

Pointers and Memory Allocations

Pointers and `malloc()`

- ~~To my horror,~~ I discovered that memory allocation was not taught in CS1010, at least not in depth
 - In which, it was supposed to be taught there
- No matter what, you need to know and use that in this module.
 - Moreover, you need to *master* it in this course
- So in last lecture, some may not know what's going on

Last Lecture: Pointers

- You can think of a C/C++ pointer is a Pokeball
- And the object it's pointer to is the Pokemon

- You have to **create** a Pokemon to be put into the Pokeball by “new”

```
new int;
```



- At the same time, you “capture” (associate) your Pokemon(object) by the Pokeball (pointer)

```
int *ptr = new int;
```



Computer Memory

- For storage of data

```
int a = 3;
```

a

This space is allocated and labeled as “a”. And we store ‘3’ inside

Computer Memory



Computer

Address	Content
ffbff7d8	
ffbff7d9	
ffbff7da	
ffbff7db	
ffbff7dc	3
ffbff7dd	
ffbff7de	
ffbff7df	
ffbff7e0	3.14
ffbff7e1	
ffbff7e2	

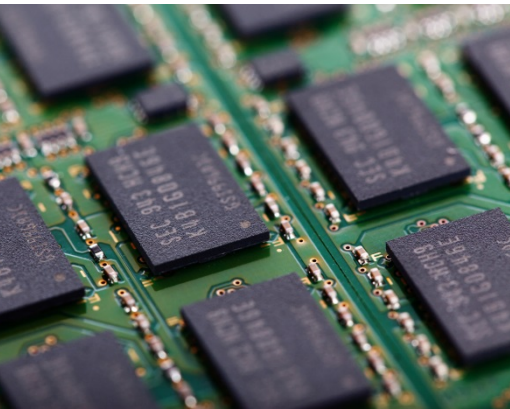
The
address of
the
variable **a**



The
content is
an integer



Another variable:
float b = 3.14;



Computer

Address	Content
---------	---------

When you have:

```
int a = 3;
```

In your code, the green space will be “chopped” for you **automatically**

The
address of
the
variable **a**

ffbff7db

ffbff7dc

ffbff7dd

ffbff7de

ffbff7df

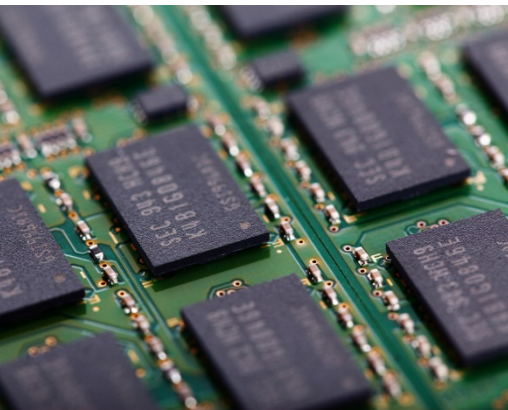
ffbff7e0

ffbff7e1

ffbff7e2

3

The
content is
an integer



Last Lecture: Pointers

- You can think of a C/C++ pointer is a Pokeball
- And the object it's pointer to is the Pokemon
- Calling a Pokemon from a Pokeball by “*”
- If you have a Pokemon, you want to FIND it's Pokeball by “&”

```
int x;
```

```
int *ptr;
```

```
ptr = &x;
```



Computer

Address	Content
ffbff7d8	
ffbff7d9	
ffbff7da	
ffbff7db	
ffbff7dc	
ffbff7dd	
ffbff7de	
ffbff7df	
ffbff7e0	
ffbff7e1	
ffbff7e2	

So **&a** will be the address **ffbff7dc**

The
address of
the
variable **a**

ffbff7db

ffbff7dc

ffbff7dd

ffbff7de

ffbff7df

ffbff7e0

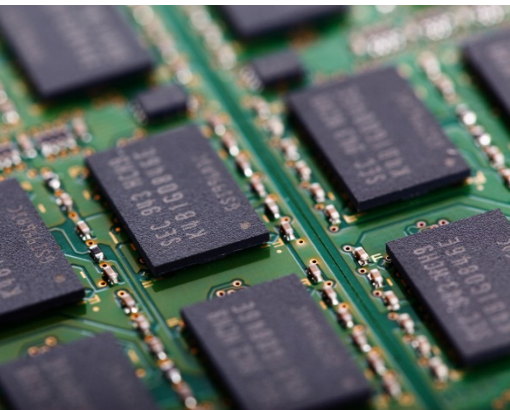
ffbff7e1

ffbff7e2

3

The
content is
an integer

What if I want to
create a variable
to store this?



Storing the Address of a

```
int a = 3;
```

```
int *a_ptr = &a;
```

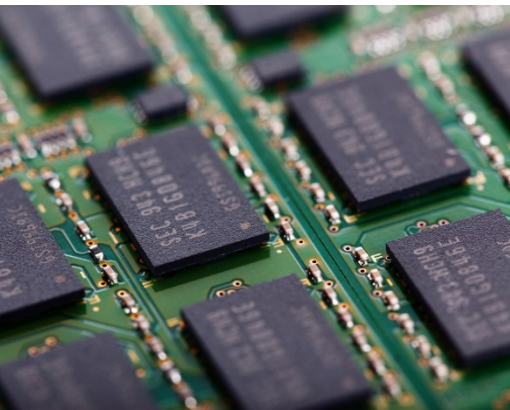
How

Address	Content
ffbff7d8	ffbff7dc
ffbff7d9	
ffbff7da	
ffbff7db	
ffbff7dc	3
ffbff7dd	
ffbff7de	
ffbff7df	
ffbff7e0	
ffbff7e1	
ffbff7e2	

The address of the variable **a_ptr**

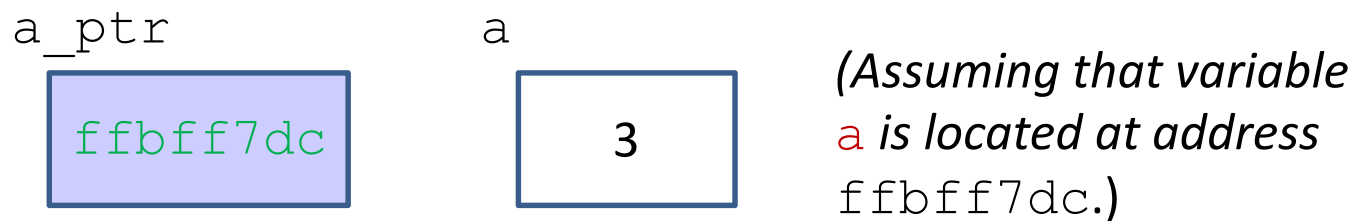
The address of the variable **a**

The content of the address is an address

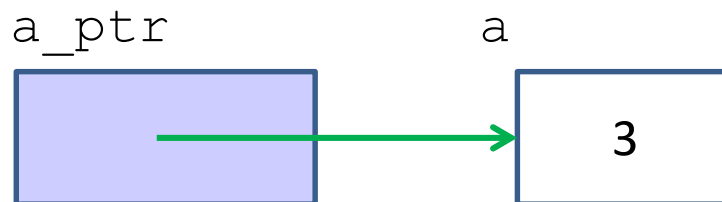


Pointer Variable

- Example: a pointer variable `a_ptr` is shown as the left box below. It contains the address of variable `a`.



- Variable `a_ptr` is said to be **pointing to** variable `a`.
- Usually, we will simply **draw an arrow** to indicate this



How

Address	Content
ffbff7d8	ffbff7dc
ffbff7d9	
ffbff7da	
ffbff7db	
ffbff7dc	3
ffbff7dd	
ffbff7de	
ffbff7df	
ffbff7e0	
ffbff7e1	
ffbff7e2	

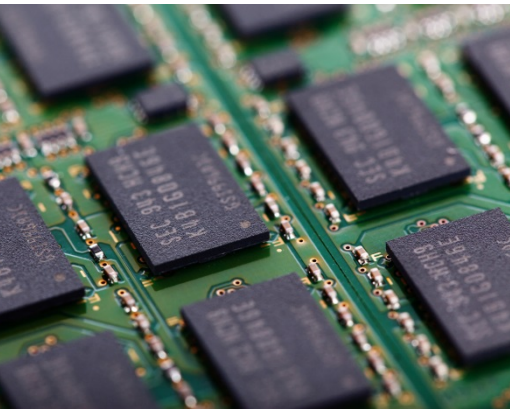
The address of the variable **a_ptr**

The address of the variable **a**

The content of the address is an address

ffbff7dc

3



Example #1

```
int i = 10, j = 20;  
int (*p) // p is a pointer to some int variable
```

```
p = (&i); // p now stores the address of variable i
```

Important!

Now `*p` is equivalent to `i`

```
printf("value of i is %d\n", *p);
```

value of i is 10

```
// *p accesses the value of pointed/referred variable
```

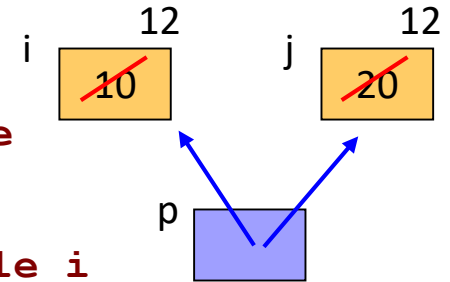
```
*p = *p + 2; // increment *p (which is i) by 2  
// same effect as: i = i + 2;
```

```
p = &j; // p now stores the address of variable j
```

Important!

Now `*p` is equivalent to `j`

```
*p = i; // value of *p (which is j now) becomes 12  
// same effect as: j = i;
```



What is “new”?

How

Address	Content
---------	---------

When you use “new”, you “chop” a place in the memory by yourself, instead of automatically, e.g

```
new int;
```

ffbff7db	
ffbff7dc	
ffbff7dd	
ffbff7de	
ffbff7df	
ffbff7e0	
ffbff7e2	



However, who knows your address?

Analogy

- Maybe 5 or 10 years later, you got married in Singapore, and you applied for an HDB
- And yes, govt gave you one. Saying, we got one HDB for you.
- Ermm... where are you going to find your own HDB flat?
 - By...?

How

Address	Content
---------	---------

You need to remember your address!!!!!!

ffbf7d9	
---------	--

ffbff7da	
----------	--

ffbff7db	
----------	--

ffbff7dc	
----------	--

ffbff7dd	
----------	--

ffbff7de	
----------	--

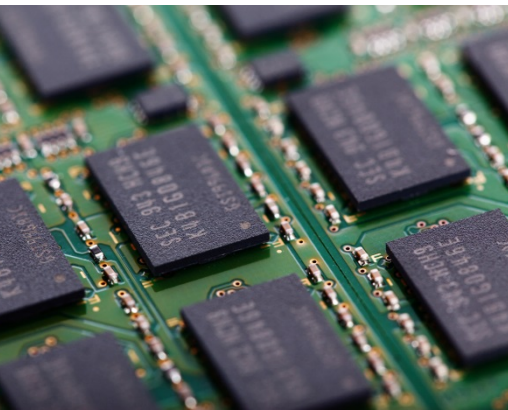
ffbff7df	
----------	--

ffbff7e0	
----------	--

ffbff7e1	
----------	--

ffbff7e2	
----------	--

?


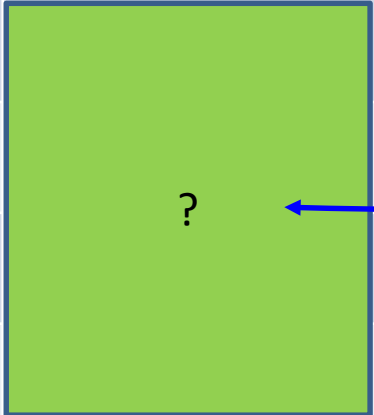


Pointer Variable

- That's why, you want to store the address in a pointer variable

```
int *ptr = new int;
```

How

Address	Content
ffbff7d8	
ffbff7d9	
ffbff7da	
ffbff7db	
ffbff7dc	
ffbff7dd	
ffbff7de	
ffbff7df	
ffbff7e0	

The address of
the variable
ptr

ffbff7dc

?

```
int *ptr = new int;
```

ffbff7e2



7. Common Mistake

Unit7_Common_Mistake.c

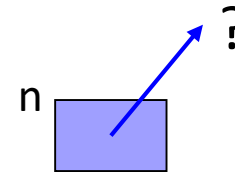
```
#include <stdio.h>

int main(void) {
    int *n;

    *n = 123;
    printf("%d\n", *n);

    return 0;
}
```

What's wrong with this?
Can you draw the picture?



- Where is the pointer `n` pointing to?
- Where is the value `123` assigned to?
- Result: Segmentation Fault (core dumped)
 - Remove the file “core” from your directory. It takes up a lot of space!

© Randy Glasbergen
glasbergen.com



**“Your shipment was delivered to the wrong
address, so technically, it’s your fault
for choosing not to live there!”**

Valid if a is allocated

The address of
the variable
a_ptr

The address of
the variable **a**

Address	Content
ffbff7d8	ffbff7dc
ffbff7d9	
ffbff7da	
ffbff7db	
ffbff7dc	123
ffbff7dd	
ffbff7de	
ffbff7df	
ffbff7e0	
ffbff7e1	

ffbff7dc

123

What if...?

The address of
the variable
a_ptr

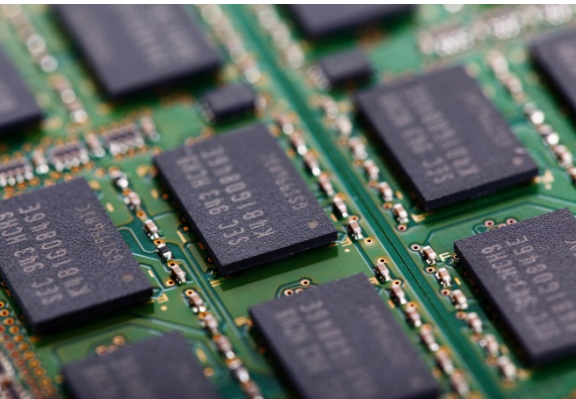
The address of
the variable **a**

Address	Content
ffbff7d8	aabbccdd
ffbff7d9	
ffbff7da	
ffbff7db	
ffbff7dc	123
ffbff7dd	
ffbff7de	
ffbff7df	
ffbff7e0	
ffbff7e1	

aabbccdd

123

?



Computer Memory

- For storage of data

```
int a = 3;
```

a

This space is allocated and labeled as “a”. And we store ‘3’ inside

Computer Memory



Computer Memory

- For storage of data

```
int a = 3;
```

```
int *a_ptr = &a;    a_ptr
```

Computer
Memory



Computer Memory

- For storage of data

```
int a = 3;
```

```
int *a_ptr; //can be rubbish
```

a_ptr

Computer
Memory



Invalid Area (Address)

- For storage of data

```
int a = 3;
```

```
int *a_ptr; //can be rubbish
```

Computer
Memory

a_ptr

Segmentation
Fault!!!!!!!

Win
Operatin
System

3

aabbccdd

Your program

Other
app

Other
app

Are you ready for



Invalid Area (Address)

- For storage of data

```
int a = 3;
```

```
int *a_ptr; //can be rubbish
```

Computer
Memory

a_ptr

Modify other
programs (or your
own)

Win
Operatin
System

3

aabbccdd

Your program

Other
app

Other
app

Are you ready for

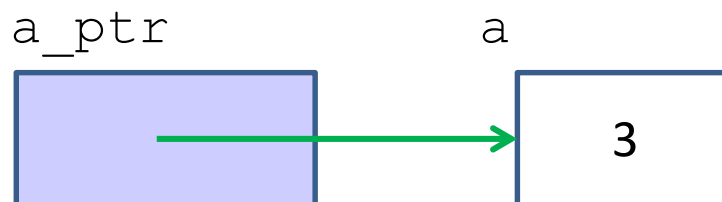


At now, you need to understand this

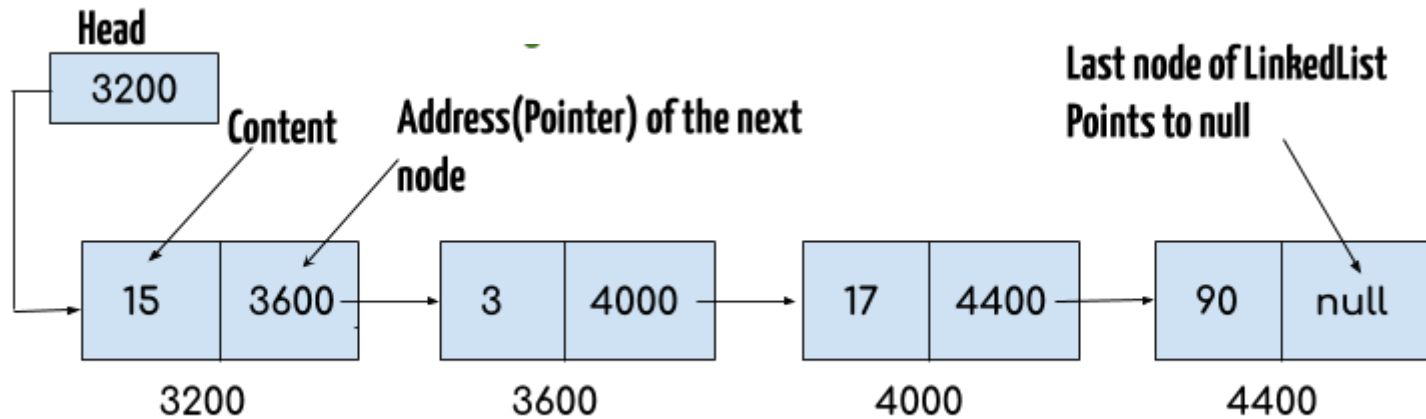
- Example: a pointer variable `a_ptr` is shown as the left box below. It contains the address of variable `a`.



- Variable `a_ptr` is said to be **pointing to** variable `a`.
- Usually, we will simply **draw an arrow** to indicate this



Linked Lists



Variable-size Storage

Return to CS2040C

Variable-sized Storage Problem

- Let's write a program to record all the marks of students in a course
 - So that, we can compute a lot of useful data, e.g. average mark
- However, how do we store all the marks
 - Make it easy, let's just store one single integer mark for each students

Let's say we input the marks by hand

- Somewhere in your code will be *looping* this:
 - For each of the student:
 - From keyboard, input the mark of one student
 - Store *this* mark to.....?
 - a) A Single variable
 - b) An array
 - c) Or....?

Natural to Use Arrays, but

- When we declare an array, we need to know the size of the array, e.g. for 100 students

```
int student_mark[100];
```

- But every course has a different number of students?!
 - 10? 100? 1000?

Can we.....?

- Declare a large array, e.g.

```
int student_mark[10000];
```

- Problems?
- How then?

Linked List in C++

Why Linked List?

- Why not just store everything in an array?
- Variable/unknown size of data
 - Even the size will change during the running of the program
- No need to store all the data in ONE connected trunk of memory

Linked List is like a Train



Each Carriage

Contents

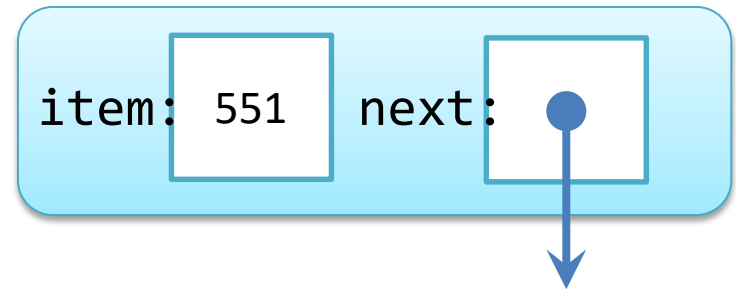
“Connector”
to the other
train



Linked List in C++

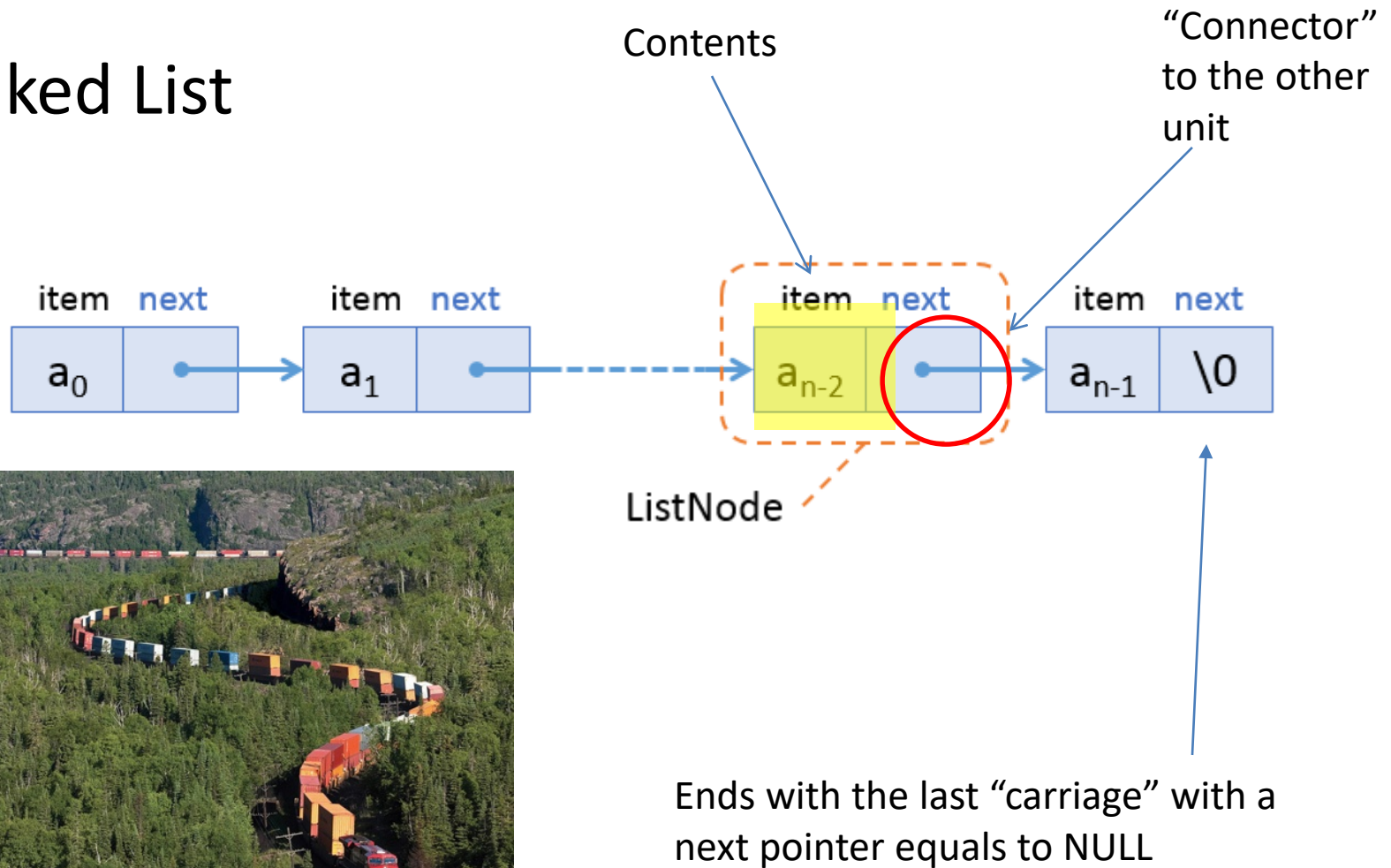
```
class ListNode {  
private:  
    int item;  
    ListNode *next;  
  
    More to come  
}
```

- In graphical form:

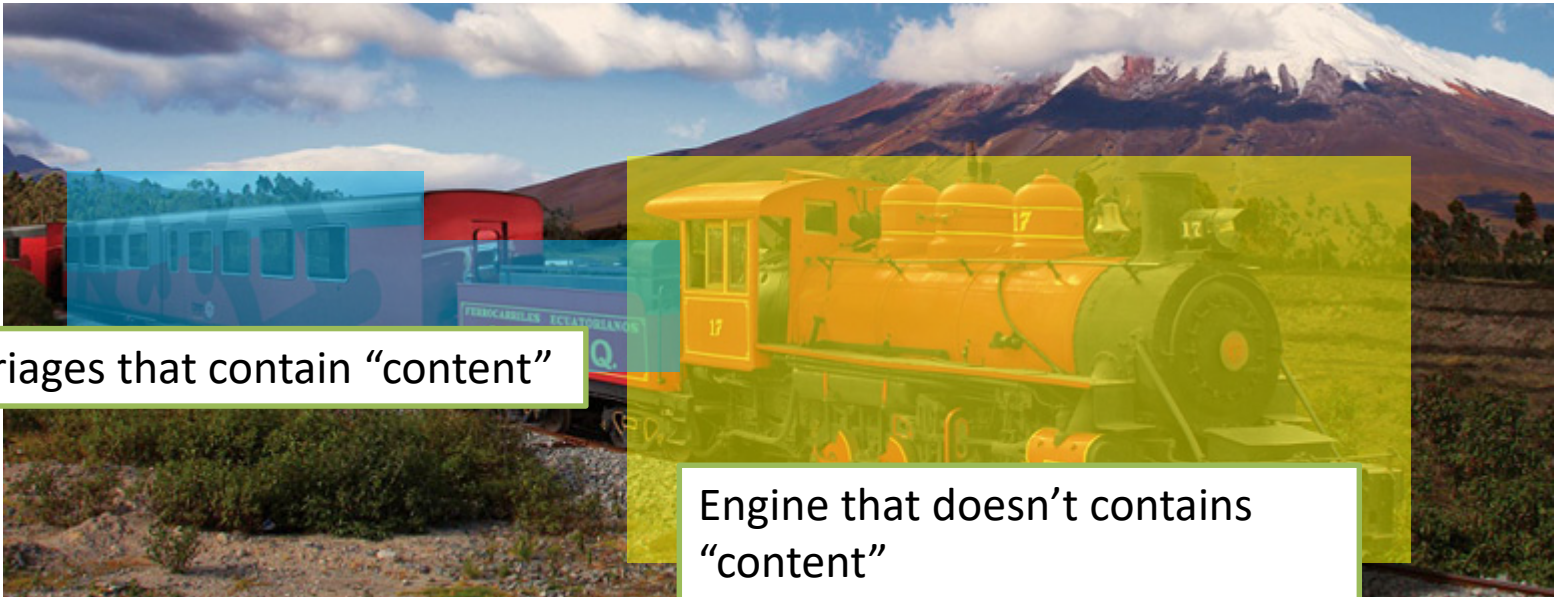


A Linked List with a lot of nodes

- Linked List



The Engine and the Carriages

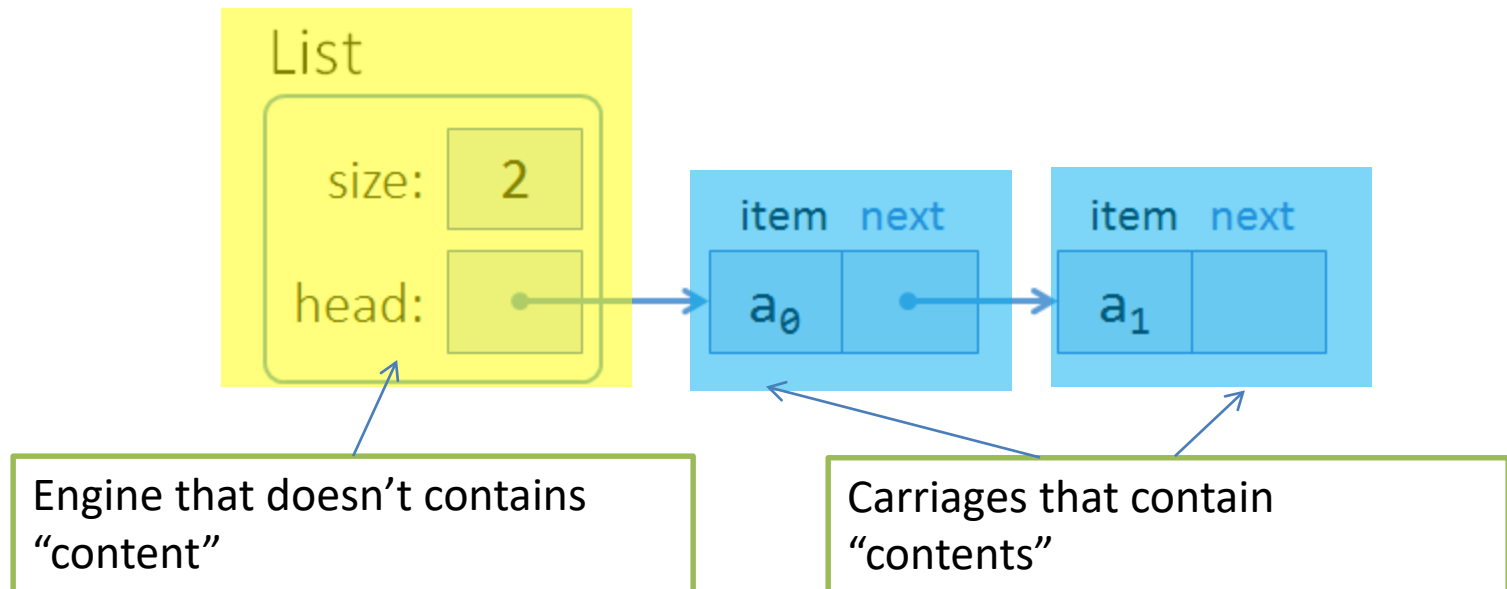


Carriages that contain “content”

Engine that doesn't contains
“content”

The “Engine”

```
class List {  
private:  
    int size;  
    ListNode *head;  
};
```



Linked List in C++

```
class ListNode {  
private:  
    int item;  
    ListNode *next;
```

More to come

}

```
class List {  
private:  
    int size;  
    ListNode *head;
```

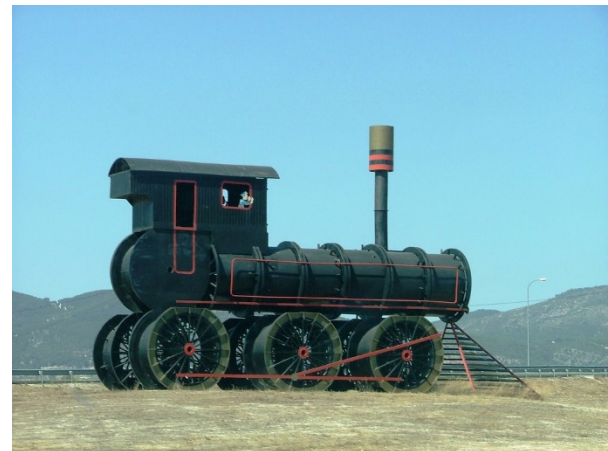
More to come

}



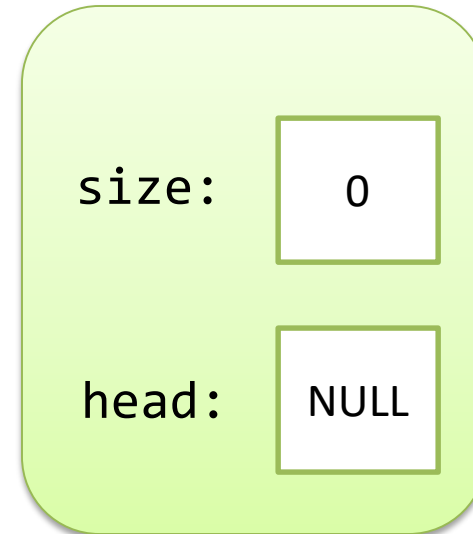
In the Beginning

- When we create a list
 - There will be no content
 - Namely, no list node



How do we initialize the two variables?

```
class List {  
private:  
    int size;  
    ListNode *head;
```

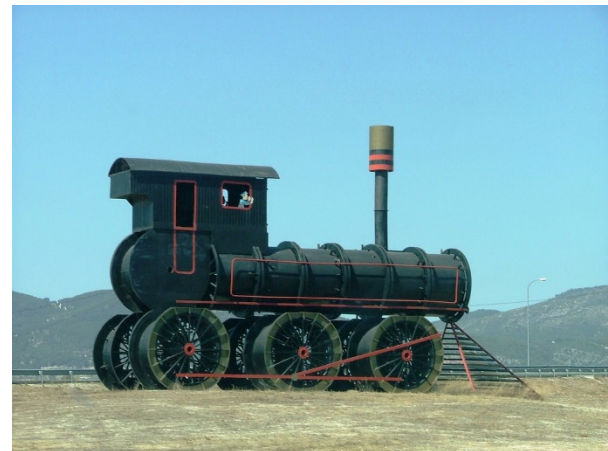
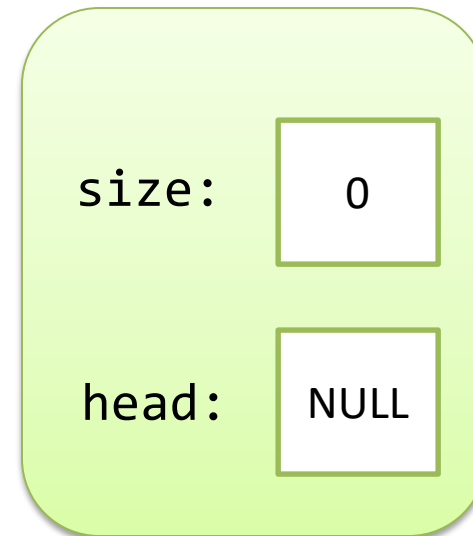


```
}
```



OOP: Constructors

```
class List {  
private:  
    int size;  
    ListNode *head;  
public:  
    List();  
  
}
```



What is a Constructor?

- A Constructor is called when your object is just created **automatically**
- For example, when you declare it

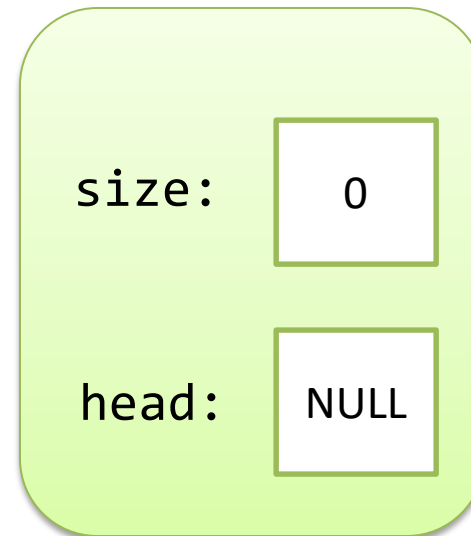
```
main() {  
    List ll; ← constructor is called here  
    ... some statements ..  
}
```

- Or when you “new” it

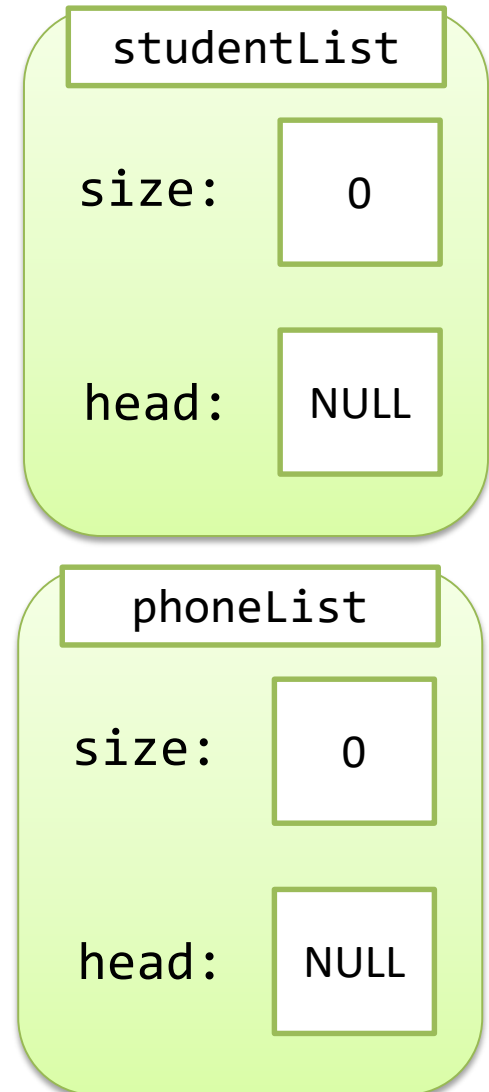
```
main() {  
    List *lp; ← constructor is NOT called here  
    .... some statements ...  
    lp = new List; ← constructor is called here  
}
```


Constructor

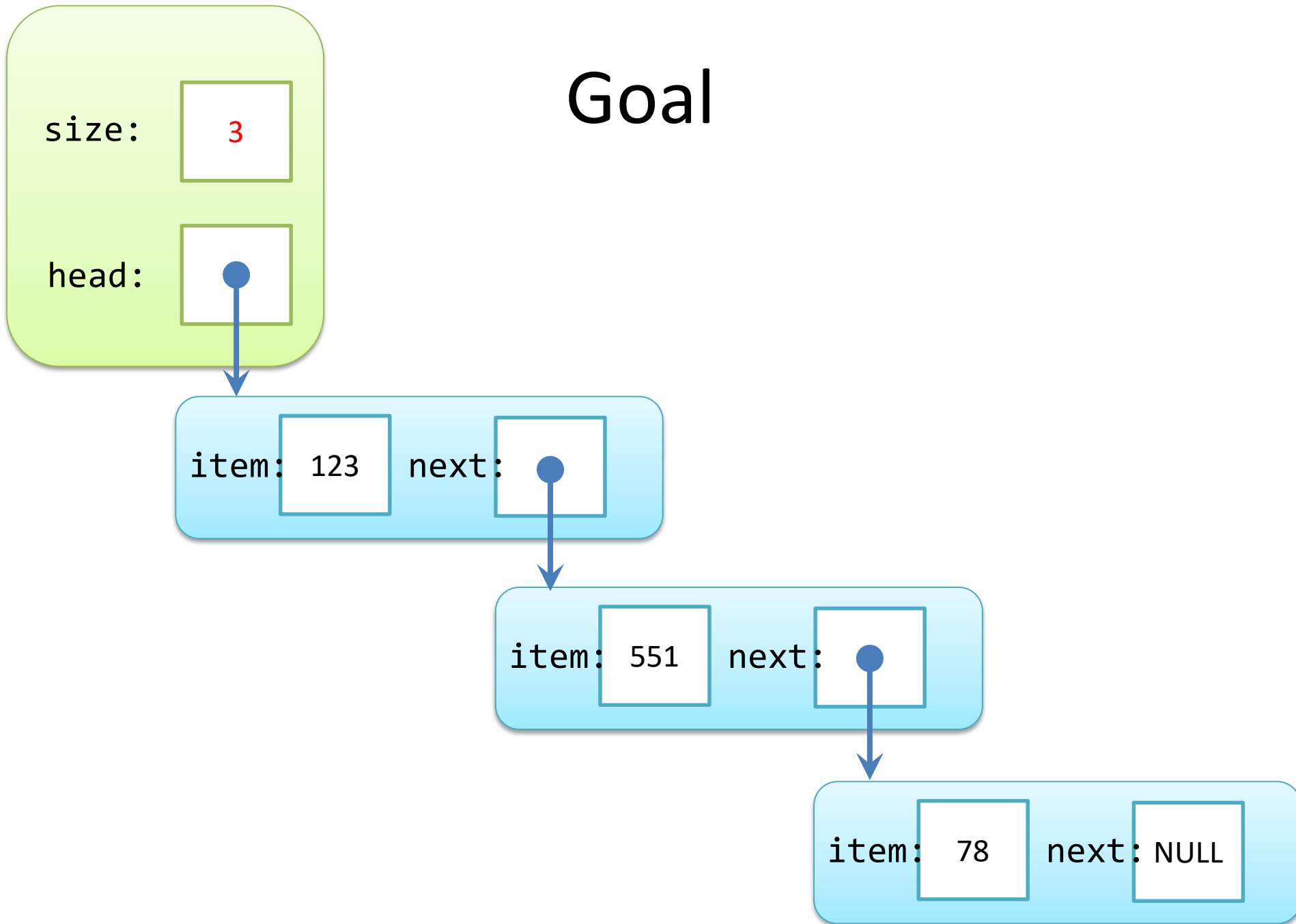
```
List::List()  
{  
    size = 0;  
    head = NULL;  
}
```



```
int main() {  
    List studentList;  
    List phoneList;  
  
    Two  
    instances of  
    ONE class  
  
}
```



Goal

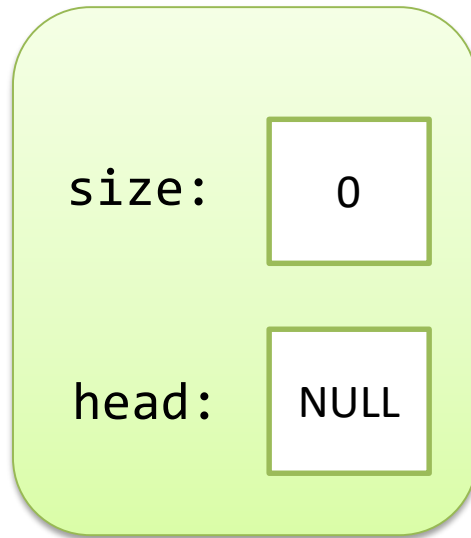


Add/insert Nodes to the List

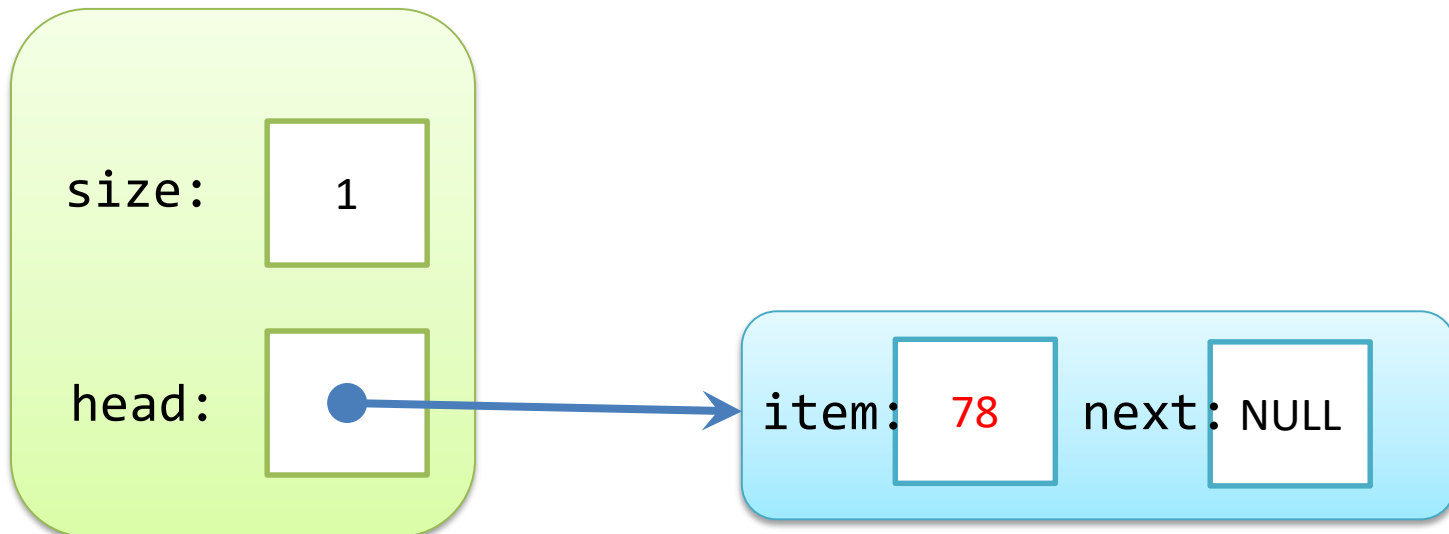
```
class List {  
private:  
    int size;  
    ListNode *head;  
public:  
    List();  
    insert(int);  
  
};
```

Insert(): Goal

- From:



- To:



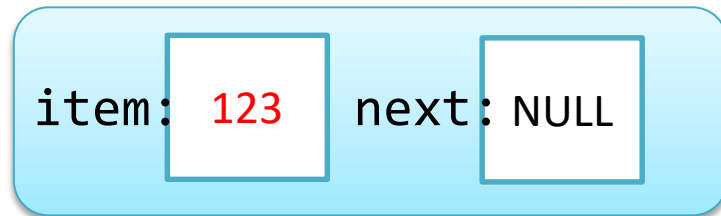
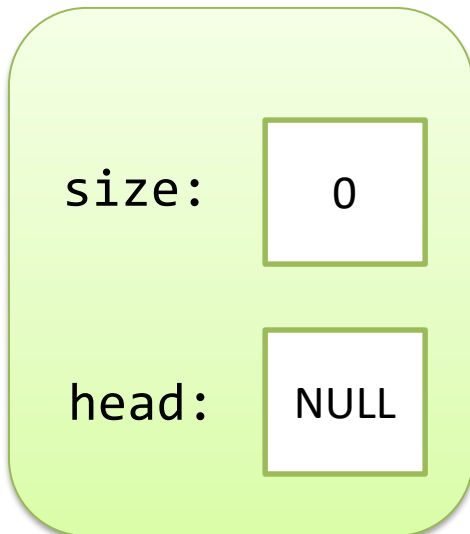
Coupling in Trains



<https://www.youtube.com/watch?v=oYkugel2zFI>

Step 1

- Create a ListNode



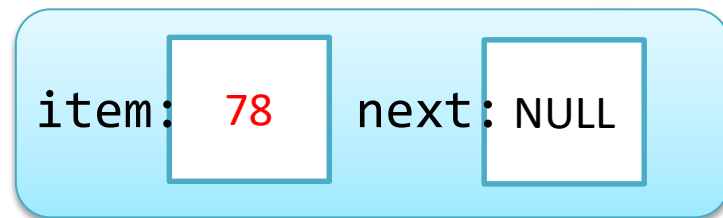
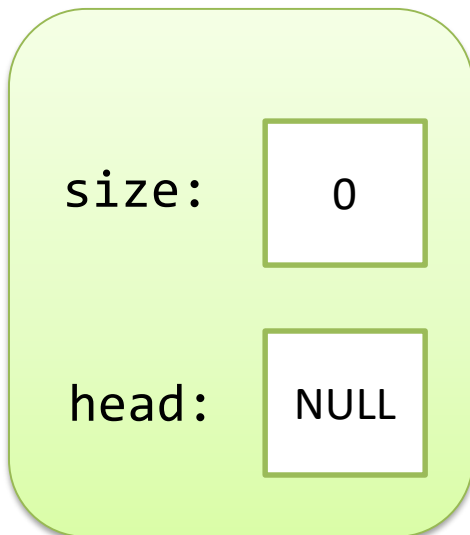
Constructor for ListNode

```
class ListNode {  
private:  
    int item;  
    ListNode *next;  
public:  
    ListNode(int);  
  
};
```

```
ListNode::ListNode(int n)  
{  
    item = n;  
    next = NULL;  
}
```


Step 1

- Create a ListNode by
`aNewNode = new ListNode(78)`
- This will call the constructor with 78 as a parameter



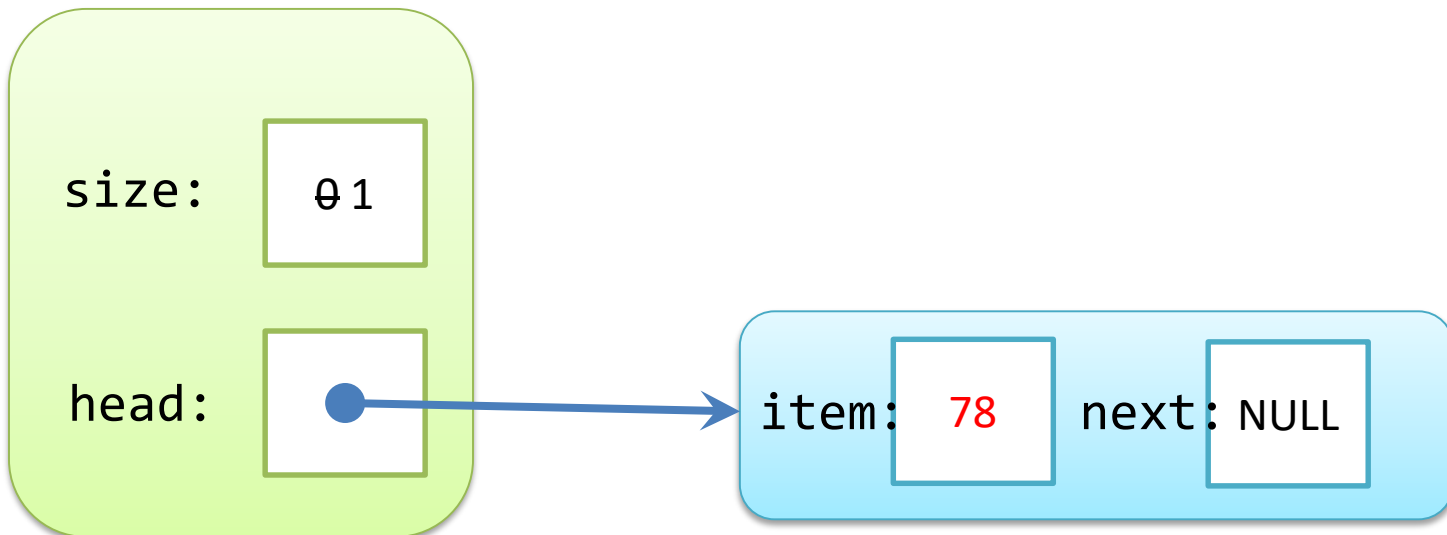
Step 2

- Connect the new node to “head”

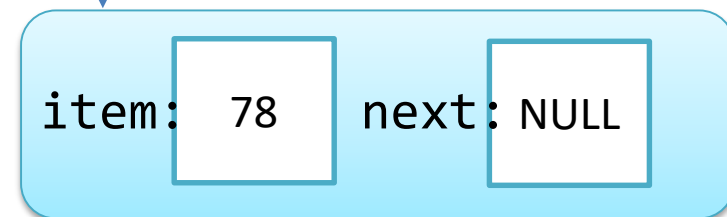
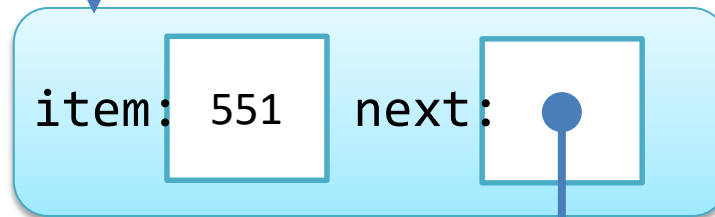
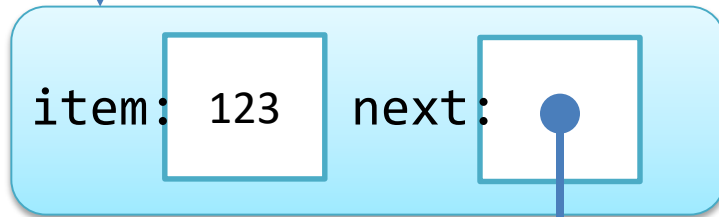
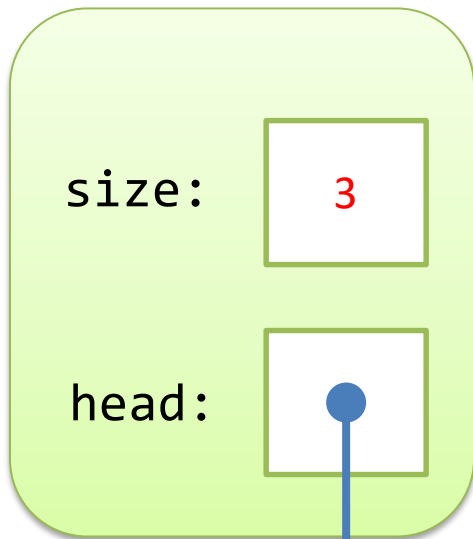
```
head = aNewNode;
```

- And increment size by 1

```
size++;
```

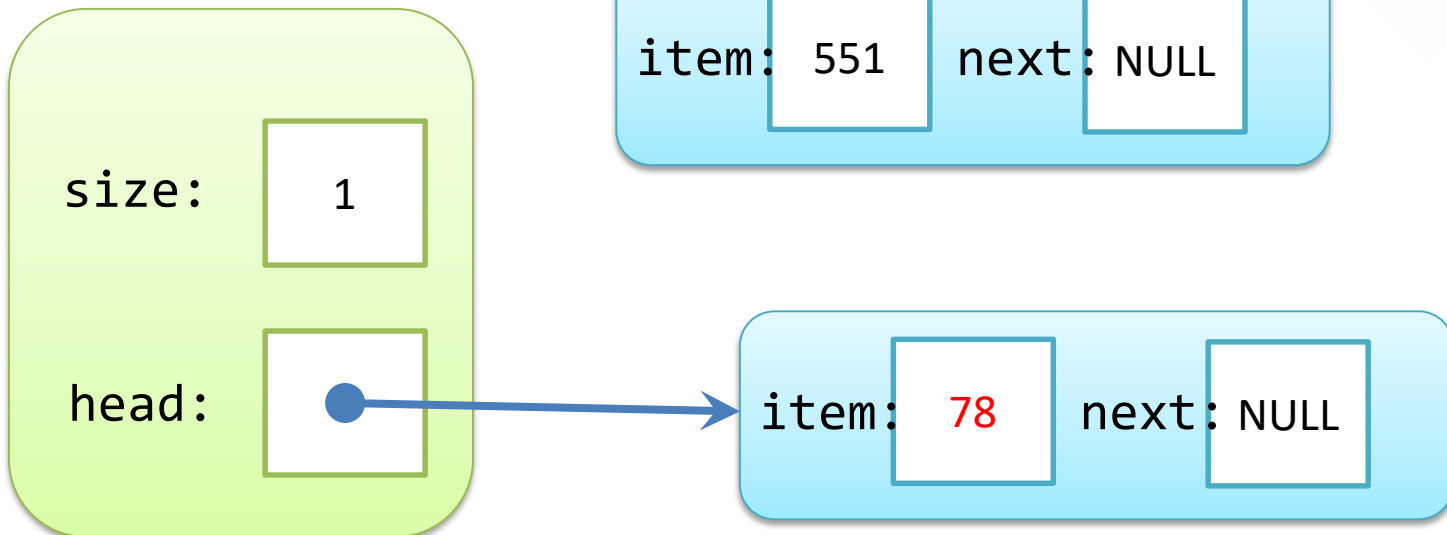


Add More Nodes?



Add More Nodes

- Same thing, create a node by “new”
- But how to connect them?



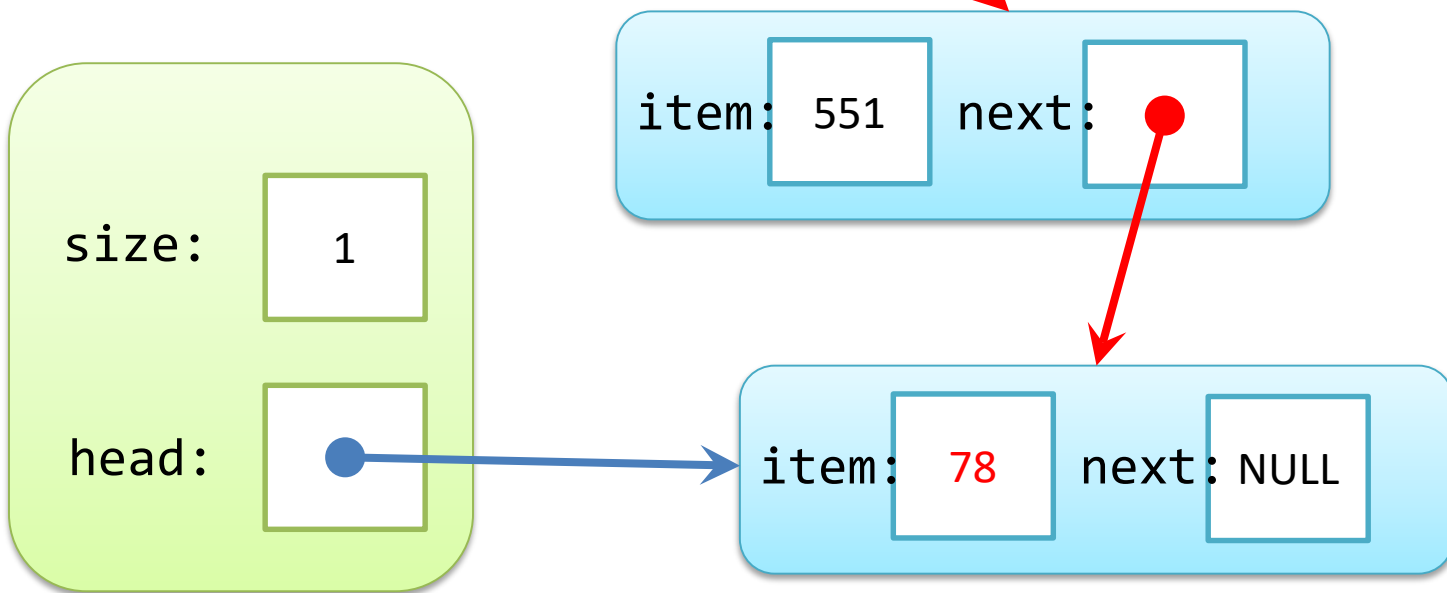
Inserting a Node

- Step 1: Create a new node by “new”

```
ListNode* aNewNode = new ListNode(551);
```

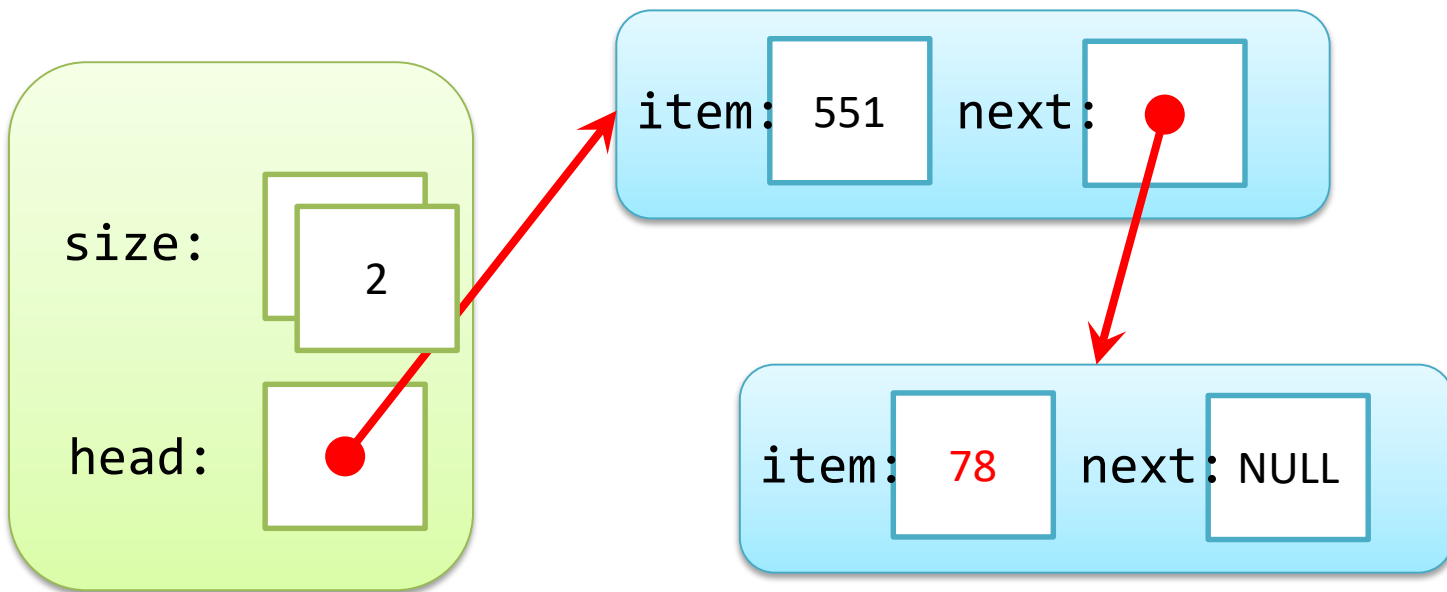
- Step 2: Point the new node TO the first node

```
aNewNode->next = head;
```



Inserting a Node

- Step 3: Point the head TO the new node
`head = aNewnode;`
- Step 4: Increase “size” by 1



Linked List in C++

```
class ListNode {  
private:  
    int item;  
    ListNode *next;
```

More to come

```
};
```

```
class List {  
private:  
    int size;  
    ListNode *head;
```

More to come

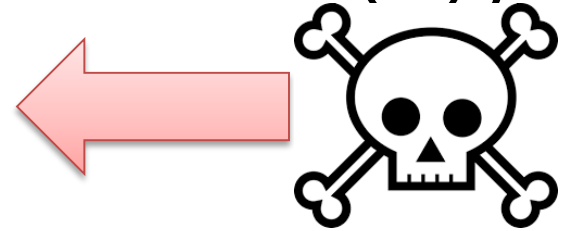
```
};
```

Linked List in C++

```
class List {  
private:  
    int size;  
    ListNode *head;  
public:  
    List();  
    void insert(int);  
  
};
```


Four Steps of Insertion

```
void List::insert(int n) {  
    ListNode *aNewNode = new ListNode(n);  
    aNewNode->next = head;  
    head = aNewNode;  
    size++;  
}
```



Wait a minute.....



“next” is a **Private** Member of ListNode!

```
void List::insert(int n) {  
    aNewNode = new ListNode(n);  
    aNewNode->next = head;  
    head = aNewNode;  
    size++;  
}
```



```
class ListNode {  
    private:  
        int item;  
        ListNode *next;  
    public:  
        ListNode(int);  
};
```

What Can We Do?

- Simply change `ListNode::next` to *public*
 - Then anyone can access it!
- Create another *public* function `ListNode::setNext()` to allow changing of the pointer

```
ListNode::setNext(ListNode *ptr) {  
    next = ptr;  
}
```

- Same as above!!

Maintain a Special Relationship

- Create a “**relationship**” such that class `List` can access “`next`” inside `ListNode`

```
void List::insert(int n) {  
    aNewNode = new ListNode(n);  
    aNewNode->next = head;  
    head = aNewNode;  
    size++;  
}
```



```
class ListNode {  
    private:  
        int item;  
        ListNode *next;  
    public:  
        ListNode(int);  
};
```

“friend” in C++

- Allow other class to access its private members or methods

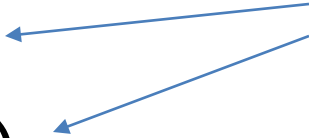
```
class ListNode {  
    private:  
        int item;  
        ListNode *next;  
    public:  
        ListNode(int);  
        friend class List;  
};
```



Other Insertions

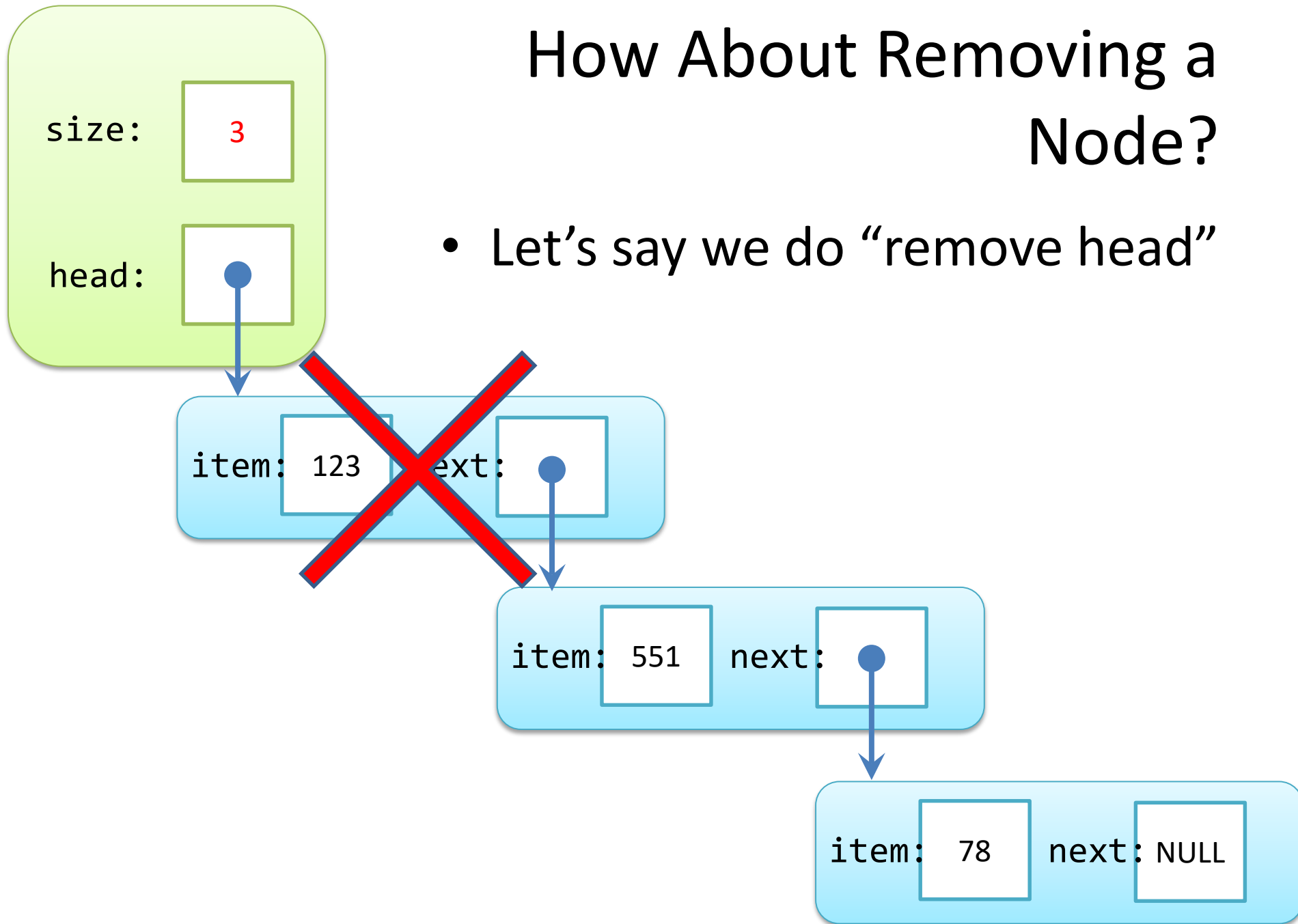
- The insertion that we have covered is actually:
 - `insertAtHead()`
- There are other varieties
 - `insertAtTail()`
 - `insertAtPos(int)`
 - `insertAtAfter(int)`

The position in
the list that the
new node is
going to be
added



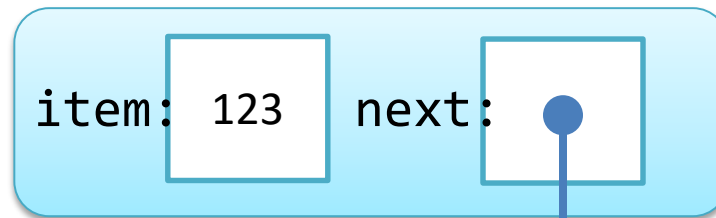
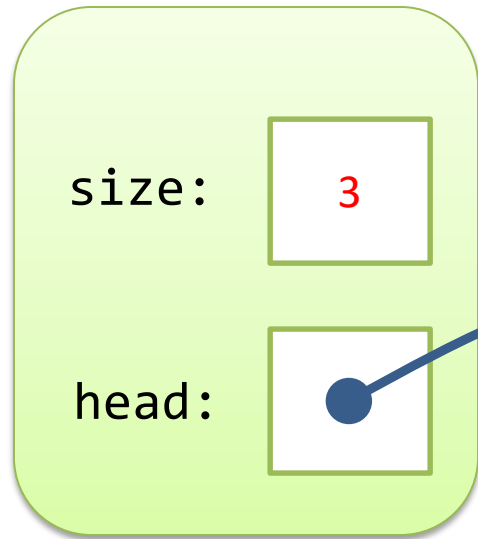
How About Removing a Node?

- Let's say we do "remove head"



Remove Head

- How about: `head = head->next;`



Remove Head

- How about: `head = head->next;`
- Problem?
 - What if the list is empty and we called “remove”?
 - Even if it’s not empty, what happens to the first node?

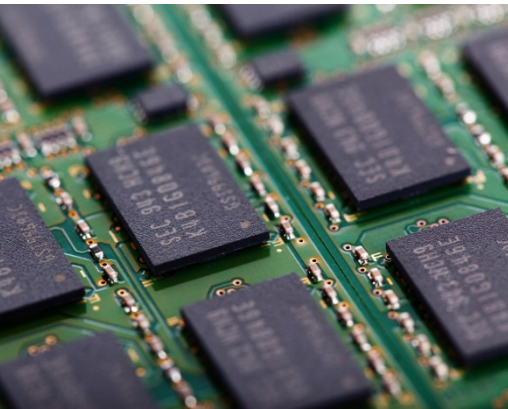


How

Address	Content
ffbff7db	
ffbff7dc	
ffbff7dd	
ffbff7de	
ffbff7df	
ffbff7e0	
ffbff7e1	
ffbff7e2	

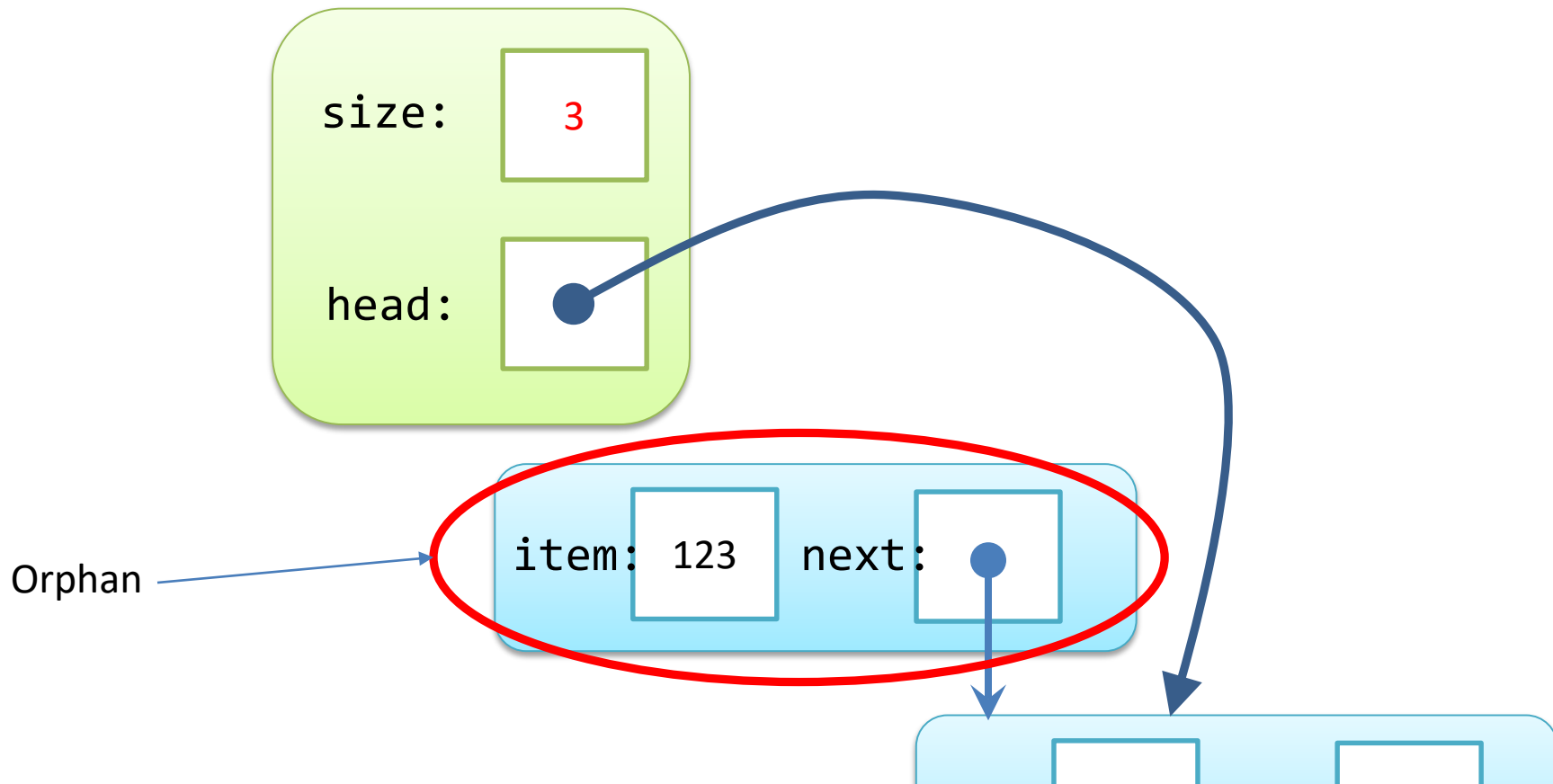
When you use “new”, you “chop” a place in the memory by yourself, instead of automatically, e.g

```
new int;
```



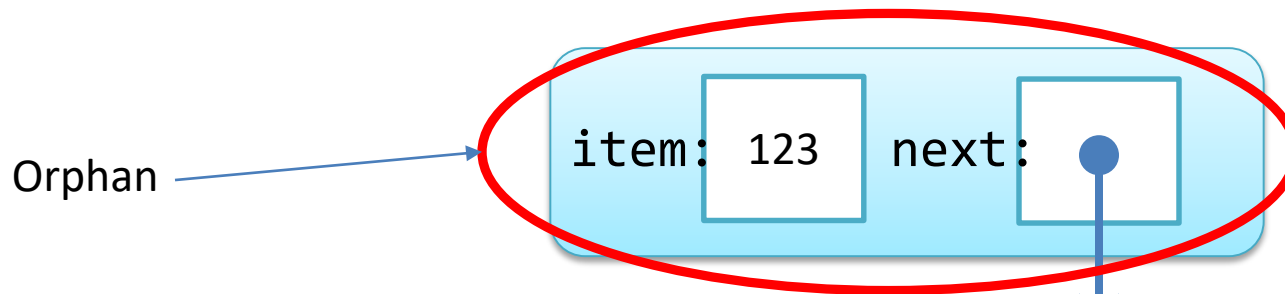
Orphan

- An object created by new, is only accessible when there is a pointer pointing to it



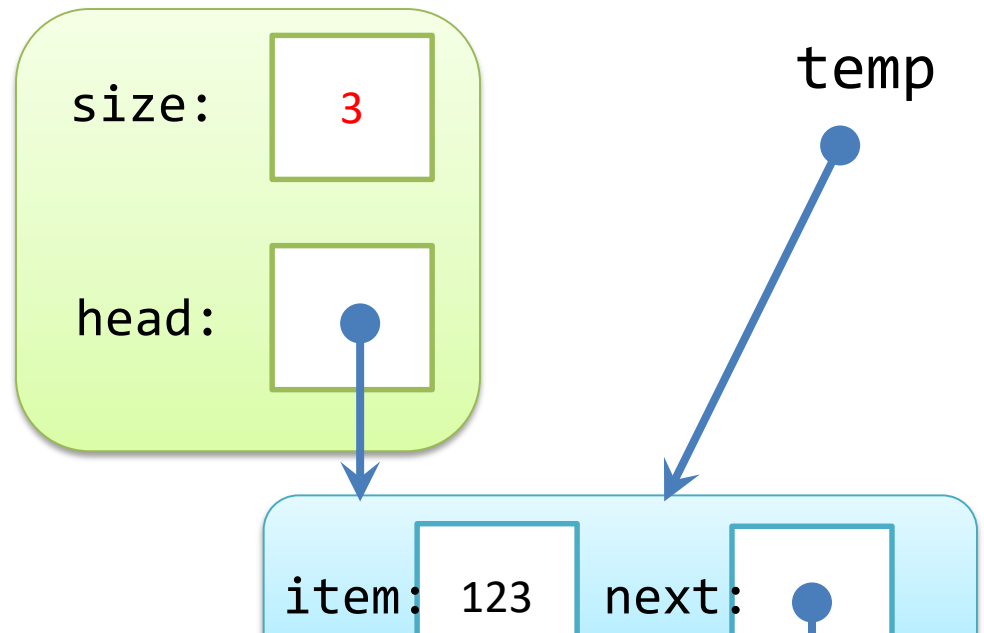
Orphan

- An object created by new, is only accessible when there is a pointer pointing to it
- And even it will not be used anymore, you should delete it
 - Otherwise: Memory Leak



Remove Head

- So, BEFORE: `head = head->next;`
- Create a temporary pointer pointing to the first node
 - `ListNode *temp = head;`



Remove Head Node

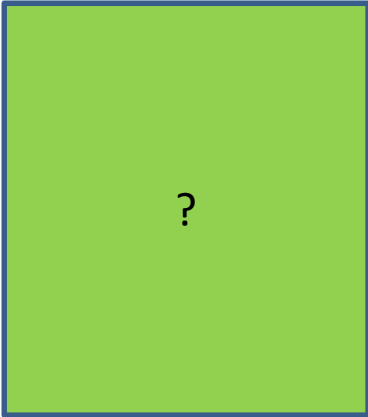
```
void List::removeHead() {  
    if(size > 0) {  
        ListNode *temp = head;  
        head = head->next;  
        delete temp;  
        size--;  
    } else  
        // what should we do?  
}
```

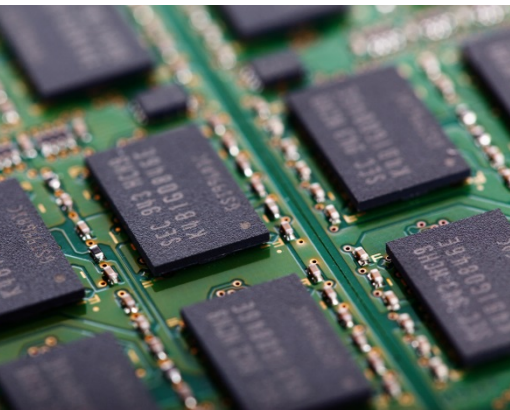
How

Address	Content
---------	---------

When you use “new”, you “chop” a place in the memory by yourself, instead of automatically, e.g

```
new int;
```

ffbff7db	
ffbff7dc	
ffbff7dd	
ffbff7de	
ffbff7df	
ffbff7e0	
ffbff7e1	
ffbff7e2	



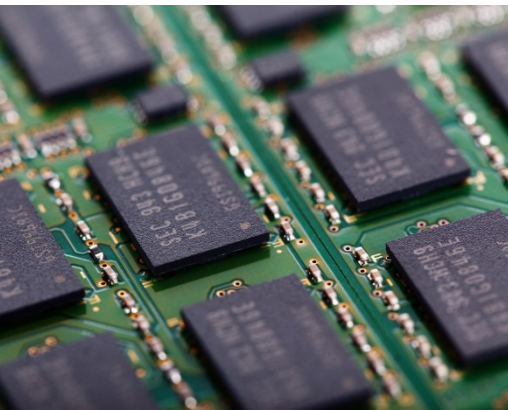
How

Address	Content
---------	---------

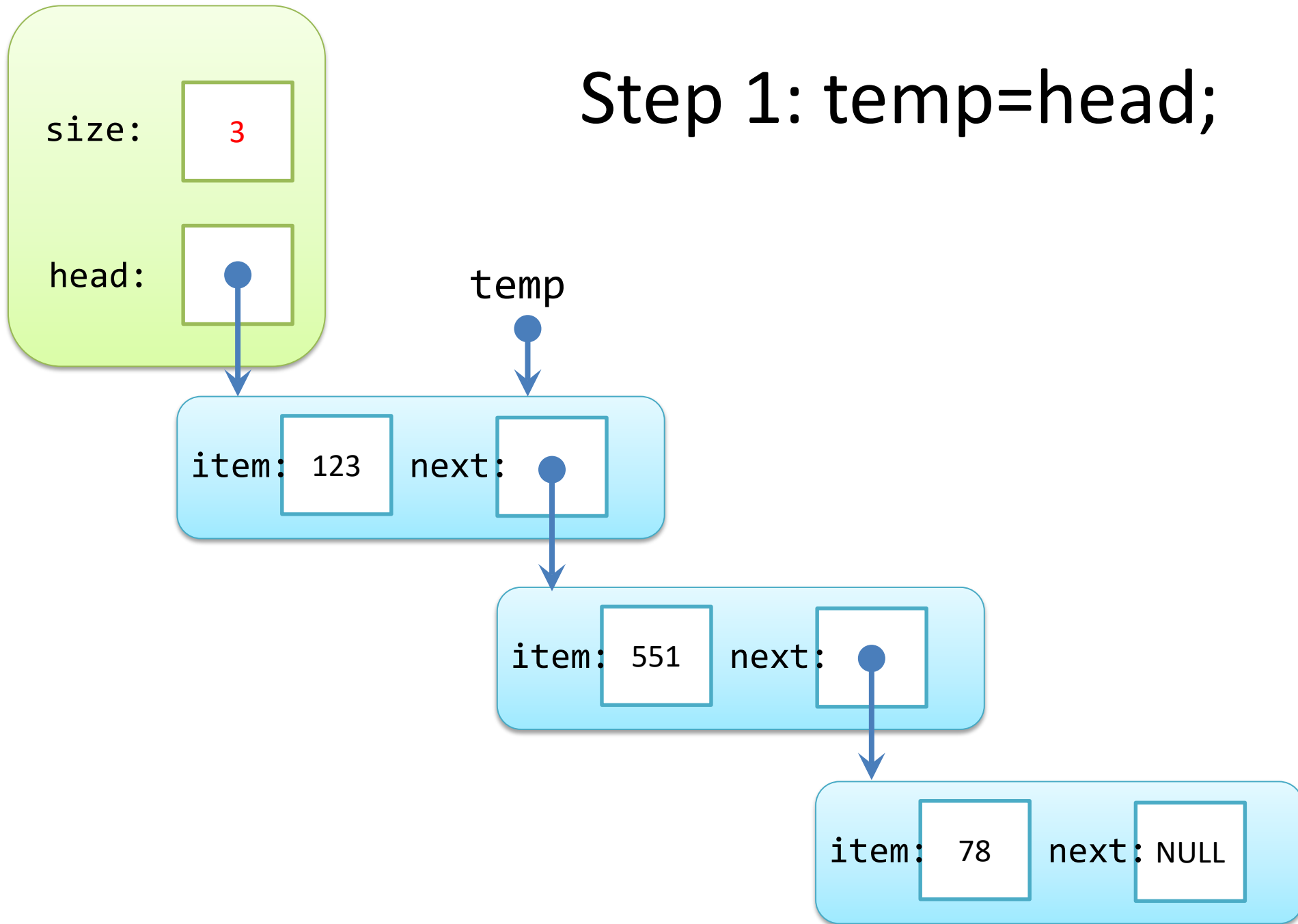
When you use “delete”, you “free” a place in the memory by yourself, instead of automatically, e.g

```
delete temp;
```

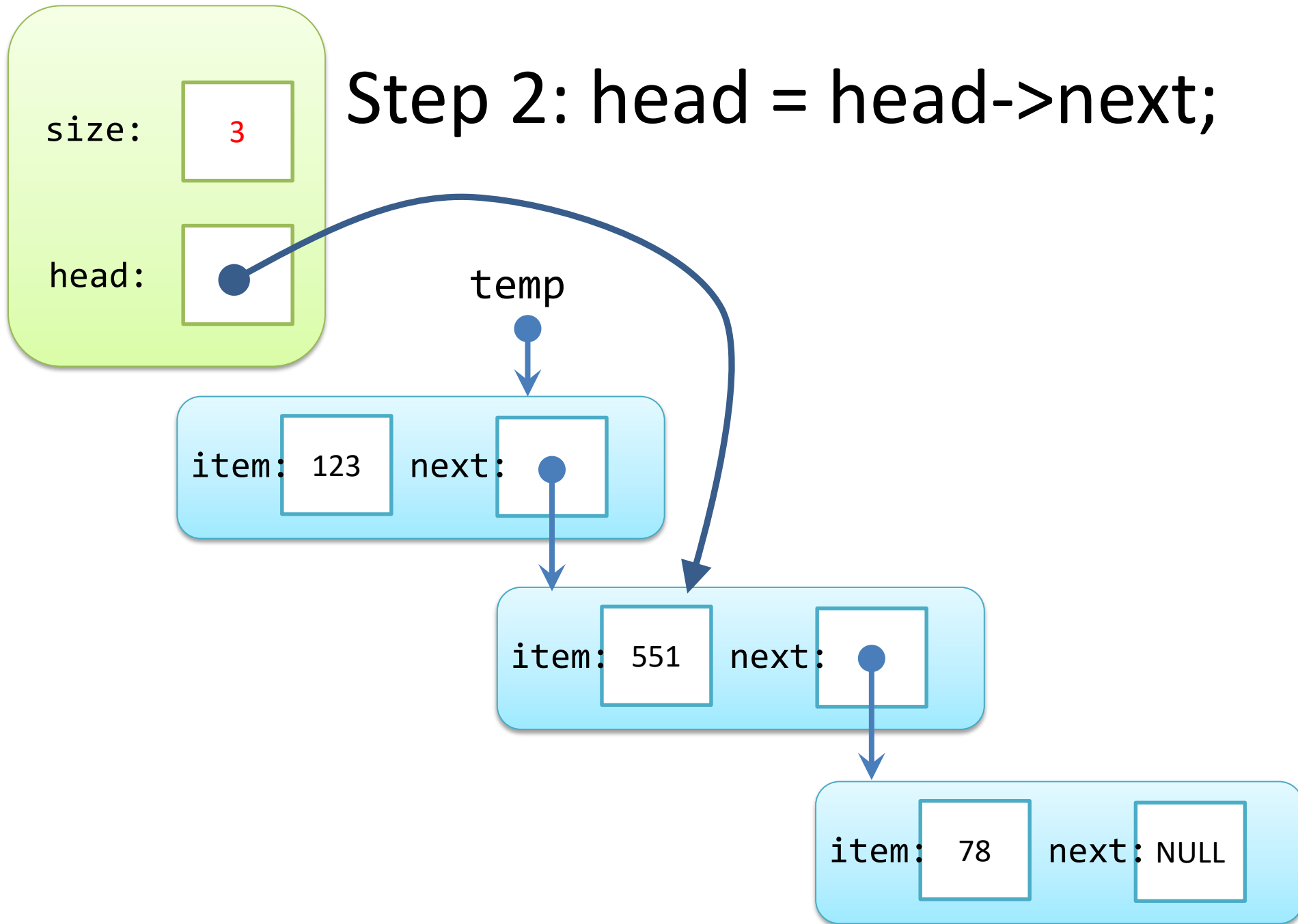
ffbff7db	
ffbff7dc	
ffbff7dd	
ffbff7de	
ffbff7df	
ffbff7e0	
ffbff7e1	
ffbff7e2	



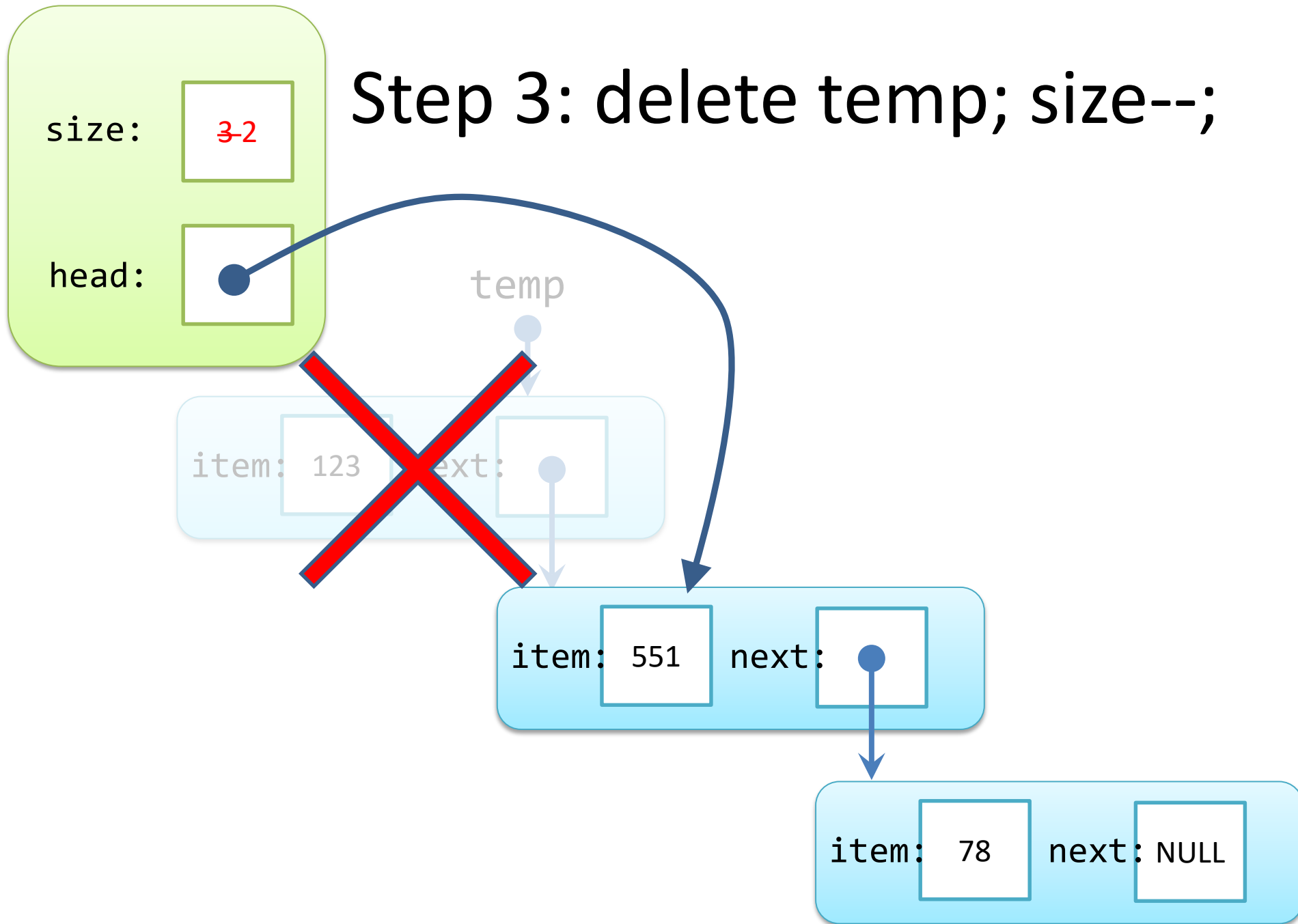
Step 1: temp=head;



Step 2: head = head->next;



Step 3: delete temp; size--;



What If size==0 When We Remove?

- Terminate program



- Print an error message



- Handle the error



Garbage Collection

- Because the node are created (new) by the class List
- When the instance is destroyed, the nodes created will not be deleted automatically
 - Thus, a lot of orphans
- We should delete them when an instance of List is delete
 - By Destructor

Destructor

- **The Constructor** of a class is called when an instance of the class is created
- **The Destructor** will be called when an instance of the class is deleted

Destructor

```
class List {  
    private:  
        int size;  
        ListNode *head;  
    public:  
        List();  
        ~List();  
        void insert(int);  
        void removeHead();  
};  
  
List::~~List() {  
    while(size!=0)  
        removeHead();  
}
```

What is a Constructor?

- A Constructor is called when your object is just created **automatically**
- For example, when you declare it

```
main() {  
    List ll; ← constructor is called here  
    ... some statements ..  
}
```

- Or when you “new” it

```
main() {  
    List *lp; ← constructor is NOT called here  
    .... some statements ...  
    lp = new List; ← constructor is called here  
}
```

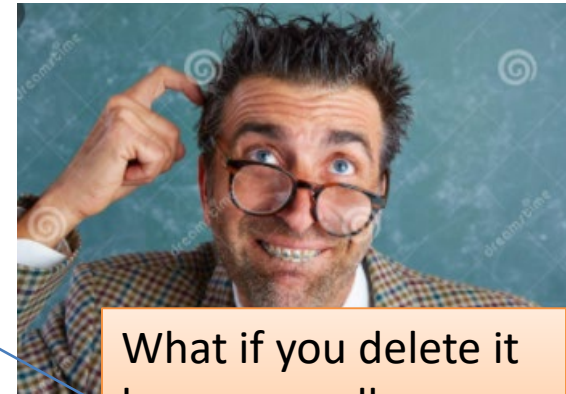

What is a Destructor?

- A Destructor is called when your object is deleted or out of scope **automatically**
- For example, when it is out of scope:

```
main() {  
    List ll; ← constructor is called here  
    ... some statements ...  
} ← Destructor is called here
```

- Or when you delete it

```
main() {  
    List *lp;  
    .... some statements ...  
    lp = new List; ← constructor is called here  
    .... some statements ...  
    delete lp; ← Destructor is called here  
}
```



What if you delete it here manually, e.g. `delete ≪`

Other Variations of Removal

- `removeAtTail`
- `removePos(int)`
- `removeItem(int)`

Why We Use Linked List?

- We can store arbitrarily many elements dynamically
 - Just the right amount of memory we need
- Each insertion is fast!
 - No matter how many items you stored already
- There are other issues we haven't talked about yet, e.g. how to search/access the data

Admin

- You should be able to access
 - Coursemology
 - Archipalego
- Homework 1 starts next week
- Tutorials will start at Week 3