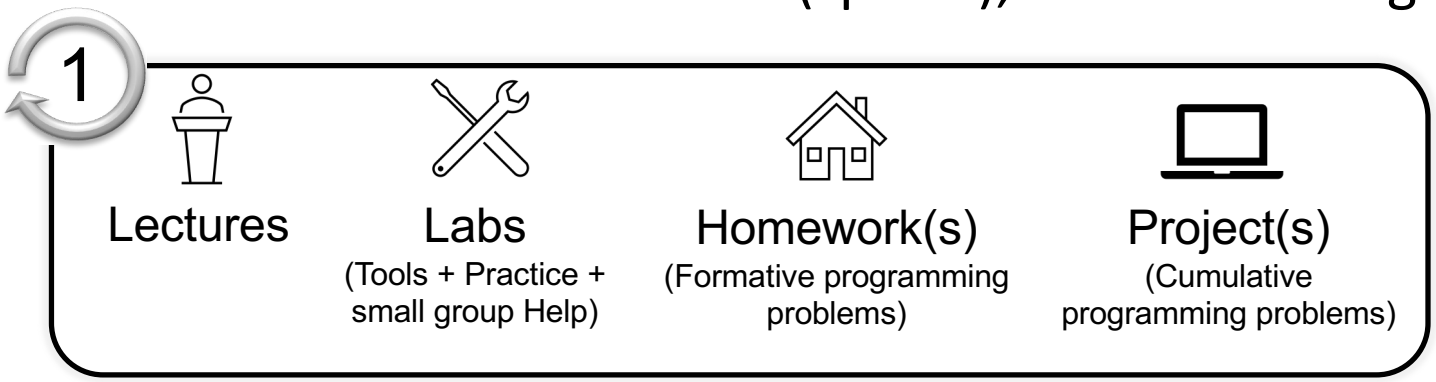
CS103 Unit 4a – Linked Lists

CSCI 103L Teaching Team

Unit 4 – Managing Data

- The course is broken into 6 units (spirals), each consisting of:



1 **C++ Language Syntax**

2 **Algorithms and Computational Thinking**

3 **Objects 1**

4 **Managing Data**

5 **Objects 2**

6 **Recursion**

nullptr Pointer

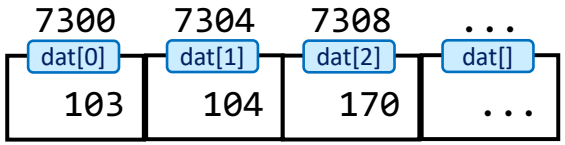
- Recall: Just like there was a null character in ASCII = `'\0'` whose value was 0 there is a **NULL pointer** whose value is **0**
  - `NULL` is "keyword" you can use in C/C++ that is defined to be 0
  - `nullptr` is an equivalent keyword in C++ version 11 and onward and has some advantages best explained later…
    - Requires special compile flags, so we may default to NULL for now
- Used to indicate that the pointer does NOT point at valid data
  - Nothing ever lives at address 0 of memory so we can use it to mean "INVALID" or "pointer to nothing"
  - Often used as an "error" return value from functions returning pointers (See http://www.cplusplus.com/reference/cstring/strchr/)
  - ```
    char* ptr = strchr("Hello", 'h')
    if(ptr != nullptr){ … } // it's a good pointer
    ```
  - `if(ptr){… }`
    ```

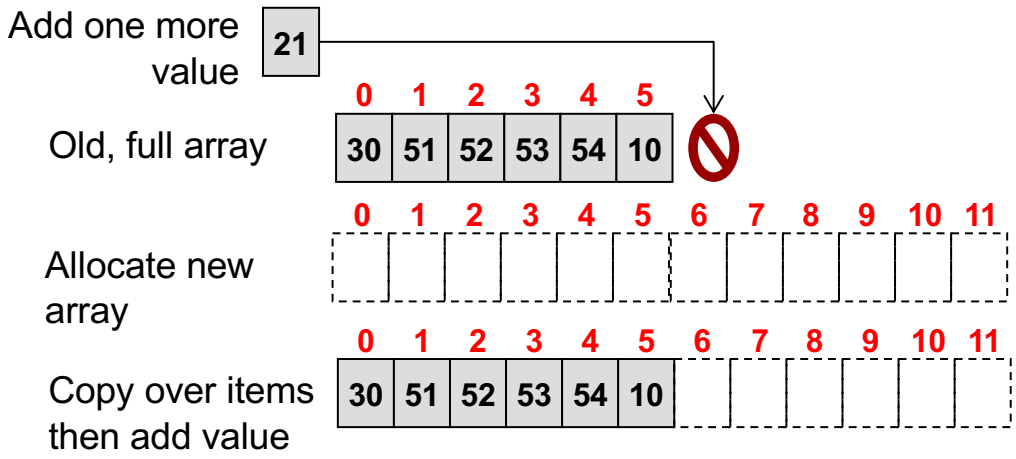
Arrays Review

- **Fast access**: Because arrays are contiguous in memory, we can jump straight to element **i** with only the start address and data type
  - Recall the formula: `start_addr + i*data_size`
  - If we know integer element i is at location 7304, do we know where element i+1 is?
- **Can't grow (resize)**: Once we declare the array (either statically on the stack or dynamically on the heap) we cannot increase its size

```
#include<iostream>
using namespace std;

int main()
{
  int data[25];
  data[20] = 7;
  return 0;
}
```






```
#include<iostream>
using namespace std;

int main()
{
  int size;
  cout << "Enter size: ";
  cin >> size;
  int *ptr = new int[size];

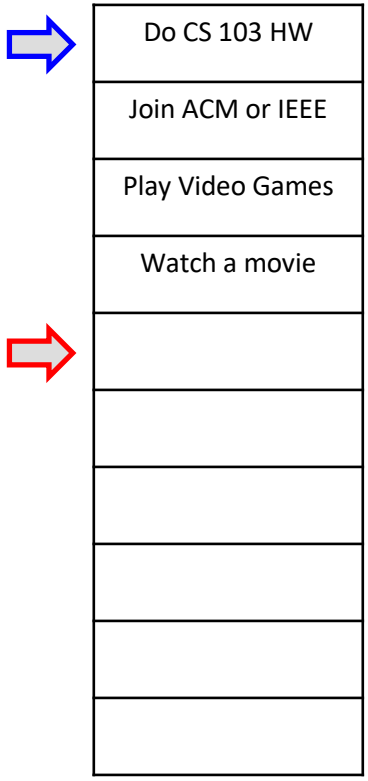
  // What if we end up
  // needing more than size?
}
```

Analogy - Lists

- Natural strategy when we have a set of items that can change is to create a **list**
  - Write down what you know now
  - Can add more items later (usually to the end of the list)
  - Remove (cross off) others when done with them
- Can only do this with an array if you know max size of list ahead of time (which is sometimes fine)
  - We could track the start and end (aka "head" and "tail") but only a portion would be used at a time (wasteful!)

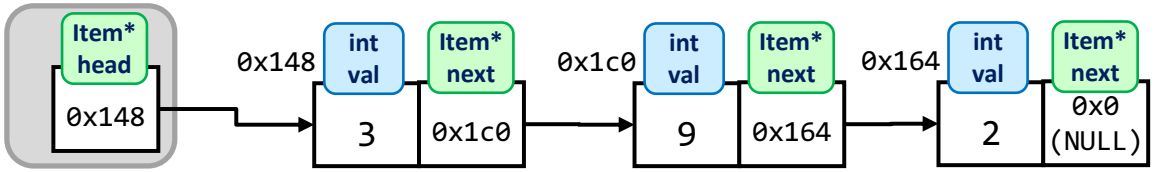
```
1. Do CS 103 HW
2. Join ACM or IEEE
3. Play Video Games
4. Watch a movie
```

```
1. Do CS 103 HW
2. Join ACM or IEEE
3. Play Video Games
4. Watch a movie
5. Exercise
6. Eat dinner
```

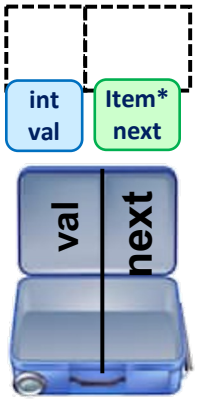


Linked Lists

- A linked list stores values in separate chunks of memory (i.e. a dynamically allocated object)
- To know where the next one is, each one stores a pointer to the next
- All we do is track where the first object is (i.e. the **head** pointer)
- We can allocate more or delete old ones as needed so we only use memory as needed, changing pointers as we go



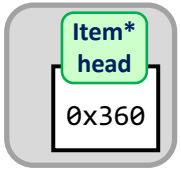
struct Item blueprint:

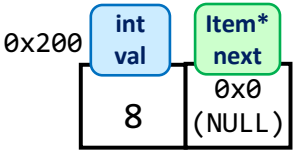


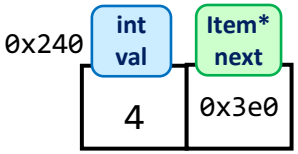

```
struct Item {
  int val;
  Item* next;
};
```

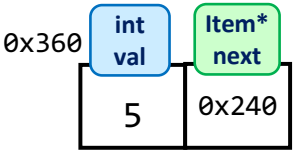
Linked Lists

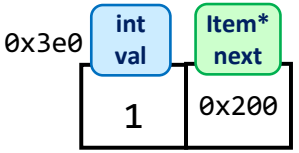
- What is the order of values in this linked list?
- How would you insert 6 at the front of the list?
- How would you remove the value 4?









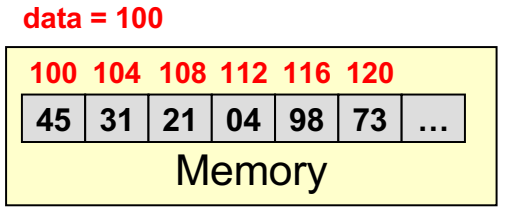


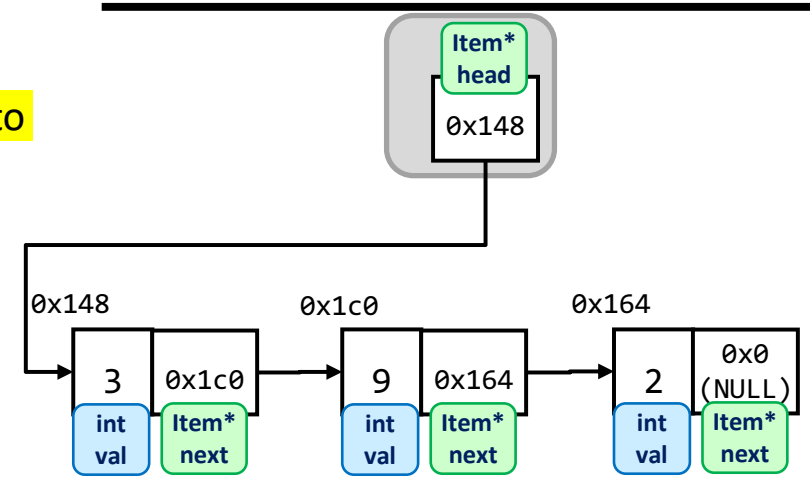
Arrays vs. Linked List

- **Recall**: if we have the start address of an array, we can get the i-th element quickly.
  - Using: `start_addr + i*data_size`
- **Question**: If we have the start (head) pointer to a linked list, can we find the i-th element quickly?
  - No!…Have to walk the linked list
  - Items are **NOT CONTIGUOUS**
- Linked lists trade offs:
  - Pro: the ability to resize (grow/shrink)
  - Con: sacrifice speed of access when attempting to get the element at an arbitrary location

```
#include<iostream>
using namespace std;

int main()
{
  int data[25];
  data[20] = 7;
  return 0;
}
```





In Class Coding

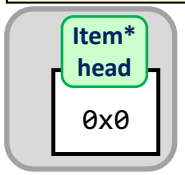
- EdStem/Codio Exercise – Writing a linked list

Linked List Class Overview

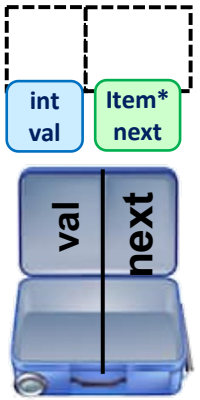
- Use structures/classes and pointers to make **linked** data structures
  - Linked lists, trees, graphs, etc.
- List
  - Arbitrarily sized collection of values
    - Can add or remove any number of new values
  - Only **REQUIRED** data member: **head** pointer
    - Though we can add a few other members to increase efficiency
  - Usually supports following set of operations:
    - Append ("push_back")
    - Prepend ("push_front")
    - Remove back item ("pop_back")
    - Remove front item ("pop_front")
    - Find (look for particular value)

```
struct Item {
  int val;
  Item* next;
};

class List {
public:
  List();
  ~List();
  void push_back(int v); ...
private:
  Item* head;
};
```



struct Item blueprint:



Rule of thumb: Still use 'structs' for objects that are purely collections of data and don't really have operations associated with them. Use 'classes' when data does have associated functions/methods.

Building A Linked List (1)

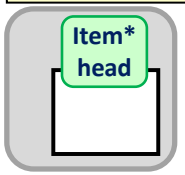
- How should we initialize our list object?

```
#include<iostream>
using namespace std;

List::List()
{
  head = _________________;
}



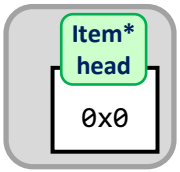

int main()
{
  List mylist;
```

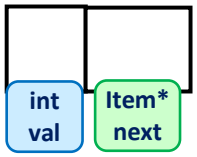


Building A Linked List (2)

- How do we add an element?
- Anytime we code for a linked data structure we should always account for **2 cases**:
  - When the structure is **EMPTY** (and thus head is NULL)
  - When the structure is **NON-EMPTY**
- How should we allocate our `Item`?
  - On the **stack** or on the **heap**?

```
#include<iostream>
using namespace std;
List::List()
{
  head = NULL;
}
void List::push_back(int v){
  if(head == NULL){ // list is empty
    // How should we allocate
    Item newItem;
    head = &newItem;
  }
  else { ... }
}
int main()
{
  List mylist;
  mylist.push_back(3);
}
```

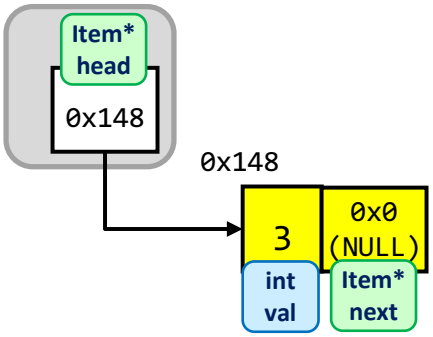




Building A Linked List (3)

- ALL allocations SHOULD be **dynamic allocations on the heap** so they persist beyond the scope of the push_back call.

```
#include<iostream>
using namespace std;
List::List()
{
  head = NULL;
}
void List::push_back(int v){
  if(head == NULL){ // list is empty
     head = new Item;
     head->val = v; head->next = NULL;
  }
  else { ... }
}
int main()
{
  List mylist;
  mylist.push_back(3);
}
```

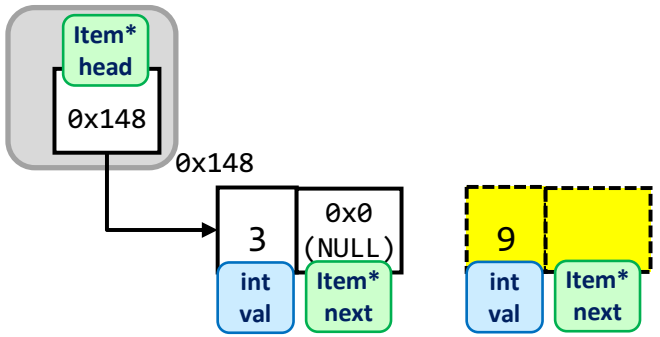


Building A Linked List (4)

- To insert a second item, what would need to change in our current list?

```
#include<iostream>
using namespace std;
List::List()
{
  head = NULL;
}
void List::push_back(int v){
  if(head == NULL){ ...  }
  else {
    // Walk to the last element


 }
}
int main() {
  List mylist;
  mylist.push_back(3);
  mylist.push_back(9);
}
```

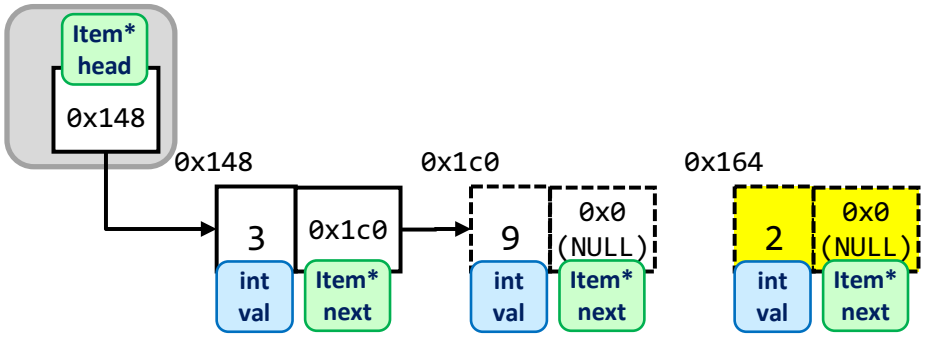


Building A Linked List (5)

- To insert a second item, what would need to change in our current list?
  - **Answer: The last element in the list**
- When the list is NON-EMPTY with ONE or MANY element already in the list, we must **iterate to the last element** in the list to insert our new Item
- Given that we only have the head pointer, how do we **step through** the list?

```
#include<iostream>
using namespace std;
List::List()
{
  head = NULL;
}
void List::push_back(int v){
  if(head == NULL){ ...  }
  else {
    // Walk to the last element


 }
}
int main() {
  List mylist;
  mylist.push_back(3);
  mylist.push_back(9);
  mylist.push_back(2);
}
```



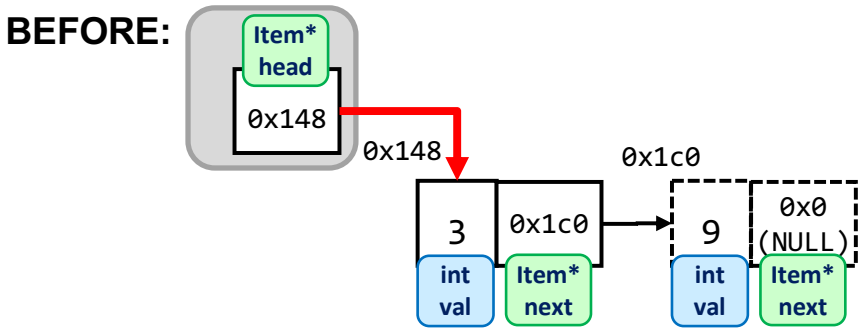
Building A Linked List (6)

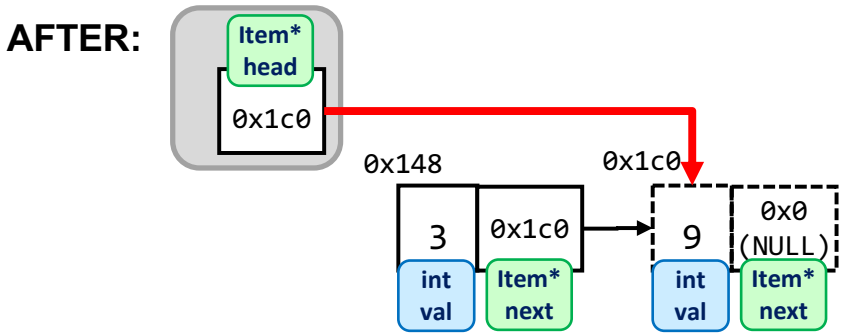
- Given that we only have the head pointer, how do we **step through** the list?
- Would we even WANT TO use head to step through the list?
- No!!!

```
#include<iostream>
using namespace std;
List::List() { ... }

void List::push_back(int v){
  if(head == NULL){ ...  }
  else {
    // Iterate to last element
    while( /* TBD */  ) {

       head = _______________
    }
 }
}
int main() {
  List mylist;
  mylist.push_back(3);
  mylist.push_back(9);
  mylist.push_back(2);
}
```





Common Linked List Mistake 1

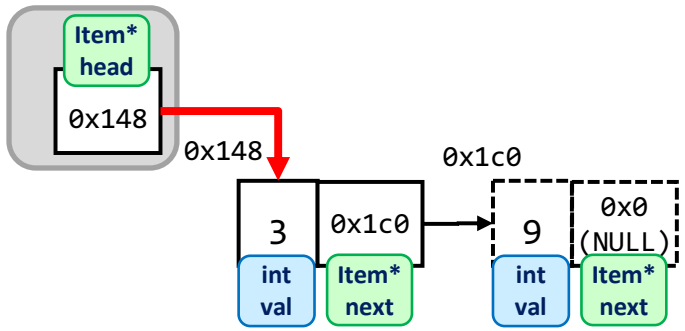
- If we change **head** we have no _____
  - Once we take a step we have "amnesia" and forget where we came from and can't retrace our steps
- **Lesson:** _____!
- Solution: Do NOT change head and use a `temp` pointer

```
#include<iostream>
using namespace std;
List::List() { ... }

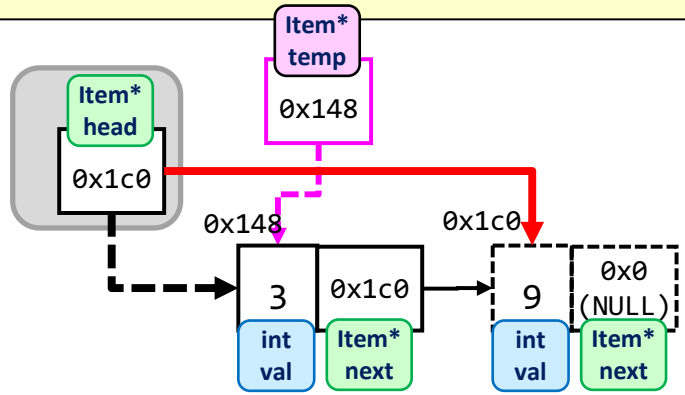
void List::push_back(int v){
  if(head == NULL){ ...  }
  else {
    // Iterate to last element
    while( /* TBD */  ) {

       head = _____________
    }
 }
}
int main() {
  List mylist;
  mylist.push_back(3);
  mylist.push_back(9);
  mylist.push_back(2);
}
```

BEFORE:



AFTER:

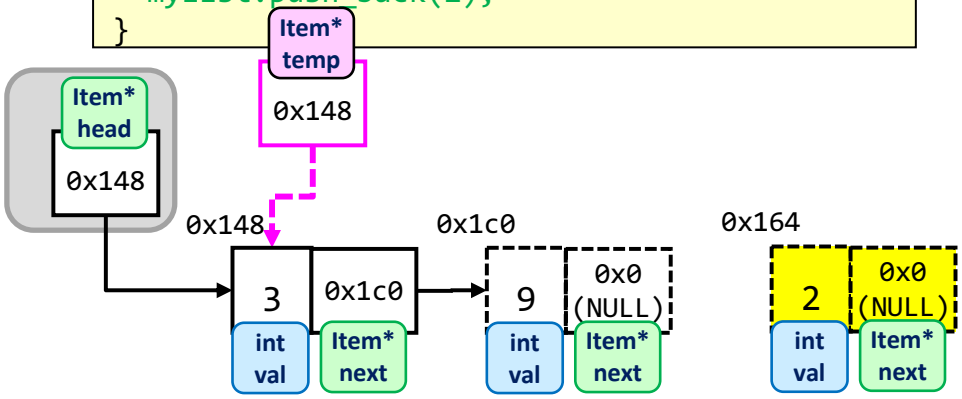


Building a Linked List (7)

- When the list is NOT EMPTY use a **temp** pointer (initialized with head) and then stepping through the list to the last element
- Questions:
  - What code would we use to "take a step"?
  - When would we want to stop (and thus what condition would we use in our while loop)?

```
#include<iostream>
using namespace std;
List::List() { ... }

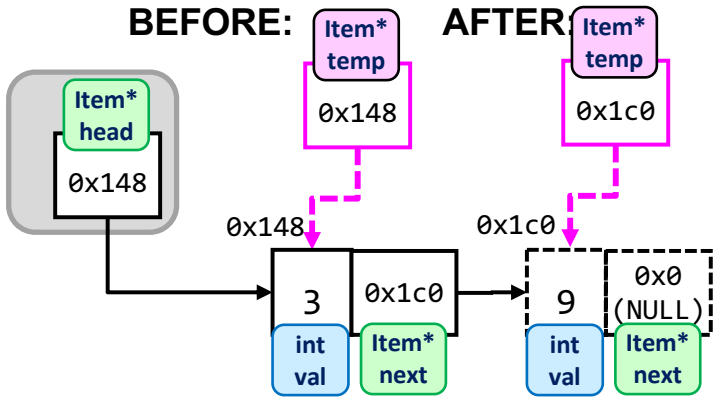
void List::push_back(int v){
  if(head == NULL){ ...  }
  else {
    // Iterate to last element
    Item* temp = head;
    while( /* when to stop? */  ) {
      temp = _________________ // take a step
    }
 }
}
int main() {
  List mylist;
  mylist.push_back(3);
  mylist.push_back(9);
  mylist.push_back(2);
}
```



Common LL Task: Taking a Step

- What is the C++ code to take a step from one item to the next
- Answer:
  - _____
- **Lesson:** To move a pointer to the next item use:

_____

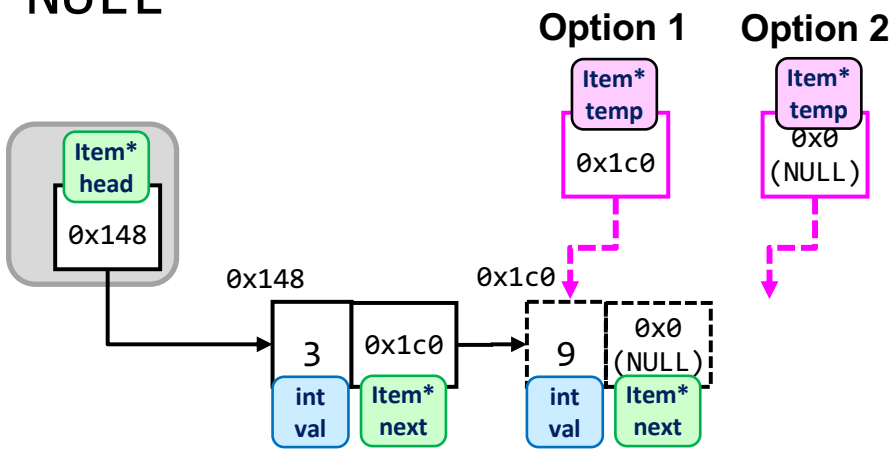



```
#include<iostream>
using namespace std;
List::List() { ... }

void List::push_back(int v){
  if(head == NULL){ ...  }
  else {
    // Iterate to last element
    Item* temp = head;
    while( /* when to stop? */  ) {
      temp = _____ // take a step
    }
 }
}
int main() {
  List mylist;
  mylist.push_back(3);
  mylist.push_back(9);
  mylist.push_back(2);
}
```

When Should We Stop

- Option 1:
  When `temp->next` is NULL?
- Option 2:
  When `temp` is NULL?
- We need to modify the LAST element's next pointer, so we must stop when `temp->next` is NULL




```
#include<iostream>
using namespace std;
List::List() { ... }

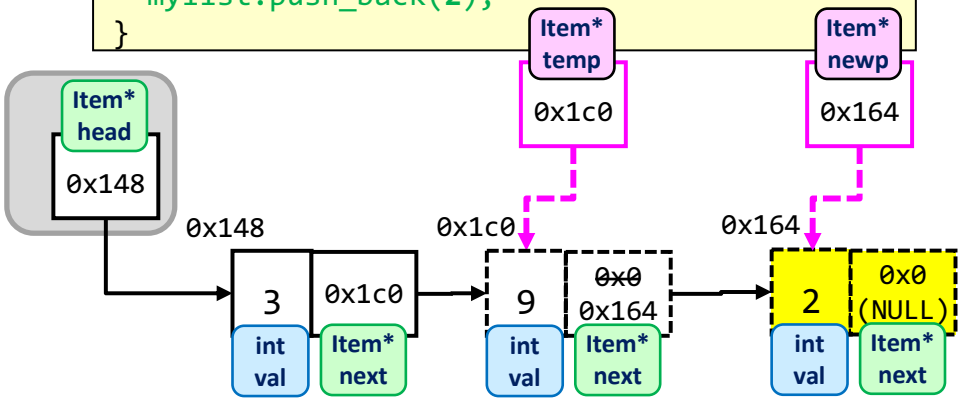
void List::push_back(int v){
  if(head == NULL){ ...  }
  else {
    // Iterate to last element
    Item* temp = head;
    while( /* when to stop? */  ) {
      temp = temp->next // take a step
    }
 }
}
int main() {
  List mylist;
  mylist.push_back(3);
  mylist.push_back(9);
  mylist.push_back(2);
}
```

Building a Linked List (8)

- When the list is NOT EMPTY use a **temp** pointer (initialized with head) and then stepping through the list to the last element
- Solution shown

```
List::List() { ... }

void List::push_back(int v){
  if(head == NULL){ ...  }
  else {
    Item* temp = head;
    while( temp->next != NULL  ) {
      temp = temp->next // take a step
    }
    Item* newp = new Item;
    newp->val = v; newp->next = NULL;
    temp->next = newp;
  }
}
int main() {
  List mylist;
  mylist.push_back(3);
  mylist.push_back(9);
  mylist.push_back(2);
}
```



Removing Items (e.g. the front) [1]

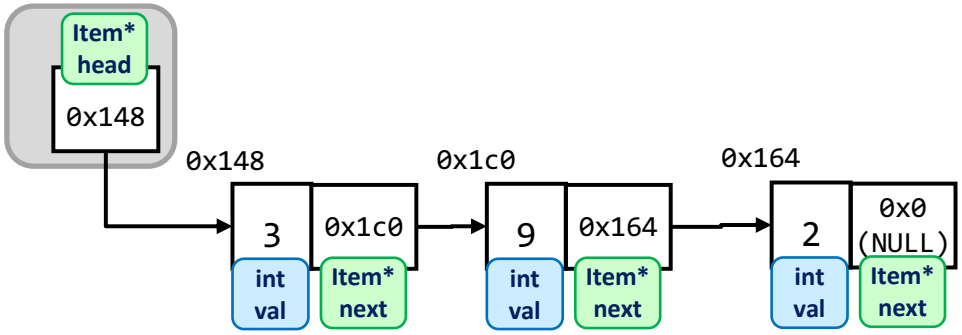
- Now let's look at removal
- What code is necessary to remove the FRONT item?
  - Modify the diagram below to show what we would WANT to happen.

```
List::List() { ... }
void List::push_back(int v) { ... }

void List::pop_front()
{
  // What code goes here



}
int main() {
  List mylist;
  mylist.push_back(3);
  mylist.push_back(9);
  mylist.push_back(2);
  mylist.pop_front();
}
```



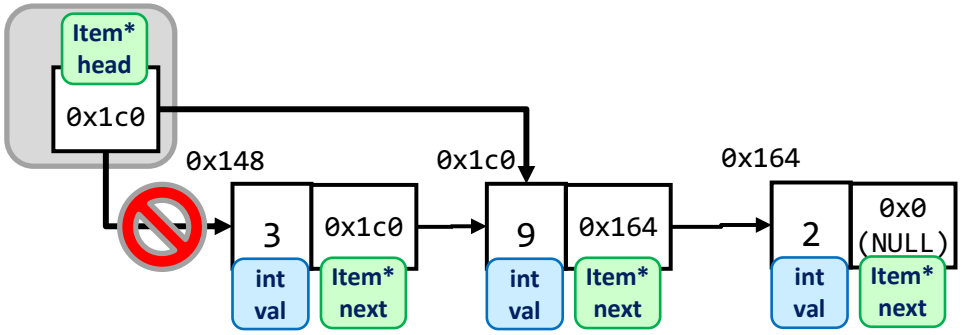
Removing Items (e.g. the front) [2]

- Could we simply update head?
- Unfortunately, that causes a problem.

```
List::List() { ... }
void List::push_back(int v) { ... }

void List::pop_front()
{
  // What code goes here
  head = head->next;


}
int main() {
  List mylist;
  mylist.push_back(3);
  mylist.push_back(9);
  mylist.push_back(2);
  mylist.pop_front();
}
```



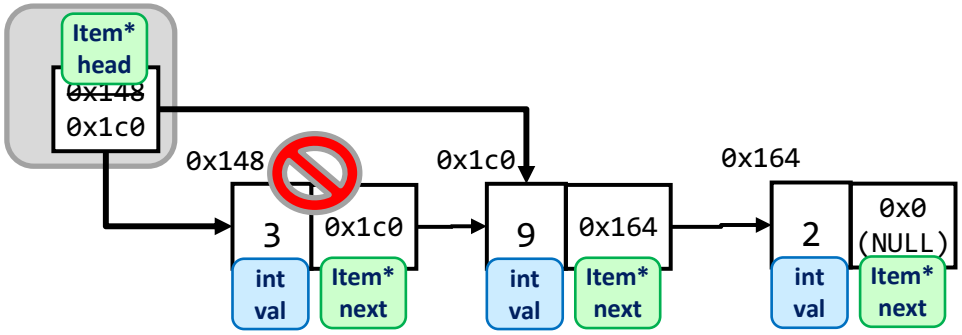
Removing Items (e.g. the front) [3]

- To avoid a memory leak, we CANNOT just change head.
- We need to free/delete the old first item!
- But what is wrong with the given code?

```
List::List() { ... }
void List::push_back(int v) { ... }

void List::pop_front()
{
  // What code goes here
  delete head;
  head = head->next;

}
int main() {
  List mylist;
  mylist.push_back(3);
  mylist.push_back(9);
  mylist.push_back(2);
  mylist.pop_front();
}
```



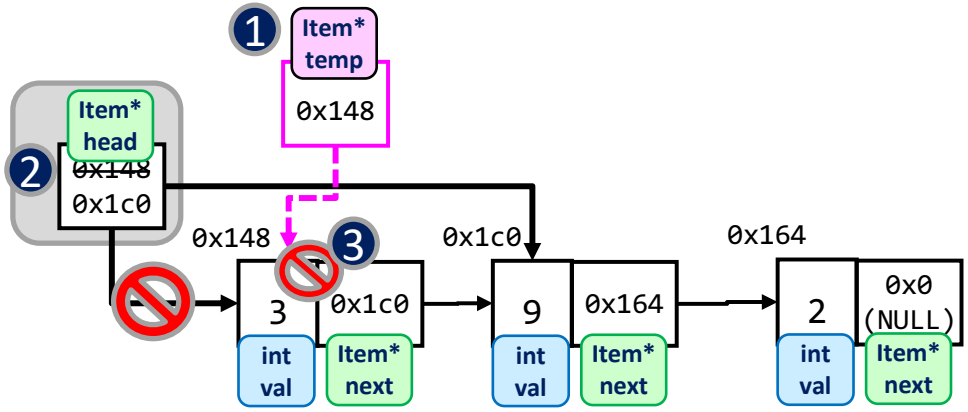
Removing Items (e.g. the front) [4]

- To avoid the chicken vs. egg problem about deleting the first time vs. using it to get the second items address, let's introduce a temp pointer

1. Introduce a temp pointer that is a copy of head
2. Get the next pointer from the first item to update head
3. Delete the old first item

```
List::List() { ... }
void List::push_back(int v) { ... }

void List::pop_front()
{
  // What code goes here
  Item* temp = head;
  head = temp->next;
  delete temp;
}
int main() {
  List mylist;
  mylist.push_back(3);
  mylist.push_back(9);
  mylist.push_back(2);
  mylist.pop_front();
}
```

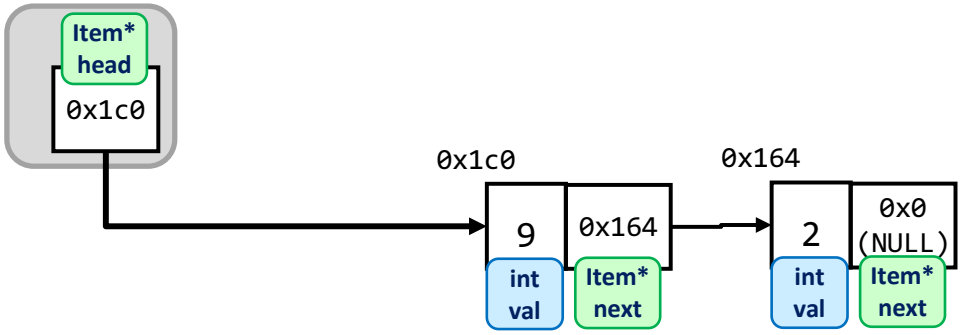


Final Result After Remove

- After execution of `pop_front()`

```
List::List() { ... }
void List::push_back(int v) { ... }

void List::pop_front()
{
  // What code goes here
  Item* temp = head;
  head = temp->next;
  delete temp;
}
int main() {
  List mylist;
  mylist.push_back(3);
  mylist.push_back(9);
  mylist.push_back(2);
  mylist.pop_front();
}
```



Destructors

- **Destructor** (**dtor** for short) is a function of the same name as class itself with a ~ in front (e.g. ~Deck())
  - Called automatically when object goes out of scope (i.e. when it is deallocated by 'delete' or when scope completes)
  - Use to free/delete any memory allocated by the object or close any open file streams, etc.
  - **Returns nothing**
  - **[Note: The Deck class did not require a destructor]**

deck.h

```
class Deck {
 public:
   Deck();   // Constructor
   ~Deck();  // Destructor
   ...
};
```

deck.cpp

```
#include<iostream>
#include "deck.h"

Deck::Deck() {
  ...
}

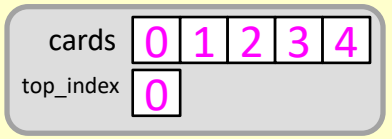
Deck::~Deck()
{ /* not needed for this class */ }
```

cardgame.cpp

```
#include "deck.h"

int main() {
  Deck d;  // Deck() is called
  ...

  return 0;
  // ~Deck() is called since
}
```



Need for Destructor

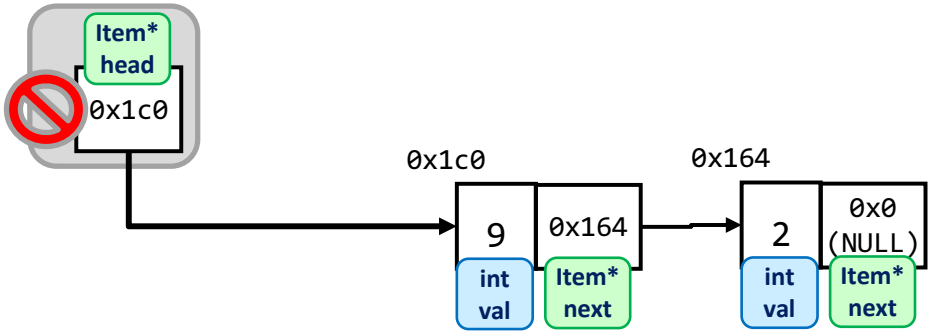
- **Important: Data members of an object are cleaned up / destroyed automatically by the destructor (without you adding any code).**
- **Implication:**
  - **Your destructor does NOT need to worry about deallocation or cleaning up your data members…**
  - **Your destructor NEEDS to clean up things that your data members POINT TO or REFERENCE that won't be cleaned up if your data member(s) die**
- **So, we'd need to deallocate each item remaining in the list.**

```
List::List() { ... }

List::~List() {
  // Do we need code to clean up
  // what our members refer to
}

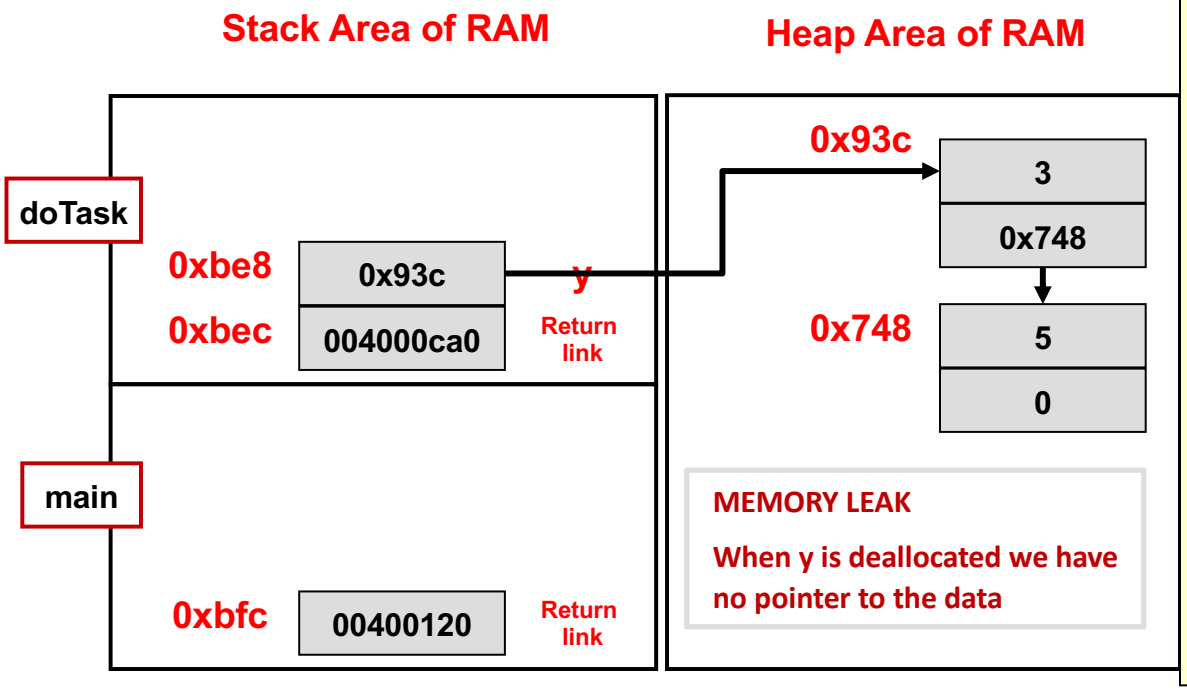
void List::push_back(int v) { ... }
void List::pop_front() { ... }

int main() {
  List mylist;
  mylist.push_back(3);
  mylist.push_back(9);
  mylist.push_back(2);
  mylist.pop_front();
  return 0;  // ~List() is called on mylist
}
```



Destructors: Another Example

- The LinkedList object is allocated as a static/local variable
  - But each element is allocated on the heap
- When y goes out of scope only the data members are deallocated
  - You may have a memory leak




```
struct Item {
  int val;  Item* next;
};
class LinkedList {
  public:
   // create a new item
   // in the list
   void push_back(int v);
  private:
   Item* head;
};

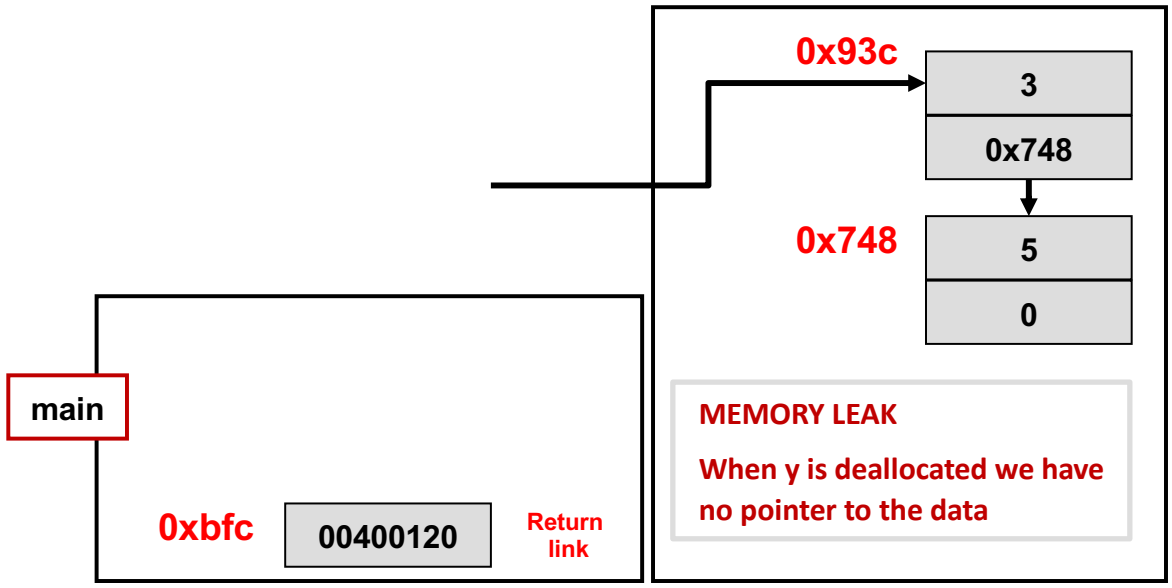
int main()
{
  doTask();
}

void doTask()
{
  LinkedList y;
  y.push_back(3);
  y.push_back(5);
  /* other stuff */
}
```

Destructors: Another Example

- The LinkedList object is allocated as a static/local variable
  - But each element is allocated on the heap
- When y goes out of scope only the data members are deallocated
  - You may have a memory leak

An Appropriate Destructor Will Help Solve This




```
struct Item {
  int val;  Item* next;
};
class LinkedList {
  public:
   // create a new item
   // in the list
   void push_back(int v);
  private:
   Item* head;
};

int main()
{
  doTask();
}

void doTask()
{
  LinkedList y;
  y.push_back(3);
  y.push_back(5);
  /* other stuff */
}
```

Comparing Performance

Arrays

- Go to element at index i
  - O(___)
- Add something to the tail (assume you have a tail index)
  - O(___)
- Adding something to the front of the list after there are already n elements
  - O(___)

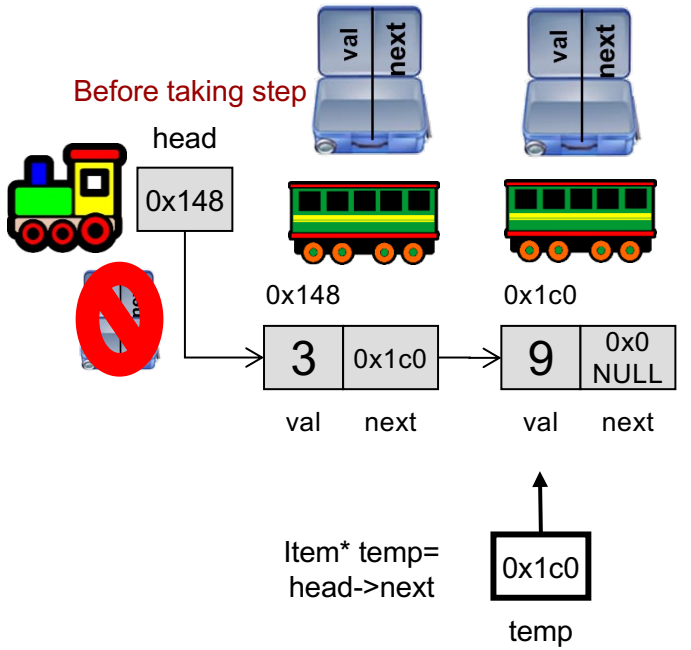
Linked Lists

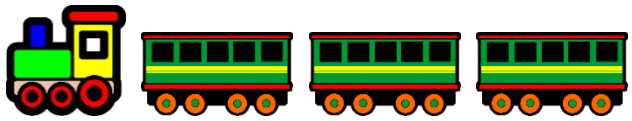
- Go to element at index i
  - O(__)
- Add something to the tail (assume you have only head pointer and n elements in the list)
  - O(__)
- Adding something to the front of the list after there are already n elements
  - O(__)

ODDS AND ENDS

Common Linked Task/Mistake 1

- What code will give us a pointer to the beginning item in the list?
- Mistake: Many students think `head->next` is how you get a pointer to the first item:
  - `Item* temp = head->next;`
- Just use head to get the pointer to the beginning item.
- head is special! It is NOT an actual ITEM struct
  - head is just a pointer that points to the beginning (data-filled) item struct
  - `head->next` actually points to the 2nd item, not the 1st because head already points to the 1st item
- **Lesson:** To get a pointer to the **FIRST** item, just use _____

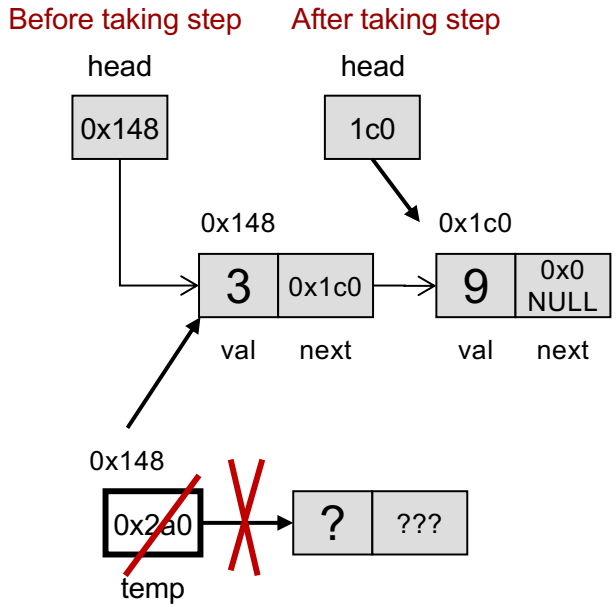




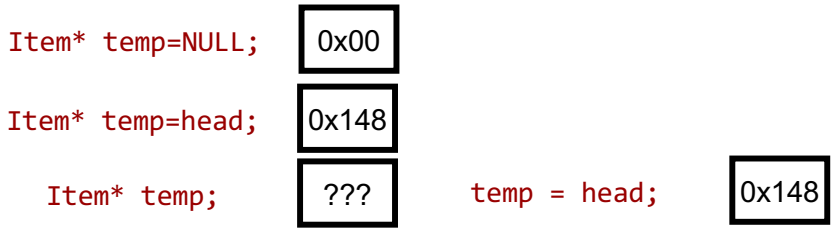
Engine = "head" Each car = "Item"

Common Linked Task/Mistake 2

- Common errors we see is that to create a temporary pointer students also dynamically allocate an item and then immediately point it at something else causing a memory leak
  - `Item* temp = _____;`
  - `temp = head;`
  - You may declare pointers w/o having to allocate anything
  - `Item* temp;`
  - `Item* temp = NULL;`
  - `Item* temp = head;`
- **Lesson:** Only use 'new' when you really want a new Item to come alive



Mistake: Allocating an item when you declare a temporary pointer



Exercises

- In-class exercises:
  - monkey_traverse
  - monkey_addstart
- Codio Exercise –
  - Writing a linked list



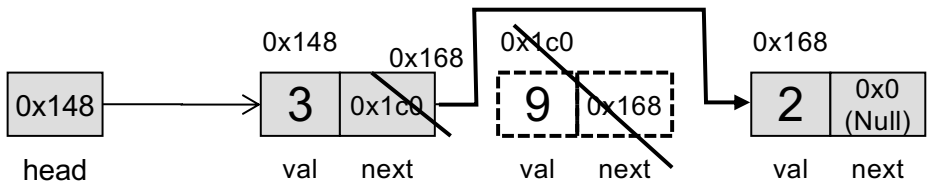
Childs toy "Barrel of Monkeys" let's children build a chain of monkeys that can be linked arm in arm

http://www.toysrus.com/graphics/tru_prod_images/Barrel-of-Monkeys-Game----pTRU1-2907042dt.jpg

Other Functions

- Write a function to print all items in list
  - Copy head to a temp pointer then use it to iterate over the items until the next pointer is NULL
  - Print each item as you iterate
- Find if an item in the list (return address of struct if present or NULL)
  - Copy head to a temp pointer then use it to iterate over the items until you find an item with the desired value or until next pointer is NULL
- Remove item with given value [i.e. find and remove]
  - If found, need to change the next link of the previous item to point at the item after the item found

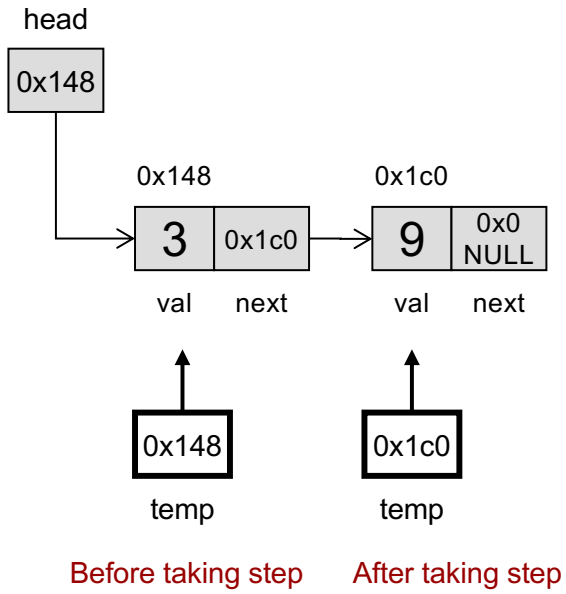
Remove VAL=9



SOLUTIONS

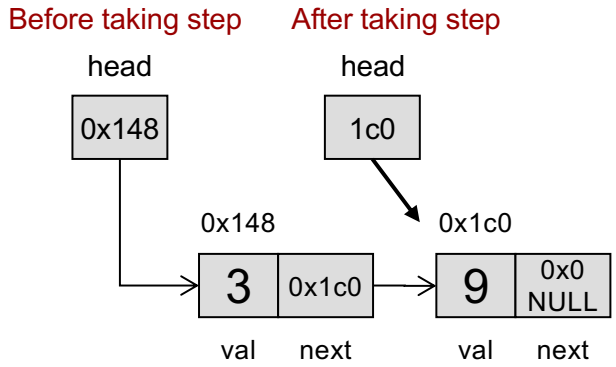
Common Linked Task/Mistake 1

- What is the C++ code to take a step from one item to the next
- Answer:
  - `temp = temp->next`
- **Lesson:** To move a pointer to the next item use:
  `'ptr = ptr->next'`



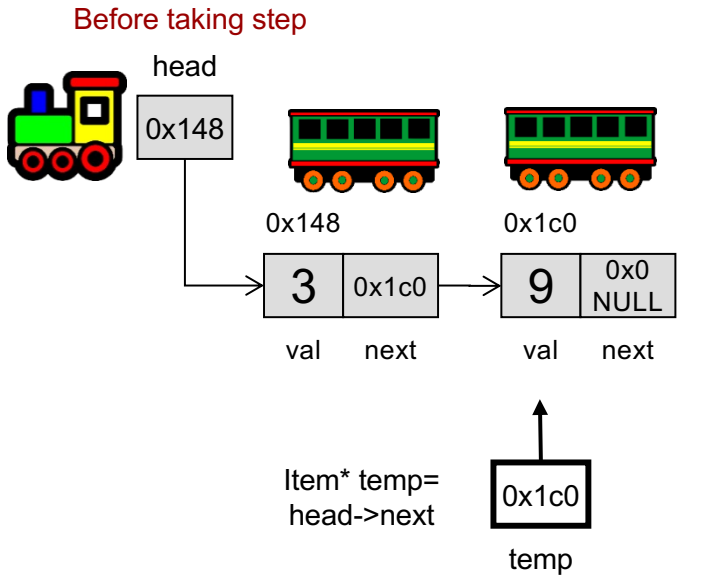
Common Linked Task/Mistake 2

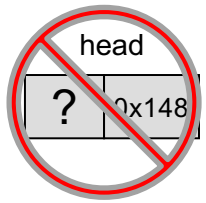
- Why do we need a temp pointer? Why can't we just use head to take a step as in:
  - `head = head->next;`
- Because if we change head we have no record of where the first item is
  - Once we take a step we have "amnesia" and forget where we came from and can't retrace our steps
- **Lesson:** Don't lose your head!



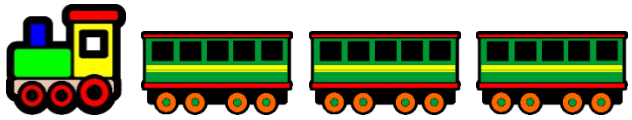
Common Linked Task/Mistake 3

- Mistake: Many students use the following code to get a pointer to the first item:
  - Item* temp = head->next;
- head is special! It is NOT an actual ITEM struct
  - head is just a pointer
  - It just points at the first data-filled struct
  - `head->next` actually points to the 2nd item, not the 1st because head already points to the 1st item
- **Lesson:** To get a pointer to the **FIRST** item, just use '**head**'





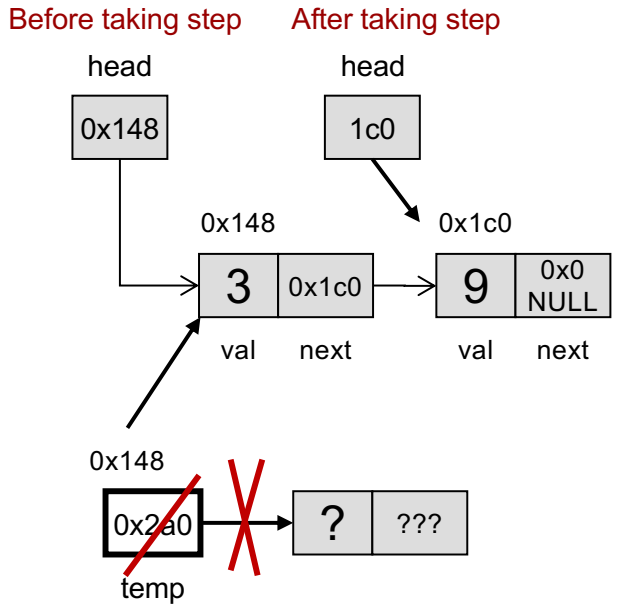
Mistake: Students think head is an Item



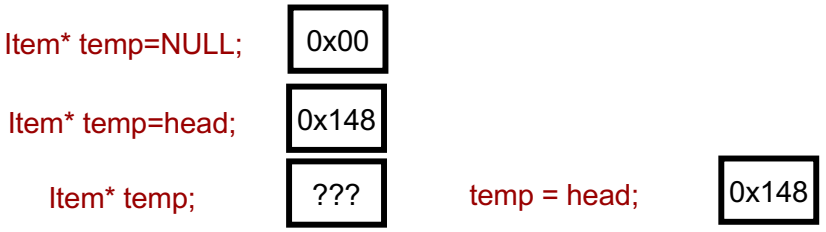
Engine = "head" Each car = "Item"

Common Linked Task/Mistake 4

- Common errors we see is that to create a temporary pointer students also dynamically allocate an item and then immediately point it at something else causing a memory leak
  - `Item* temp = new Item;`
  - `temp = head; or temp = head->next;`
- You may declare pointers w/o having to allocate anything
  - `Item* temp;`
  - `Item* temp = NULL;`
  - `Item* temp = head;`
- **Lesson:** Only use 'new' when you really want a new Item to come alive



Mistake: Allocating an item when you declare a temporary pointer



Comparing Performance

Arrays

- Go to element at index I
  - O(1)
- Add something to the tail (assume you have a tail index)
  - O(1)
- Adding something to the front of the list after there are already n elements
  - O(n)

Linked Lists

- Go to element at index i
  - O(i)
- Add something to the tail (assume you have only head pointer and n elements in the list)
  - O(n)
- Adding something to the front of the list after there are already n elements
  - O(1)