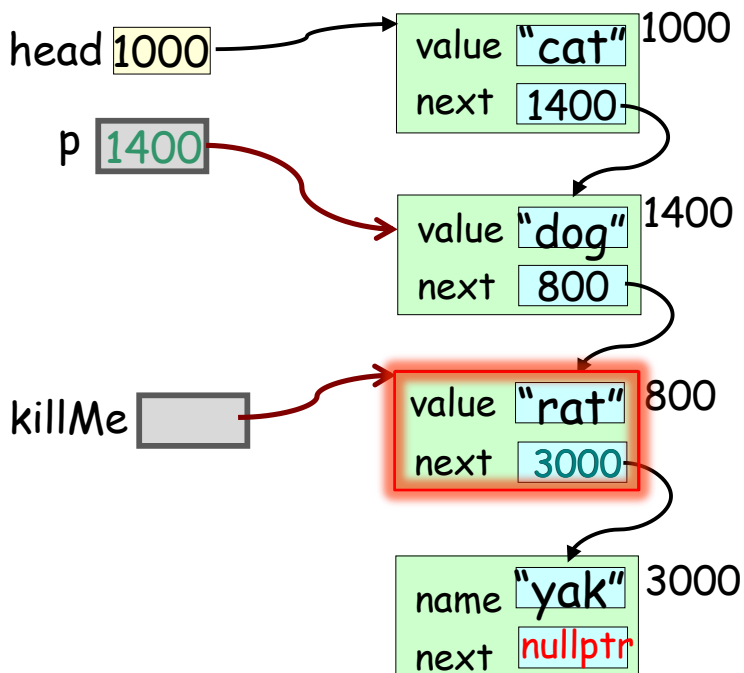
struct Node
{
    string value;
    Node *next;
};

class LinkedList
{
public:
    void deleteItem(string v)
    {
        ... // the code we just wrote
        Node *p = head;
        while (p != nullptr)
        {
            if ( p->next != nullptr &&
                p->next->value == v )
                break; // p pts to node above
            p = p->next;
        }
        If we found our target node
        {
            killMe = addr of target node
            Link the node above to
            the node below
            Delete our target node
        }
        ...
    };
};
  
```

Deleting an Item in a Linked List

OK, let's see the C++ code now!

And believe it or not, this same code works when the **target node** is the **last one in the list!**



```
struct Node
{
    string value;
    Node *next;
};

class LinkedList
{
public:
    "rat"
    void deleteItem(string v)
    {
        ... // the code we just wrote
        Node *p = head;
        while (p != nullptr)
        {
            if ( p->next != nullptr &&
                p->next->value == v )
                break; // p pts to node above
            p = p->next;
        }
        if (p != nullptr) // found our value!
            Node *killMe = p->next;
            p->next = killMe->next;
            delete killMe;
        }
        ...
    };
};
```

Linked List Challenge

Now it's your turn!

How would you write the `findItem()` method?

It should return `true` if it can find the passed-in item, and `false` otherwise.

```
int main()
{
    Linked List myFriends;
    myFriends.addToFront("David");
    ...

    if (myFriends.findItem("Carey") == true)
        cout << "I'm so lucky!\n";
}
```

```
struct Node
{
    string value;
    Node *next;
};

class LinkedList
{
public:
    bool findItem(string v)
    {

    }

    ...

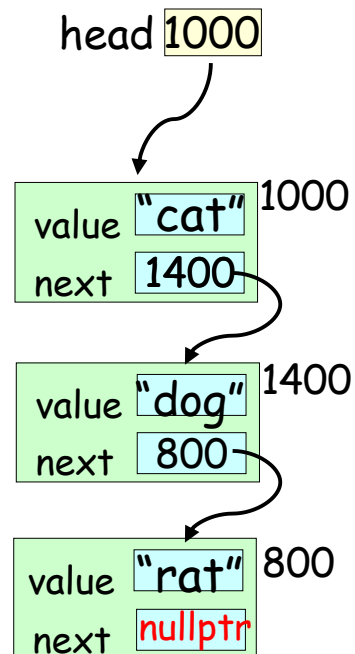
private:
    Node *head;
};
```


Destructing a Linked List

OK, so how do we **completely destruct** a linked list once we're done with it?

Well, perhaps we can use something like our existing **printItems()** code?

Let's see what happens!



```

struct Node
{
    string value;
    Node *next;
};

class LinkedList
{
public:
    ~LinkedList()
    {
        Node *p;
        p = head;
        while (p != nullptr)
        {
            delete p;
            p = p->next;
        }
        ...
    }

private:
    Node *head;
};
  
```

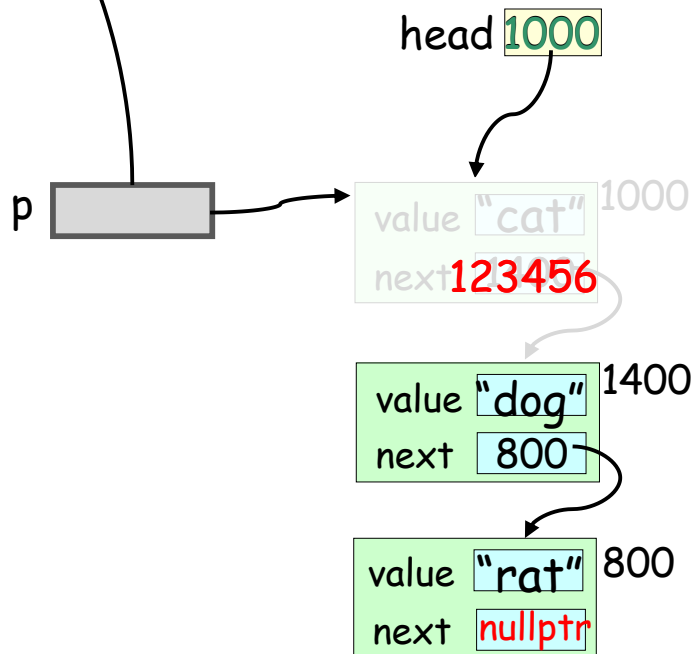


Linked List

we completely destruct a
we we're done with it?

Well, perhaps we can use something like
our existing `printItems()` code?

Let's see what happens!



```
struct Node
{
    string value;
    Node *next;
};
```

```
class LinkedList
{
public:
```

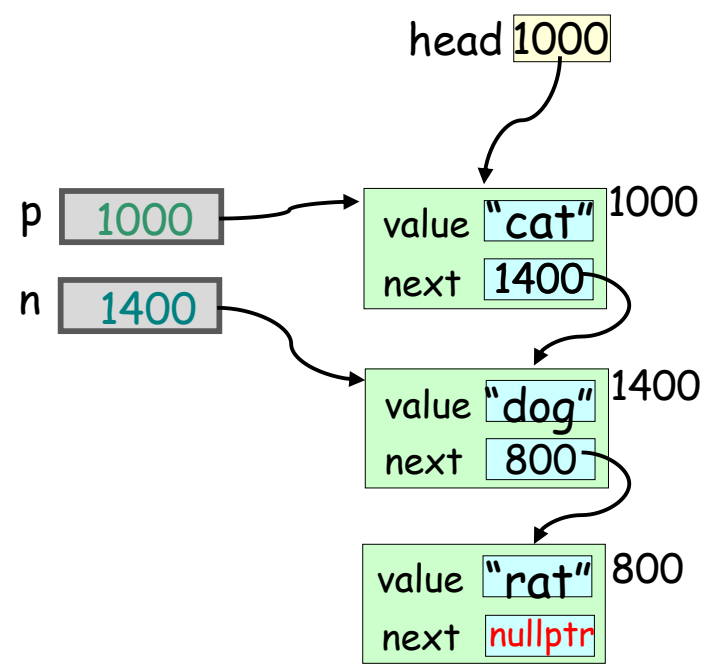
```
    ~LinkedList()
    {
        Node *p
        p = head;
        while (p != nullptr)
        {
            delete p;
            p = p->next;
        }
    }
    ...
```

Houston... We have
a problem!

```
    Node *head;
};
```

Destructing a Linked List

OK, let's fix it.



```

struct Node
{
    string value;
    Node *next;
};

class LinkedList
{
public:
    ~LinkedList()
    {
        Node *p;
        p = head;
        while (p != nullptr)
        {
            Node *n = p->next;
            delete p;
            p = n;
        }
        ...
    }

private:
    Node *head;
};
  
```

Linked Lists Aren't Perfect!

As you can already tell, linked lists aren't perfect either!

First of all, they're **much more complex** than arrays!!!



Second, to **access the k^{th} item**, I have to **traverse down $k-1$ times** from the head first! **No instant access!!!**

And to **add an item at the end** of the list... I have to **traverse through all N existing nodes** first!



Well, as it turns out, we can **fix this last problem**... Let's see how!

Linked Lists and Tail Pointers

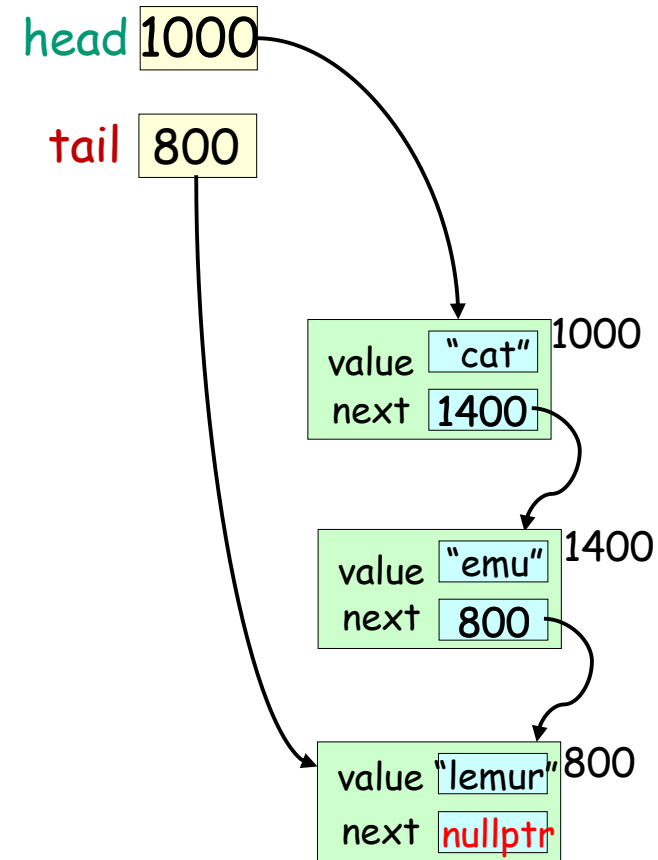
Since we have a **head pointer**...

Why not maintain a **"tail" pointer** too?

A **tail pointer** is a pointer that always **points to the last node** of the list!

```
class LinkedList
{
public:
    LinkedList() {...}
    void addToFront(string v) {...}
    ...

private:
    Node *head;
    Node *tail;
};
```

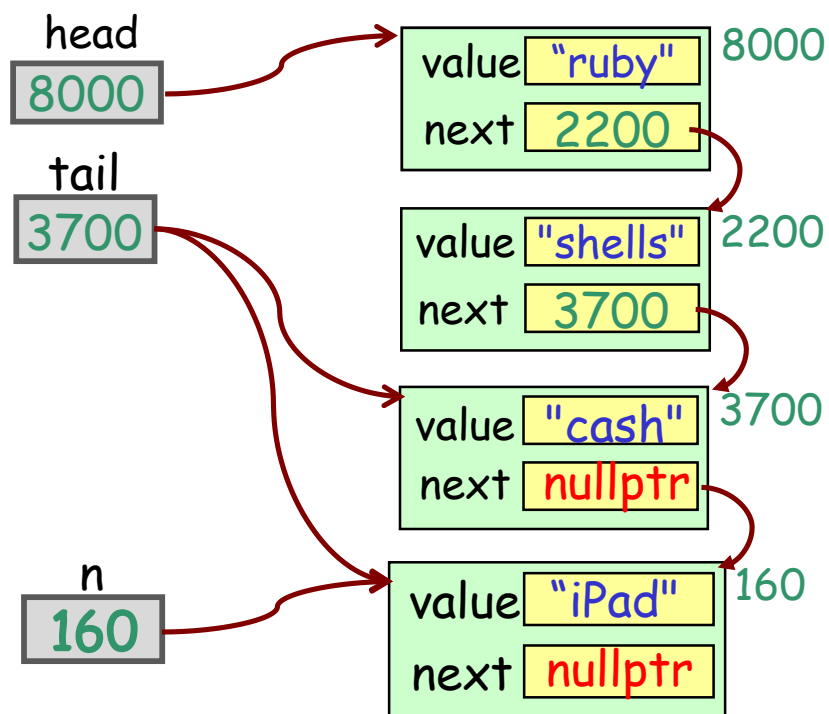


Using the **tail pointer**, we can **add new items** to the end of our list **without traversing**!

Adding an Item to the Rear... With a Tail Pointer

Let's see how to update our `addToRear()` function once our class has a `tail pointer`.

WARNING: You have to update all of your other methods to use the `tail pointer` (e.g., `constructor`, `addToFront()`) as well!



```
class LinkedList
{
public:
    void addToRear(string v)
    {
        if (head == nullptr)
            addToFront(v);
        else
        {
            Node *n = new Node;
            n->value = v;
            tail->next = n;
            n->next = nullptr;
            tail = n;
        }
    }

private:
    Node *head;
    Node *tail;
};
```

Doubly-linked Lists

One of the downsides with our simple linked list is that we can **only travel in one direction... down!**

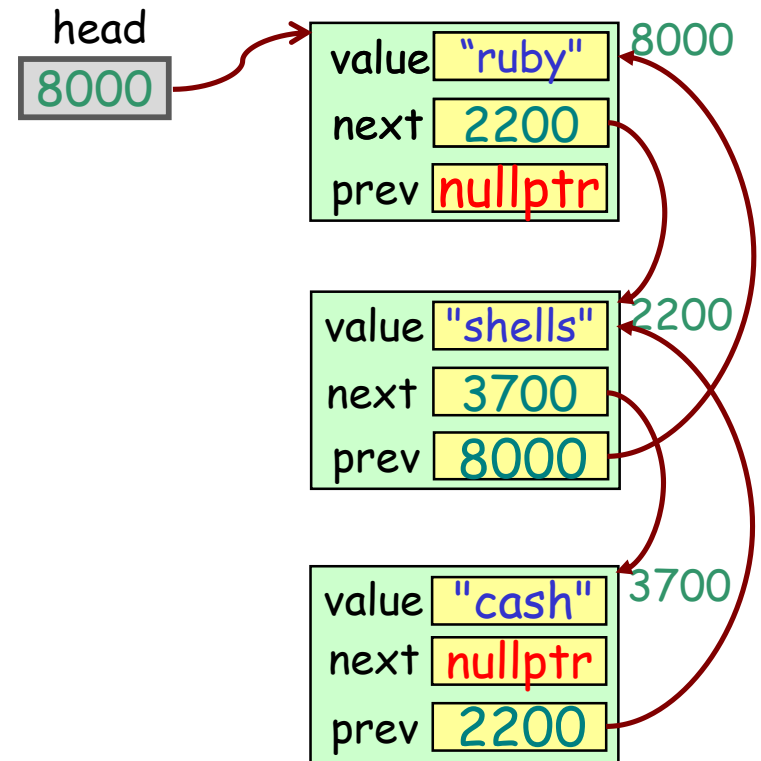
Given a pointer to a node, I can only find nodes below it!

Wouldn't it be nice if we could **move both directions** in a linked list?

We can! With a **doubly-linked list**!

A doubly-linked list has both *next* and *previous* pointers in every node:

```
struct Node
{
    string  value;
    Node *  next;
    Node *  prev;
};
```



Doubly-linked Lists

And, if I like, I can have a **tail pointer** too!

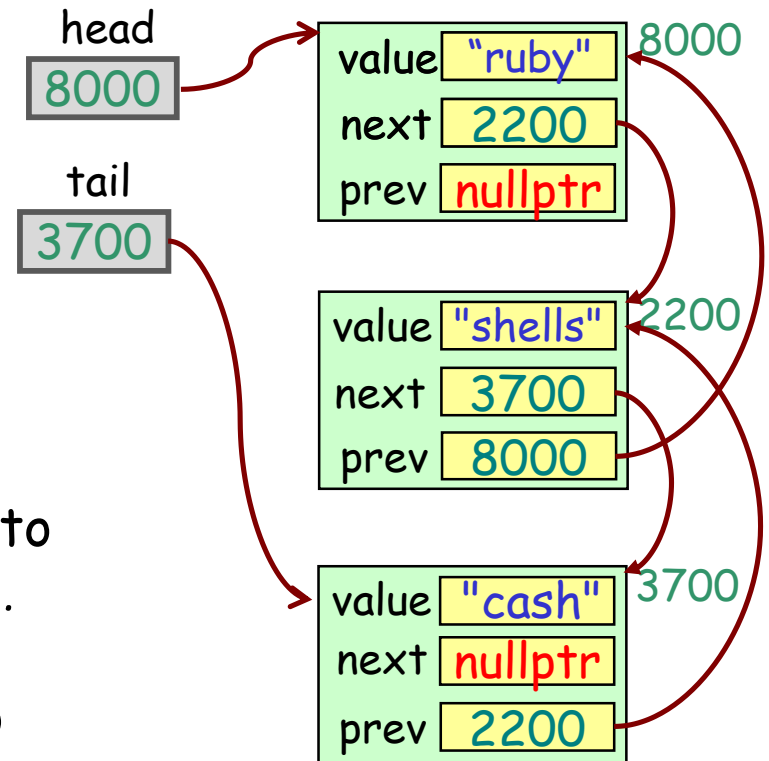
Now I can **traverse** in both directions!

```
Node *p;
```

```
p = head;
while (p != nullptr)
{
    cout << p->value;
    p = p->next;
}
```

```
Node *p;
```

```
p = tail;
while (p != nullptr)
{
    cout << p->value;
    p = p->prev;
}
```



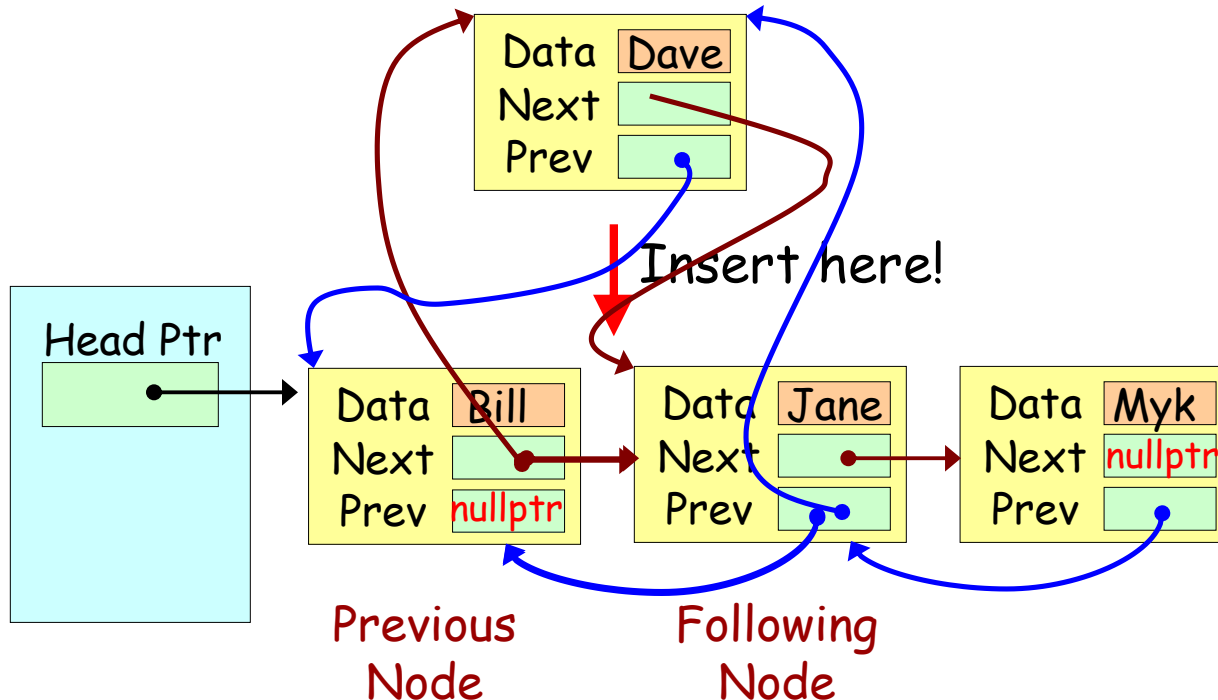
Of course, now we're going to have to **link up lots of additional pointers...**

But nothing comes free in life! 😊

Doubly-linked Lists: What Changes?

Every time we **insert a new node** or **delete an existing node**, we must update **three** sets of pointers:

1. The new node's next and previous pointers.
2. The previous node's next pointer.
3. The following node's previous pointer.



And of course, we still have **special cases** if we insert or delete nodes at the top or the bottom of the list.

Linked List Cheat Sheet

Given a pointer to a node: `Node *ptr;`

NEVER access a node's data until **validating its pointer**:

```
if (ptr != nullptr)
    cout << ptr->value;
```

To advance ptr to the next node/end of the list:

```
if (ptr != nullptr)
    ptr = ptr->next;
```

To see if ptr points to the last node in a list:

```
if (ptr != nullptr && ptr->next == nullptr)
    then-ptr-points-to-last-node;
```

To get to the next node's data:

```
if (ptr != nullptr && ptr->next != nullptr)
    cout << ptr->next->value;
```

To get the head node's data:

```
if (head != nullptr)
    cout << head->value;
```

To check if a list is empty:

```
if (head == nullptr)
    cout << "List is empty";
```

```
struct Node
{
    string value;

    Node *next;
    Node *prev;
};
```

Does our traversal meet this requirement?

```
NODE *ptr = head;
while (ptr != nullptr)
{
    cout << ptr->value;
    ptr = ptr->next;
}
```

To check if a pointer points to the first node in a list:

```
if (ptr == head)
    cout << "ptr is first node";
```

Linked Lists vs. Arrays

Which is Faster?

Getting to the 753rd item in a linked list or an array?

Which is Faster?

Inserting a new item at the front of a linked list or at the front of an array?

Which is faster?

Removing an item from the middle of a linked list or the middle of an array?

Which is easier to program?

Which data structure will take less time to program and debug?

Class Challenge

Write a function called **insert** that accepts two **NODE pointers** as arguments:

b4node: points to a node in a doubly-linked list

newnode: points to a new node you want to insert

When your function is called, it should insert **newnode** after **b4node** in the list, properly linking all nodes.

(You may assume that a valid node follows b4node prior to insertion.)

```
struct NODE
{
    string data;
    NODE *next, *prev;
};
```

