CSCI 104

Rafael Ferreira da Silva
rafsilva@isi.edu

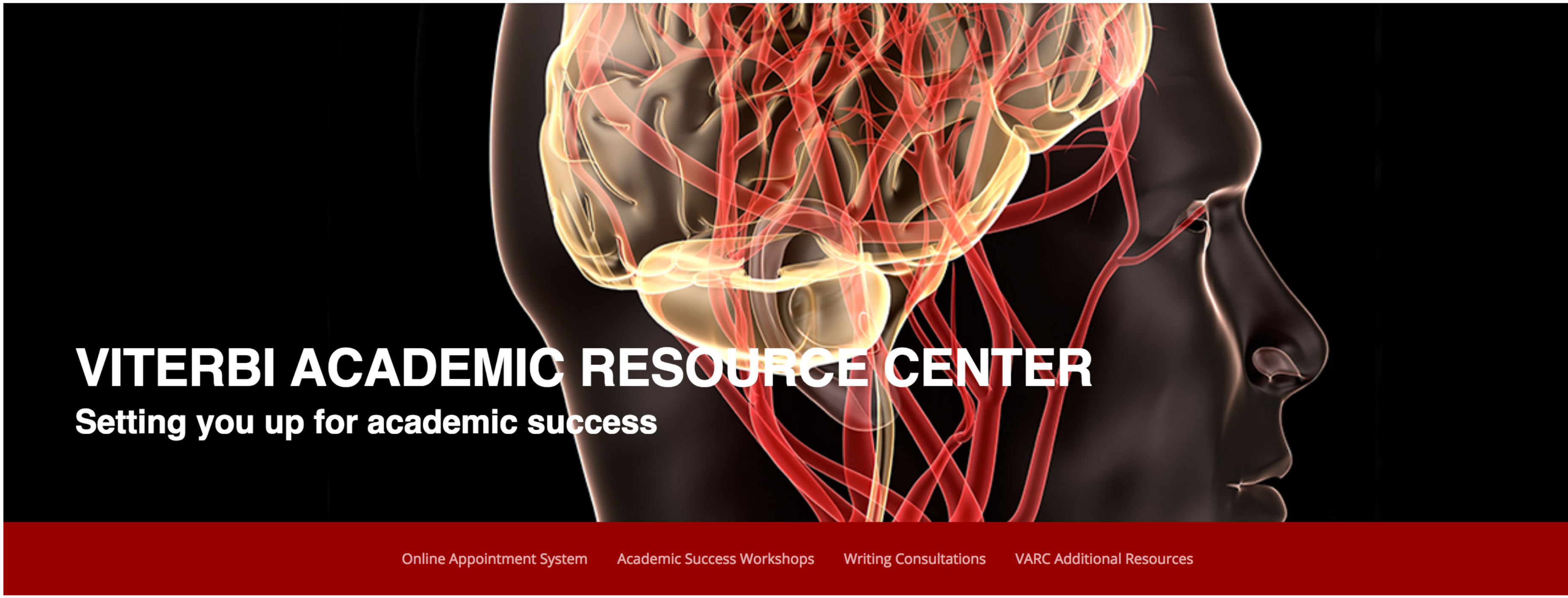
Slides adapted from: Mark Redekopp and David Kempe

XKCD #1739

Courtesy of Randall Munroe @ http://xkcd.com

USC Viterbi UNDERGRAD

MY ACADEMIC EXPERIENCE BUILDING MY FUTURE MY VITERBI LIFE



http://viterbiundergrad.usc.edu/varc/

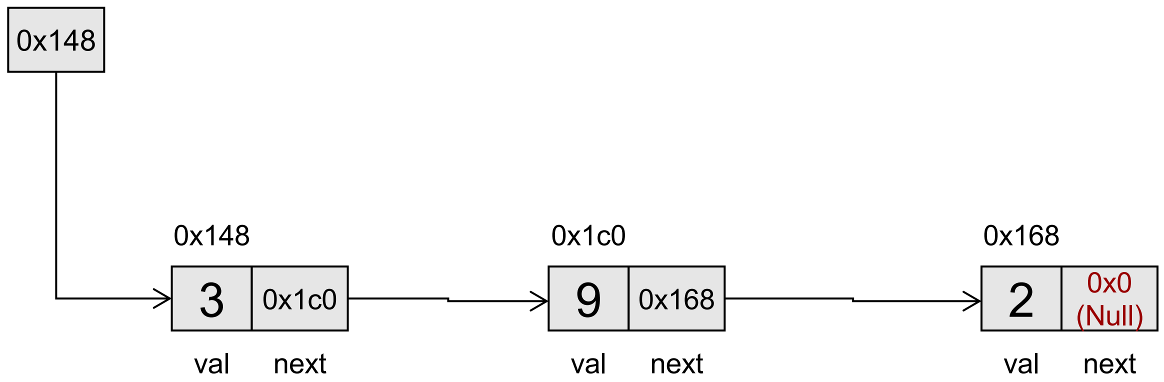
Offering tutoring for CSCI 104

http://viterbiundergrad.usc.edu/varc/#onlineappointment

RECURSION & LINKED LISTS

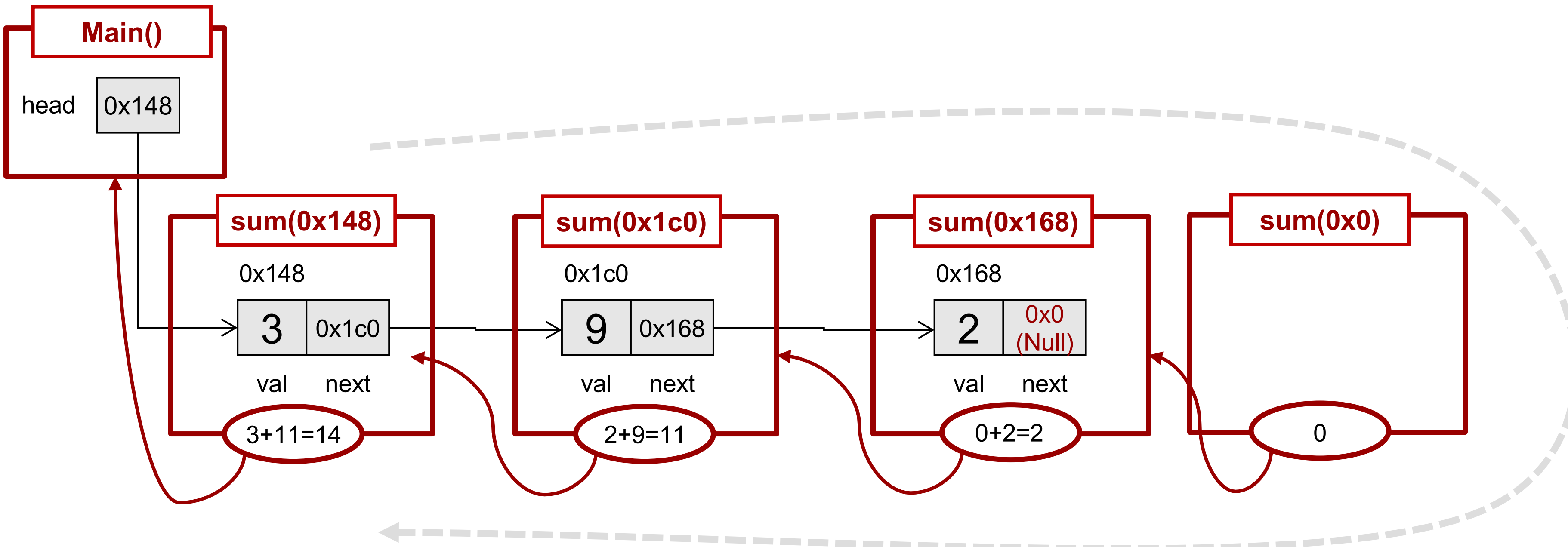
Summing the Values

- Write a recursive routine to sum the values of a linked list
  - Head Recursion (recurse first, do work on the way back up)
  - Tail Recursion (do work on the way down, then recurse)



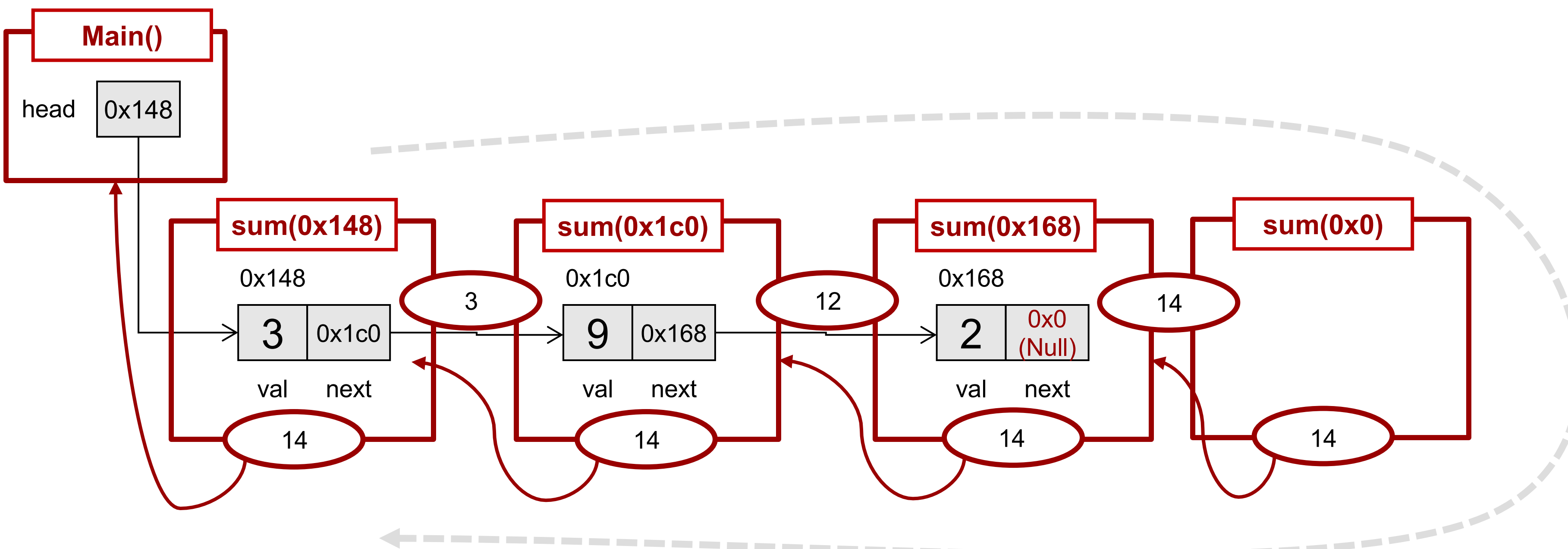
Head Recursion

- Recurse to the end of the chain (head == NULL) and then start summing on the way back up
  – What should the base case return
  – What should recursive cases (normal nodes) return?



Tail Recursion

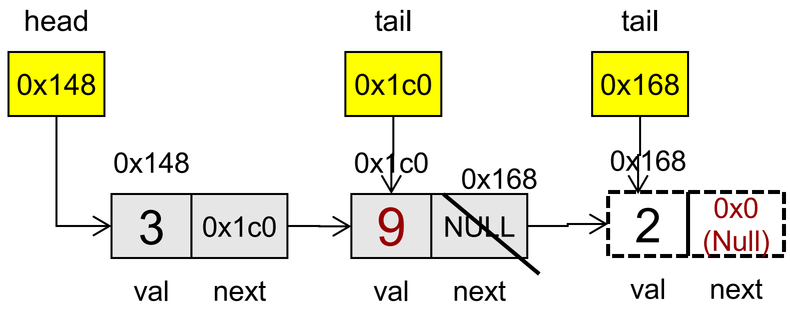
- Produce sum as you walk down the list then just return the final answer back up the list



INCREASING EFFICIENCY OF OPERATIONS + DOUBLY LINKED LISTS

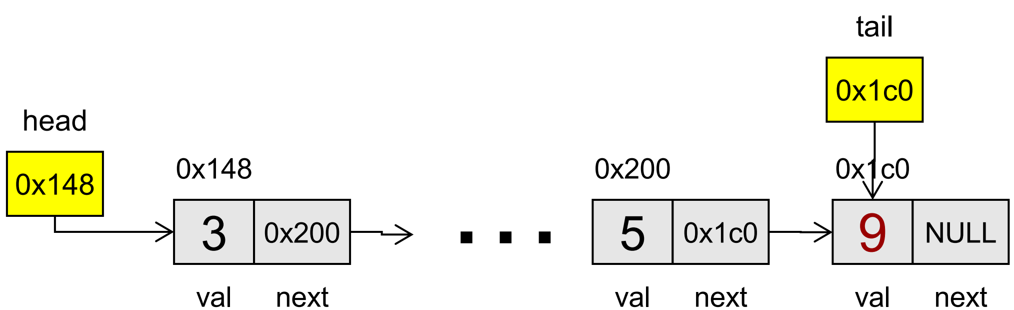
Adding a Tail Pointer

- If in addition to maintaining a head pointer we can also maintain a tail pointer
- A tail pointer saves us from iterating to the end to add a new item
- Need to update the tail pointer when…
  - We add an item to the end (fast)
  - We remove an item from the end (slow)



Removal

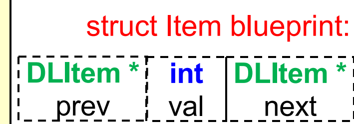
- To remove the last item, we need to update the 2nd to last item (set it's next pointer to NULL)
- We also need to update the tail pointer
- But this would require us to traverse the full list
- ONE SOLUTION: doubly-linked list

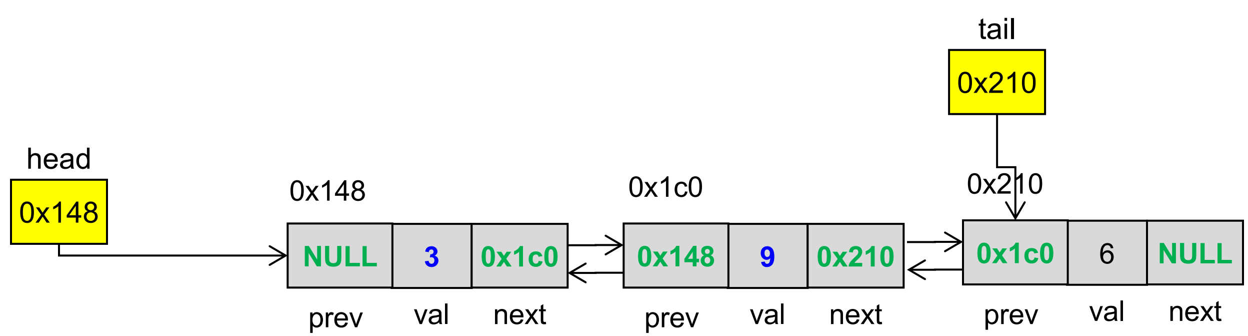


Doubly-Linked Lists

- Includes a previous pointer in each item so that we can traverse/iterate backwards or forward
- First item's previous field should be NULL
- Last item's next field should be NULL

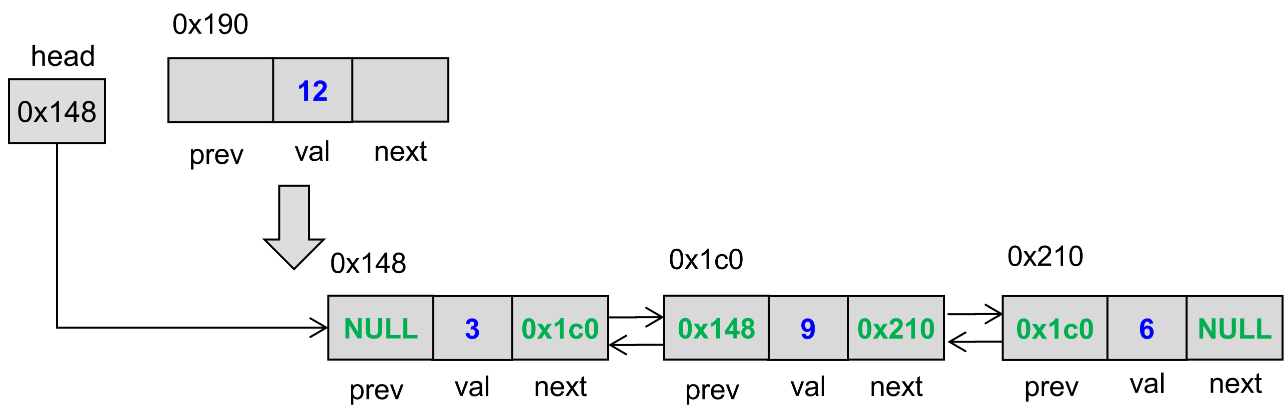
```
#include<iostream>

using namespace std;
struct DLItem {
  int val;
  DLItem* prev;
  DLItem* next;
};

int main()
{
  DLItem* head, *tail;
};
```





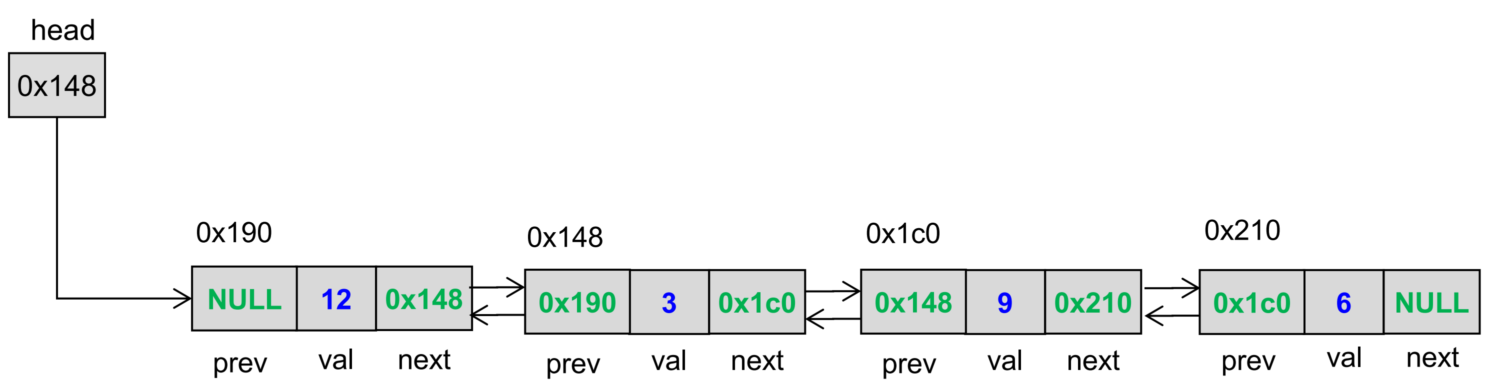
Doubly-Linked List Add Front

- Adding to the front requires you to update…
- …Answer
  - Head
  - New front's next & previous
  - Old front's previous



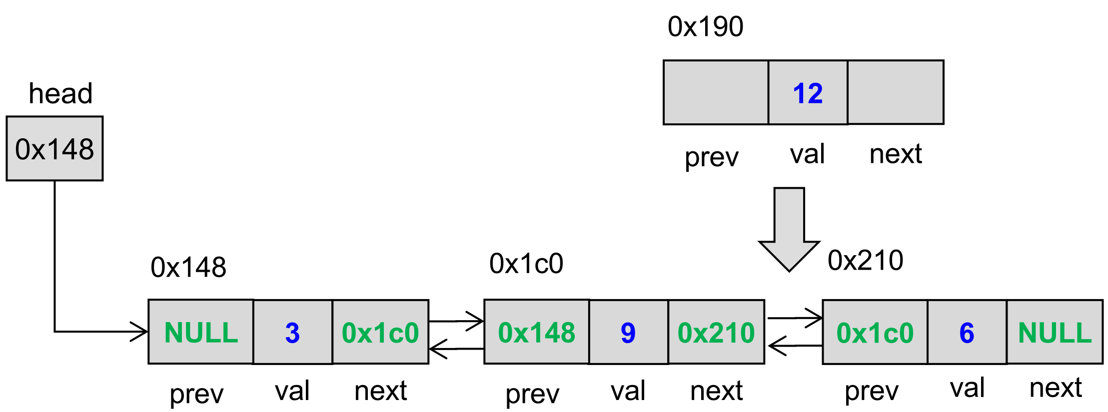
Doubly-Linked List Add Front

- Adding to the front requires you to update…
  - Head
  - New front's next & previous
  - Old front's previous



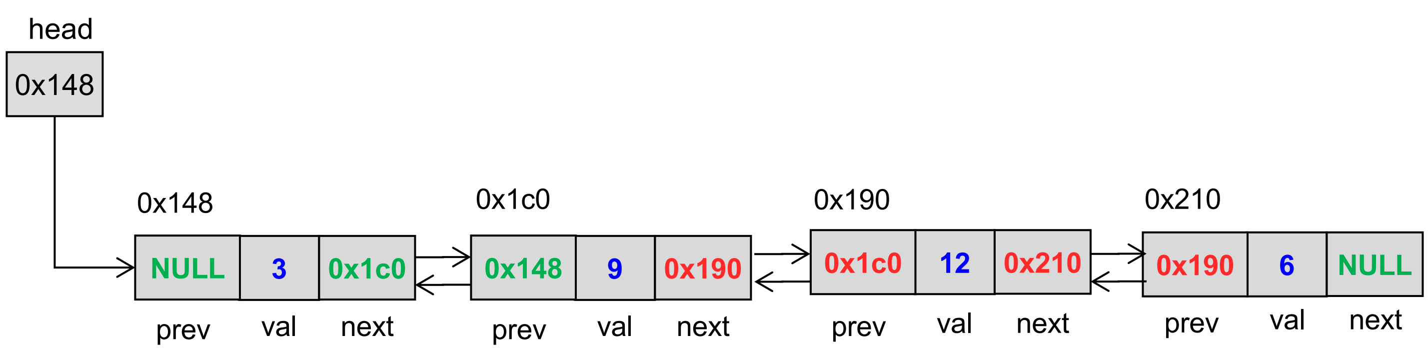
Doubly-Linked List Add Middle

- Adding to the middle requires you to update…
  - Previous item's next field
  - Next item's previous field
  - New item's next field
  - New item's previous field



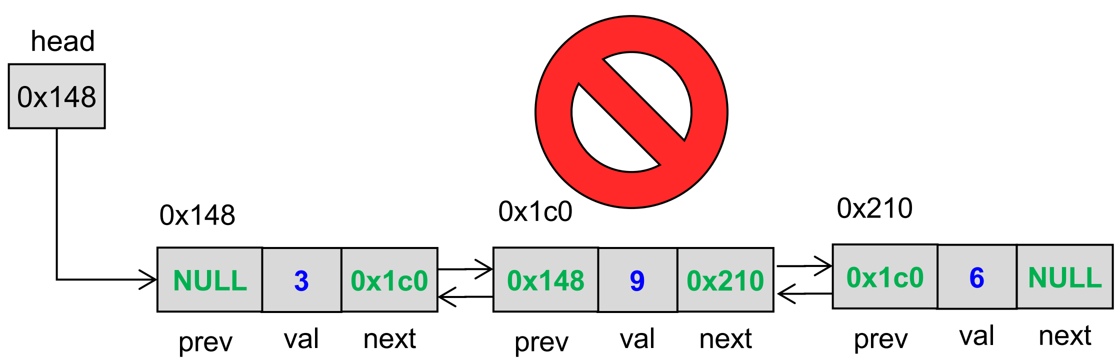
Doubly-Linked List Add Middle

- Adding to the middle requires you to update…
  - Previous item's next field
  - Next item's previous field
  - New item's next field
  - New item's previous field



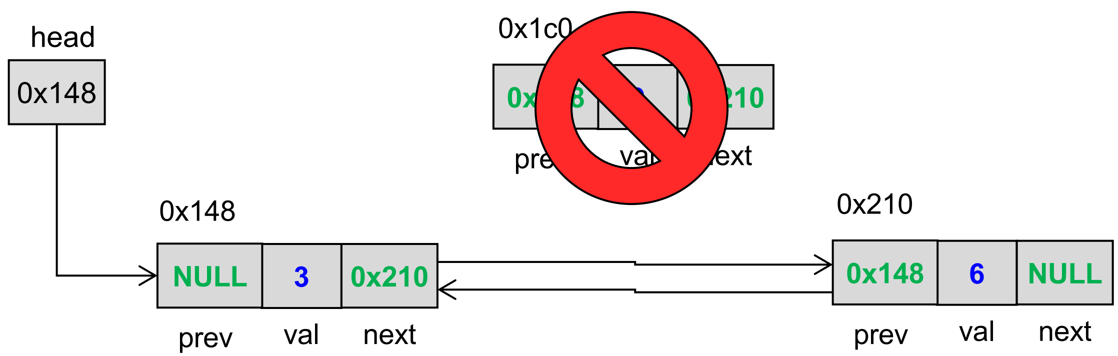
Doubly-Linked List Remove Middle

- Removing from the middle requires you to update…
  - Previous item's next field
  - Next item's previous field
  - Delete the item object



Doubly-Linked List Remove Middle

- Removing from the middle requires you to update…
  - Previous item's next field
  - Next item's previous field
  - **Delete the item object**



ABSTRACT DATA TYPE (ADT)

Abstract Data Types

- DAPS defines an **abstract data type**, or ADT, as:
  - Specification/model for a group of values/data and the operations on those values
- The model allows us to separate…
  - The decision of what data structure to use and how it will be used in our higher level application
  - And the implementation of the specific data structure
- DAPS defines a **data structure** as:
  - An implementation of an ADT in a given programming language
- Each ADT we will examine in this course has certain:
  - Well defined operations and capabilities that are often useful
  - Time & space advantages
  - Time & space disadvantages
- You need to know those operations, advantages and disadvantages

Data Abstraction & Problem Solving with C++, Carrano and Henry will henceforth be abbreviated as DAPS

3 Popular ADTs

- List
- Dictionary/Map
- Set
- (Possible 4th: Priority Queue)

Lists

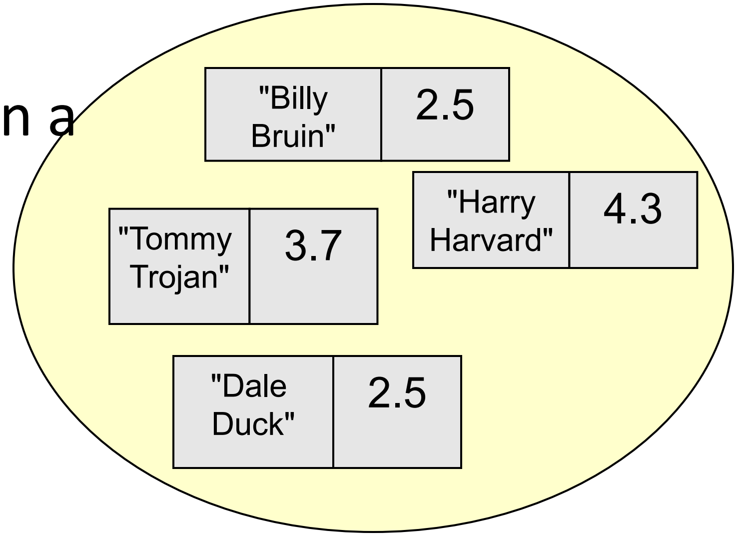
- Ordered collection of items, which may contain duplicate values, usually accessed based on their position (index)
  - Ordered = Each item has an index and there is a front and back (start and end)
  - Duplicates allowed (i.e. in a list of integers, the value 0 could appear multiple times)
  - Accessed based on their position (list[0], list[1], etc.)
- What are some operations you perform on a list?

List Operations

Operation	Description	Input(s)	Output(s)
insert	Add a new value at a particular location shifting others back	Index : int Value	
remove	Remove value at the given location	Index : int	Value at location
get / at	Get value at given location	Index : int	Value at location
set	Changes the value at a given location	Index : int Value	
empty	Returns true if there are no values in the list		bool
size	Returns the number of values in the list		int
push_back / append	Add a new value to the end of the list	Value	
find	Return the location of a given value	Value	Int : Index

Maps / Dictionaries

- Stores <key, value> pairs
  - Example: Map student names to their GPA
- Keys must be unique (can only occur once in the structure)
- No constraints on the values
- What operations do you perform on a map/dictionary?
- No inherent ordering between key,value pairs
  - Can't ask for the 0th item…

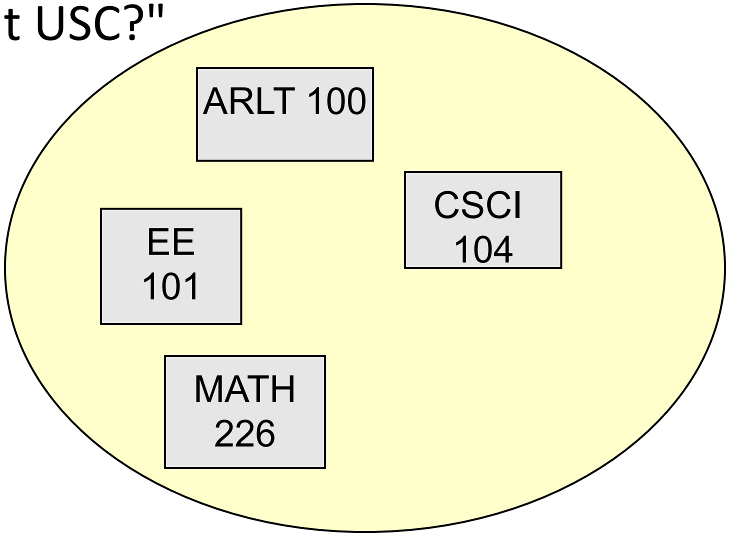


Map / Dictionary Operations

Operation	Description	Input(s)	Output(s)
Insert / add	Add a new key,value pair to the dictionary (assuming its not there already)	Key, Value	
Remove	Remove the key,value pair with the given key	Key	
Get / lookup	Lookup the value associated with the given key or indicate the key,value pair doesn't exist	Key	Value associated with the key
In / Find	Check if the given key is present in the map	Key	bool (or ptr to pair/NULL)
empty	Returns true if there are no values in the list		bool
size	Returns the number of values in the list		int

Set

- A set is a dictionary where we only store keys (no associated values)
  - Example: All the courses taught at USC (ARLT 100, …, CSCI 104, MATH 226, …)
- Items (a.k.a. Keys) must be unique
  - No duplicate keys (only one occurrence)
- Not accessed based on index but on value
  - We wouldn't say, "What is the 0th course at USC?"
- In DAPS textbook Chapter 1, this is the 'bag' ADT
- What operations do we perform on a set?



Set Operations

Operation	Description	Input(s)	Output(s)
Insert / add	Add a new key to the set (assuming its not there already)	Key	
Remove	Remove	Key	
In / Find	Check if the given key is present in the map	Key	bool (or ptr to item/NULL)
empty	Returns true if there are no values in the list		bool
size	Returns the number of values in the list		Int
intersection	Returns a new set with the common elements of the two input sets	Set1, Set2	New set with all elements that appear in both set1 and set2
union	Returns a new set with all the items that appear in either set	Set1, Set2	New set with all elements that appear in either set1 and set2
difference	Returns a set with all items that are just in set1 but not set2	Set1, Set2	New set with only the items in set1 that are not in set2

CLASSES

C Structs

- Needed a way to group values that are related, but have different data types
- NOTE: struct has changed in C++!
  - C
    - Only data members
    - Some declaration nuances
  - C++
    - Like a class (data + member functions)
    - Default access is public

```
struct Person{
  char name[20];
  int age;
};

int main()
{
  // Anyone can modify
  // b/c members are public
  Person p1;
  p1.age = -34;
  // probably not correct

  return 0;
}
```

Classes & OO Ideas

- In object-oriented programming languages (C++) classes are used as the primary way to organize code
- Encapsulation
  - Place data and operations on data into one code unit
  - Keep state hidden/separate from other programmers via private members
- Abstraction
  - Depend only on an interface!
    - Ex. a microwave…Do you know how it works? But can you use it?
  - Hide implementation details to create low degree of ***coupling*** between different components
- Polymorphism & Inheritance
  - More on this later…

```
struct Machine{
  Piece* pieces;
  Engine* engine;
};

int main()
{
  Machine m;

  init_subsystemA(&m);

  change_subsystemB(&m);

  replace_subsystemC(&m);

  m.start();
  // Seg. Fault!! Why?
}
```

Protect yourself from users & protect your users from themselves

Coupling

- Coupling refers to how much components depend on each other's implementation details (i.e. how much work it is to remove one component and drop in a new implementation of it)
  - Placing a new battery in your car vs. a new engine
  - Adding a USB device vs. a new video card to your laptop
- OO Design seeks to reduce coupling as much as possible by
  - Creating well-defined interfaces to change (write) or access (read) the state of an object
  - Enforcing those interfaces are adhered to
    - Private vs. public
  - Allow alternate implementations that may be more appropriate for different cases

C++ Classes

- A composition mechanism
  - Create really large and powerful software systems from tiny components
  - Split things up into manageable pieces
    - Somewhat of a bottom up approach (define little pieces that can be used to compose larger pieces)
  - Delegation of responsibility
- An abstraction and encapsulation mechanism
  - Make functionality publicly available, but hide data & implementation details
- A mechanism for polymorphism
  - More on this later

C++ Classes: Overview

- What are the main parts of a class?
  - Member variables
    - What data must be stored?
  - Constructor(s)
    - How do you build an instance?
  - Member functions
    - How does the user need to interact with the stored data?
  - Destructor
    - How do you clean up after an instance?

C++ Classes: Overview

- Member data can be public or private (for now)
  - Defaults is private (only class functions can access)
  - Must explicitly declare something public
- Most common C++ operators will not work by default (e.g. ==, +, <<, >>, etc.)
  - You can't cout an object (`cout << myobject;` won't work)
  - The only one you get for free is '=' and even that may not work the way you want (more on this soon)
- Classes may be used just like any other data type (e.g. int)
  - Get pointers/references to them
  - Pass them to functions (by copy, reference or pointer)
  - Dynamically allocate them
  - Return them from functions

this Pointer

- How do member functions know which object's data to be operating on?
- d1 is implicitly passed via a special pointer call the 'this' pointer

0x7e0 — d2

cards[52]	37	21	4	9	16	43	20	39
top_index	0							

0x2a0 — d1

cards[52]	41	27	8	39	25	4	11	17
top_index	1							

poker.cpp

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

int main(int argc, char *argv[]) {
  Deck d1, d2;
  d1.shuffle();
```

d1 is implicitly passed to shuffle()

this

0x2a0

deck.cpp

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

void Deck::shuffle()
{
  cut(); // calls cut()
         // for this object
  for(i=0; i < 52; i++){
    int r = rand() % (52-i);
    int temp = cards[r];
    cards[r] = cards[i];
    cards[i] = temp;
  }
}
```

Actual code you write

deck.cpp

```
int main() { Deck d1;
 d1.shuffle();
}
void Deck::shuffle(Deck *this)
{
  this->cut(); // calls cut()
               // for this object
  for(i=0; i < 52; i++){
    int r = rand() % (52-i);
    int temp = this->cards[r];
    this->cards[r] = this->cards[i];
    this->cards[i] = temp;
  }
}
```

Compiler-generated code

Exercises

- cpp/cs104/classes/this_scope

Another Use of 'this'

- This can be used to resolve scoping issues with similar named variables

```
class Student {
 public:
  Student(string name, int id, double gpa);

   ~Student();  // Destructor
private:
   string name;
   int id;
   double gpa;
};


Student::Student(string name, int id, double gpa)
{ // which is the member and which is the arg?
  name = name; id = id; gpa = gpa;
}


Student::Student(string name, int id, double gpa)
{ // Now it's clear
  this->name = name;
  this->id = id;
  this->gpa = gpa;
}
```

C++ Classes: Constructors

- Called when a class is instantiated
  - C++ won't automatically initialize member variables
  - No return value
- Default Constructor
  - Can have one or none in a class
  - Basic no-argument constructor
  - Has the name ClassName()
  - If class has no constructors, C++ will make a default
    - But it is just an empty constructor (e.g. Item::Item() { })
- Overloaded Constructors
  - Can have zero or more
  - These constructors take in arguments
  - Appropriate version is called based on how many and what type of arguments are passed when a particular object is created
  - If you define a constructor with arguments you ***should* also** define a default constructor

```
class Item
{ int val;
 public:
  Item(); // default const.
  Item(int v); // overloaded
};
```

Identify that Constructor

- Prototype what constructors are being called here
- s1
  - string::string()
    // default constructor
- s2
  - string::string(const char*)
- dat
  - vector<int>::vector<int>(int);

```
#include <string>
#include <vector>
using namespace std;

int main()
{
  string s1;
  string s2("abc");

  vector<int> dat(30);

  return 0;
}
```

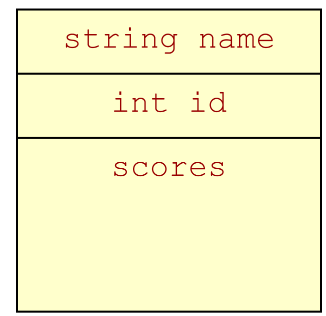
Exercises

- cpp/cs104/classes/constructor_init

Consider this Struct/Class

- Examine this struct/class definition…

```
#include <string>
#include <vector>
using namespace std;

struct Student
{
  string name;
  int id;
  vector<double> scores;
   // say I want 10 test scores per student

};

int main()
{
  Student s1;
}
```



Composite Objects

- Fun Fact: Memory for an object comes alive before the code for the constructor starts at the first curly brace '{'

```
#include <string>
#include <vector>
using namespace std;

struct Student
{
  string name;
  int id;
  vector<double> scores;
   // say I want 10 test scores per student

  Student() /* mem allocated here */
  {
    // Can I do this to init. members?
    name("Tommy Trojan");
    id = 12313;
    scores(10);
  }
};

int main()
{
  Student s1;
}
```

string name
int id
scores

Composite Objects

- You cannot call constructors on data members once the constructor has started (i.e. passed the open curly '{')
  - So what can we do??? Use initialization lists!

```
#include <string>
#include <vector>
using namespace std;

struct Student
{
  string name;
  int id;
  vector<double> scores;
   // say I want 10 test scores per student

  Student() /* mem allocated here */
  {
    // Can I do this to init. members?
    name("Tommy Trojan");
    id = 12313;
    scores(10);
  }
};

int main()
{
  Student s1;
}
```

This would be "constructing" name twice. It's too late to do it in the {…}

string name
int id
scores

Constructor Initialization Lists

```
Student::Student()
{
  name = "Tommy Trojan";
  id = 12313
  scores.resize(10);
}
```

If you write this…

```
Student::Student() :
    name(), id(), scores()
    // calls to default constructors
{
  name = "Tommy Trojan";
  id = 12313
  scores.resize(10);
}
```

The compiler will still generate this.

- Though you do not see it, realize that the **default constructors** are implicitly called for each data member before entering the {…}
- You can then assign values but this is a **2-step** process

Constructor Initialization Lists

```
Student:: Student() /* mem allocated here */
{
  name("Tommy Trojan");
  id = 12313;
  scores(10);
}
```

You can't call member constructors in the {…}

```
Student::Student() :
    name("Tommy"), id(12313), scores(10)
{   }
```

You would have to call the member constructors in the initialization list context

- Rather than writing many assignment statements we can use a special initialization list technique for C++ constructors
  - Constructor(param_list) : member1(param/val), …, memberN(param/val) { … }
- We are really calling the respective constructors for each data member

Constructor Initialization Lists

```
Student::Student()
{
  name = "Tommy Trojan";
  id = 12313
  scores.resize(10);
}
```

```
Student::Student() :
    name(), id(), scores()
    // calls to default constructors
{
  name = "Tommy Trojan";
  id = 12313
  scores.resize(10);
}
```

You can still assign data members in the {…}

But any member not in the initialization list will have its default constructor invoked before the {…}

- You can still assign values in the constructor but realize that the **default constructors** will have been called already
- So generally if you know what value you want to assign a data member it's **good practice** to do it in the initialization list

Exercises

- cpp/cs104/classes/constructor_init2

Member Functions

- Have access to all member variables of class
- **Use "const" keyword if it won't change member data**
- Normal member access uses dot (.) operator
- Pointer member access uses arrow (->) operator

```
class Item
{ int val;
 public:
  void foo();
  void bar() const;
};

void Item::foo() // not Foo()
{ val = 5; }

void Item::bar() const
{ }

int main()
{
  Item x;
  x.foo();
  Item *y = &x;
  (*y).bar();
  y->bar();  // equivalent
  return 0;
}
```

Exercises

- cpp/cs104/classes/const_members
- cpp/cs104/classes/const_members2
- cpp/cs104/classes/const_return

C++ Classes: Destructors

- Called when a class goes out of scope or is freed from the heap (by “delete”)
- Why use it?
  - Not necessary in simple cases
  - Clean up resources that won't go away automatically (e.g. stuff you used “new” to create in your class member functions or constructors
- Destructor
  - Has the name ~ClassName()
  - Can have one or none
  - No return value
  - Destructor (without you writing any code) will automatically call destructor of any data member objects…but NOT what data members point to!
    - You only need to define a destructor if you need to do more than that (i.e. if you need to release resources, close files, deallocate what pointers are point to, etc.)

C++ Classes: Other Notes

- Classes are generally split across two files
  - ClassName.h – Contains interface description
  - ClassName.cpp – Contains implementation details
- Make sure you remember to prevent multiple inclusion errors with your header file by using #ifndef, #define, and #endif

```
#ifndef CLASSNAME_H
#define CLASSNAME_H
class ClassName { … };

#endif
```

```
#ifndef ITEM_H
#define ITEM_H
class Item
{ int val;
 public:
  void foo();
  void bar() const;
};
#endif
```

item.h

```
#include "item.h"
void Item::foo()
{ val = 5; }

void Item::bar() const
{ }
```

item.cpp

CONDITIONAL COMPILATION

Multiple Inclusion

- Often separate files may #include's of the same header file
- This may cause compiling errors when a duplicate declaration is encountered
  - See example
- Would like a way to include only once and if another attempt to include is encountered, ignore it

```
class string{
... };
```

string.h

```
#include "string.h"
class Widget{
 public:
  string s;
};
```

widget.h

```
#include "string.h"
#include "widget.h"
int main()
{ }
```

main.cpp

```
class string { // inc. from string.h
};

class string{ // inc. from widget.h
};
class Widget{
... }
int main()
{ }
```

main.cpp after preprocessing

Conditional Compiler Directives

- Compiler directives start with '#'
  - #define XXX
    - Sets a flag named XXX in the compiler
  - #ifdef, #ifndef XXX … #endif
    - Continue compiling code below until #endif, if XXX is (is not) defined
- Encapsulate header declarations inside a
  - #ifndef XX
    #define XX
    …
    #endif

```
#ifndef STRING_H
#define STRING_H
class string{

... };
#endif
```

String.h

```
#include "string.h"
class Widget{
 public:
  string s;
};
```

Character.h

```
#include "string.h"
#include "string.h"
```

main.cpp

```
class string{ // inc. from string.h
};

class Widget{ // inc. from widget.h

...
```

main.cpp after preprocessing

Conditional Compilation

- Often used to compile additional DEBUG code
  - Place code that is only needed for debugging and that you would not want to execute in a release version
- Place code in a #ifdef XX...#endif bracket
- Compiler will only compile if a #define XX is found
- Can specify #define in:
  - source code
  - At compiler command line with (-Dxx) flag
    - g++ -o stuff –DDEGUG stuff.cpp

```
int main()
{
  int x, sum=0, data[10];
  ...
  for(int i=0; i < 10; i++){
    sum += data[i];
#ifdef DEBUG
    cout << "Current sum is ";
    cout << sum << endl;
#endif
  }

  cout << "Total sum is ";
  cout << sum << endl;
```

stuff.cpp

```
$ g++ -o stuff -DDEBUG stuff.cpp
```

Example Code

- Login to your VM, start a terminal
- Download this example
  - Create an 'lecture_code' directory
  - $ wget http://ee.usc.edu/~redekopp/ee355/code/coninit.cpp
  - $ make coninit