Function pointer

void (*f)(int);
Named f, takes one parameter which is an int. Won't be on the midterm
f = &squared; (where squared is the name of a function that takes one int)
f(10);

Dynamic Memory Allocation
- Sometimes you won't know how many variables you'll need until your program runs
- Can dynamically ask the operating system to reserve new memory for variables
- Operating system will allocate room for your variable in the computer's free memory and then return the address of the new variable

For an array:
We want to define an array, don't know how big to make it
New command can be used to allocate an arbitrary amount of memory for an array

int *arr;
Int size;
Cin >> size;
arr = new int[size];
delete [] arr; -- don't forget brackets when deleting an array
When you call delete, you are deleting the memory that has been allocated, not the pointer itself

If you have an array of pointers to objects, you need to do a loop to delete every single thing in that array

Copy Construction
- Required in all nontrivial C++ programs
- Define a constructor that accepts another of the same class as a parameter → acceptable
- Every variable of the same type can access the privates of every other of the same type of variable
- You can do Circ b = a; which does the same thing; calls a copy constructor, simpler but ugly
- Otherwise you do Circ(const Circ &old) {}
- Circ b = a is bad. Shallow copy causes problems when you destruct either copy
- Just define your own copy constructor

Assignment Operators

- Will be on the exam
- Required
- If you fail to use them properly, it can result in nasty bugs and crashes
- Change the value of an existing variable to the value of another variable
- Function name is operator=, the return type is a reference
- E.g. Circ &operator=(const Circ &src) {}
- Will call the function if you do bar=foo and call the operator thing and put foo as src

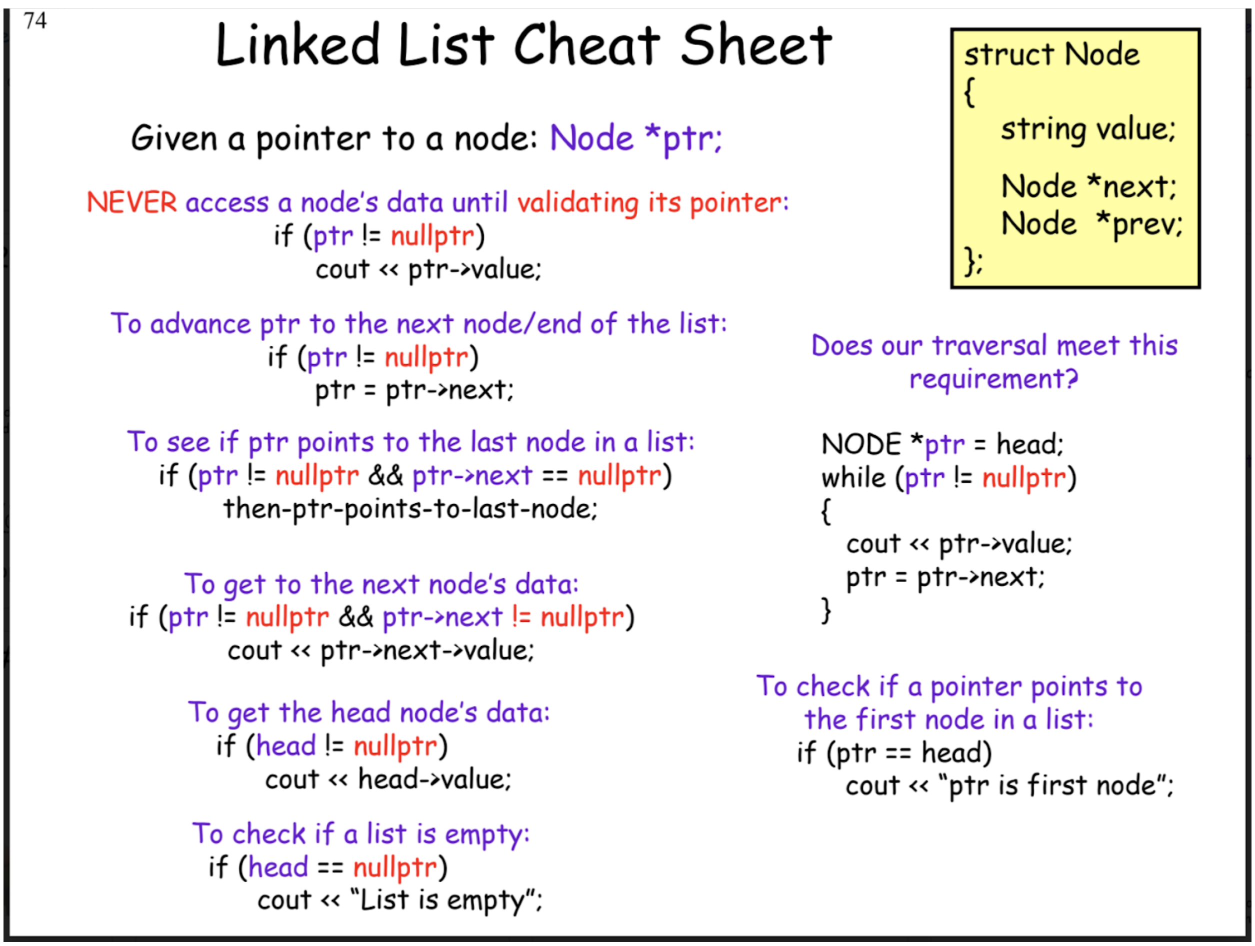
Linked lists
- Super important
- Arrays aren't always great
-

Boelter 2444 Thursday 1/22 6-8
Hackathon on friday

Linked Lists
- Have a head pointer
- That points to the next node
- Which points to the next node
- Etc
- Adding to front
    - Allocate a new node
    - Put value in the node
    - Link the new node to the old top node
        - Node *p;
        - p=new Node;
        - p->value=v;
        - p->next=head
        - head=p
- Adding to rear
    - 2 cases
        - List is totally empty
            - Just add the node
            - If linked list is empty use addToFront code
            - If head==nullptr use addtofront
        - List has stuff

        - Else:
        - Traverse until we find the last node
        - Node *p;
        - p=head;
        - while(p->next != nullptr)
            - p=p->next;
        - Node *n = new Node;
        - n->value=v;
        - p->next=n;
        - n->next=nullptr;
- And more stuff - see slides/handouts

Doubly-linked lists
- Downside of linked list is that you can only look next
- Doubly linked lists have both next and previous



Stacks and Queues
Stack: Last-in first-out

Implementing a stack - this is a simple version

```
const int SIZE=100;
class Stack
{
public:
	Stack() {	m_top = 0;	}
	void push (int val) {
		if(m_top >= SIZE) return; //overflow
		m_stack[m_top] = val;
		m_top++;
	}

	int pop() {
		if(m_top == 0) return -1; //underflow
		m_top--;
		return m_stack[m_top];
	}
private:
	int m_stack[SIZE];
	int m_top; //index
};

int main(void) {
	Stack is;
	int a;
	is.push(5);
	is.push(10);
	a = is.pop();
	cout << a; //10
	is.push(7);
}
```

How to use a stack in C++:

```
#include <iostream>
#include <stack>

int main()
{
	std::stack<int> istack; //stack of ints
	std::stack<string> stackOfStrings; //stack of strings

	istack.push(10);
	istack.push(20);
	cout << istack.top(); //gets top value
	istack.pop(); //stl pop() command simply throws away the top item but doesn’t return it

	if(istack.empty() == false)
		cout << istack.size();
}
```

stack challenge output:

6
n = 6
istack: nothing
push 0
push 12
n = 12
pop 12
0 1 24

Common uses for the stack

- storing undo items for word processor
- evaluating math expressions
- converting from infix to postfix expressions
    - A + B → A B +
- Solve mazes

Every CPU has a built-in stack

Postfix and infix
Evaluating Postfix algorithm:
- inputs: postfix expression string
- output: number representing answer
- private data: a stack
    1. start with the left-most token
    2. if the token is a number:
        a. push it onto the stack
    3. else if the token is an operator:
        a. pop the top value into a variable called v2 and the second-to-top value into v1
        b. apply the operator to v1 and v2 (e.g. v1/v2)
        c. push the result of the operation onto the stack
    4. If there are more tokens, advance to the next token and go back to step 2
    5. After all the tokens have been processed, the top # on the stack is the answer

Infix to postfix conversion
For example: from (3+5) * (4+3/2) -5 to 3 5 + 4 3 2 / + * 5 -
long ass algorithm see slides

Solving a maze with a stack
- starting point onto the stack
- flag point as seen
- check directions, move forward, mark as seen, etc etc until we can't go forward
- then go backwards until can move to an unseen spot
- and then move forwards again etc
- if stack is empty then it's over
- "Depth first search"

QUEUES
Used for:
- optimal route navigation
- streaming video buffering
- flood-filling in paint programs
- searching through mazes
- tracking calls in call centers

Another ADT: The Queue

- another ADT that is just like a line
- first person in line is the first person out of line and served
- enqueue items at the rear and dequeue from the front

Queues and mazes

- breadth first search
- check all directions, add undiscovered stuff to queue and mark as discovered
- then it checks the first item on the stack, and adds to queue, then the next item, etc
- breadth instead of depth
- Guaranteed to find the fastest solution of a maze first

Implementing queues

- can use linked list
- OR an array: a circular queue
    - goes around and around
    - arr, head, tail, count
    - adding stuff: stick it at count and increment tail
    - dequeuing: increment head and decrease count

INHERITANCE

new classes with classes already been defined

subclass and superclass

class Robot {class definition}

class ShieldedRobot **: public** Robot //gets everything from Robot already, only have to declare unique stuff

```
{
public:
	int getShield() { //def }
	void setShield() { //defj }
private:
	int m_shield
}
```

“is a” vs “has a”

- a student is a type of person
- a shielded robot is a type of robot
- A is a type of B → C++ inheritance
- BUT
- a person has a name → a person does not inherit from name

You can inherit more than once
e.g. Person → Student → CompSciStudent

3 Uses of Inheritance

- reuse: write code once in a base class and reuse the same code in your derived classes
- extension: add new behaviors that didn’t exist in the base class
- specialization: redefine an existing behavior from the base class with a new behavior in your derived class

ONLY PUBLIC MEMBERS ARE REUSED IN SUBCLASS. PRIVATE MEMBERS ARE HIDDEN FROM SUBCLASS

protected: //methods that you want subclass to reuse it within the class, but don’t want the program to use them

```
    void chargeBattery();
```

but can’t say Robot r;
r.chargeBattery(); //fails because protected

Don’t do protected variables tho

Specialization/overriding:

- use the keyword virtual to override

```
class Student {
public:
    virtual void WhatDoISay() { cout << “hi”; }
}

class NerdyStudent : public Student {
public:
    virtual void WhatDoISay() { cout << “WHEEE”; }
```

}

if you want to call superclass function with the virtual thing

Student::WhatDoISay();

NerdyStudent s;
s.Student::cheer();

Inheritance and Construction

order of construction and stuff ** see slides
first base class (super), then sub

***construct parents before children

destruction is opposite the order of construction

Polymorphism and stuff

- only works when you use a reference or a pointer to pass an object
- otherwise, something called chopping occurs
    - only the class part in the function's part gets called
    - just a chopped temporary variable
- never pass a derived variable to a function that takes a base variable unless it passes a pointer or reference

class Shape
has virtual double getArea()

class Square : public Shape
Square (int side)
virtual double getArea() { return side * side}

class CIrcle : public Shape
Circle(int rad)
virtual double getrea() {return 3.14 * rad * rad)

Inheritance:
- publicly derive one or more classes from a common base class
- all of the derived classes inherit a common set of functions from the base class
- each derived class may re-define any function originally defined in the base class; the derived class will then have its own specialized version of that function

Polymorphism
- use a base pointer/reference to access to access any variable that is of a type that is derived from our Base class

When to use virtual keyword?
- in base class any time you expect to redefine a function
- etc idk

** ALWAYS USE THE VIRTUAL KEYWORD FOR THE DESTRUCTOR IN YOUR BASE CLASS (and in derived classes for clarity) *** THIS WILL BE ON THE EXAMS

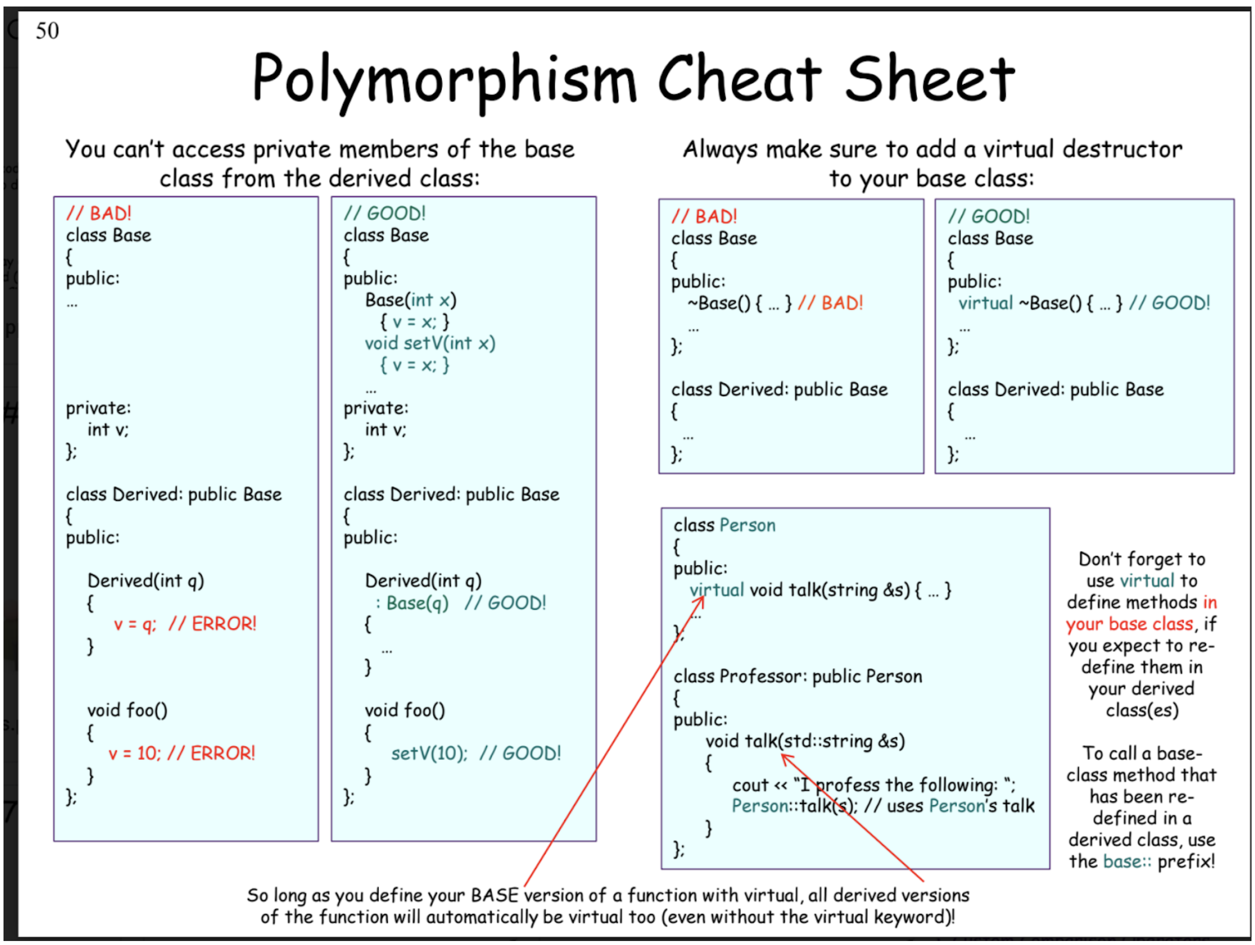
Polymorphism and pointers
- can do something like:
    - Politician carey;
    - Person *p;
    - p = &carey;
- but you can't go the other way
    - a Person *is not* a Politician, but a Politician *is* a Person

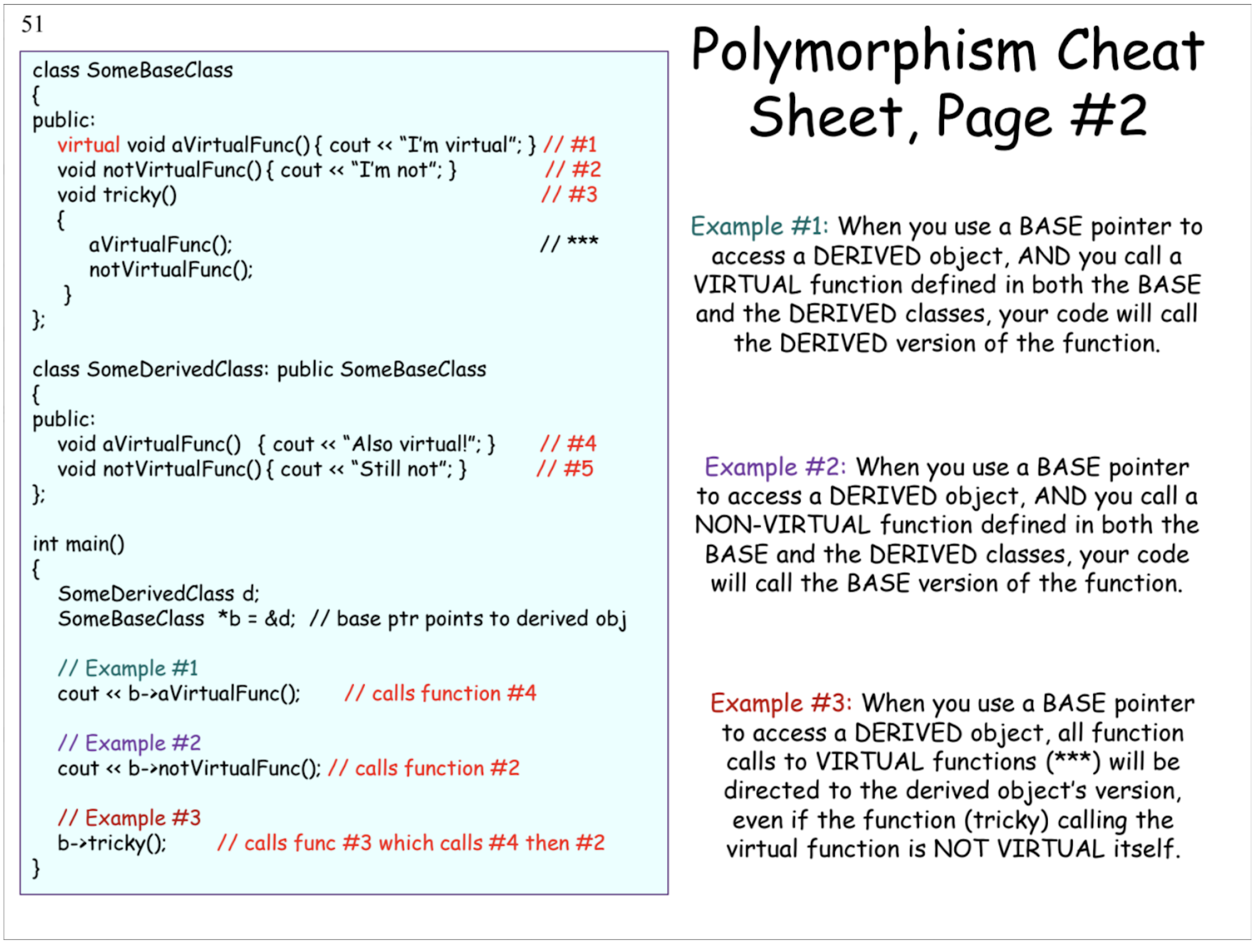
Polymorphism and virtual destructors
- define the virtual destructor first
- virtual ~Prof() P{}
- then virtual ~MathProf() { delete [] m_ptable; }
- need to have virtual destructors in polymorphism destruction otherwise there will be memory leaks
- **ALWAYS USE A VIRTUAL DESTRUCTOR ANYWAYS**

Summary of polymorphism
- first, figure out what we want to represent (like a bunch of shapes)
- then we define a base class that contains functions common to all of the derived classes (e.g. getArea, plotShape)

- Then, we write our derived classes, creating specialized versions of each common function
- we can access derived variables with a base class pointer or reference
- Finally, we should (MUST) always define a virtual destructor in our base class, whether it needs it or not. (no vd in the base class, no points)

Abstract base class:
- contains at least one pure virtual function
- cannot create objects of this class
- if you define an abstract base class, its derived classes must provide code for all pure virtual functions or the class becomes an abstract class itself





Recursion

- difficult and powerful
- the function calls itself over and over again

<><><><><><><><><><><><><><><><><><><><><><><>
Generic Programming
<><><><><><><><><><><><><><><><><><><><><><><>

- build algorithms that can operate on many different types of data
- Allowing generic comparisons

Custom comparison operators
We can do the comparison inside the class with <

```
bool operator<(const Dog &other) const
{
	if(m_weight < other.m_weight)
		return true;
	return false;
}
```

Do >= if you define the comparison outside the class

```
bool operator>=(const Dog &a, const Dog &b)
{
	if(a.getWeight() >= b.getWeight())
		return true;
	return false;
}
```

turn a function into a generic function:

```
template <typename Item>
void swap (Item &a, Item &b)
{
	Item temp;
	temp = a;
	a = b;
	b = temp;
}
```

but you need the assignment operator defined probably

Always place templated functions in a header file

- then include the header file in CPP
- ENTIRE template function in the header file, not just the prototype

- each time you use a templated function with a different type of variable, the compiler generates a new version of the function in your program

If a function has 2 templated parameters with the same data type, you *must* pass in the same type of variable/value for both

You can also override a templated function with a specialized (non-templated) version if you like

You can also have multi-type templates with:

```
template<typename Type1, typename Type2>
void foo(Type1 a, Type2 b)
{
        etc...
}
```

TEMPLATE CHEAT SHEET - GET THAT

21

Carey's Template Cheat Sheet

- To templatize a non-class function called bar:
  - Update the function header: int bar(int a) → template <typename ItemType> ItemType bar(ItemType a);
  - Replace appropriate types in the function to the new ItemType: { int a; float b; … } → {ItemType a; float b; …}
- To templatize a class called foo:
  - Put this in front of the class declaration: class foo { … }; → template <typename ItemType> class foo { … };
  - Update appropriate types in the class to the new ItemType
  - How to update internally-defined methods:
    - For normal methods, just update all types to ItemType: int bar(int a) { … } → ItemType bar(ItemType a) { … }
    - Assignment operator: foo &operator=(const foo &other) → foo<ItemType>& operator=(const foo<ItemType>& other)
    - Copy constructor: foo(const foo &other) → foo(const foo<ItemType> &other)
  - For each externally defined method:
    - For non inline methods: int foo::bar(int a) → template <typename ItemType> ItemType foo<ItemType>::bar(ItemType a)
    - For inline methods: inline int foo::bar(int a) → template <typename ItemType> inline ItemType foo<ItemType>::bar(ItemType a)
    - For copy constructors and assignment operators
    - foo &foo::operator=(const foo &other) → foo<ItemType>& foo<ItemType>::operator=(const foo<ItemType>& other)
    - foo::foo(const foo &other) → foo<ItemType>::foo(const foo<ItemType> &other)
  - If you have an internally defined struct blah in a class: class foo { … struct blah { int val; }; … };
    - Simply replace appropriate internal variables in your struct (e.g., int val;) with your ItemType (e.g., ItemType val;)
  - If an internal method in a class is trying to return an internal struct (or a pointer to an internal struct):
    - You don't need to change the function's declaration at all inside the class declaration; just update variables to your ItemType
  - If an externally-defined method in a class is trying to return an internal struct (or a pointer to an internal struct):
    - Assuming your internal structure is called "blah", update your external function bar definitions as follows:
    - blah foo::bar(…) { … } → template<typename ItemType>typename foo<ItemType>::blah foo<ItemType>::bar(…) { … }
    - blah *foo::bar(…) { … } → template<typename ItemType>typename foo<ItemType>::blah *foo<ItemType>::bar(…) { … }
- Try to pass templated items by const reference if you can (to improve performance):
  - Bad: template <typename ItemType> void foo(ItemType x)
  - Good: template <typename ItemType> void foo(const ItemType &x)

The “STL”

- standard template library
- stacks and queues are part of the STL
- these classes are called “container” classes because they hold groups of items

Some STL Classes

- Vector
    - kinda like an array, but doesn’t have a fixed size
    - sorta like an arraylist

Map Class

- string and an int
- kinda a dictionary
- map<string, int> = name2age;
- name2age[“Carey”] = 30;
- map<string, int>::iterator it;
- it = name2age.find(“Dan”);
- it->first
- it->second
- where the stuff in the map is a string first and int second
- alphabetized automatically

Set:

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

set<int> a;
a.insert(2);
a.insert(3);
a.insert(4);
a.insert(2); //duplicate, doesn’t do anything

a.erase(2);
```

- sets are alphabetized automatically
- sets can be of other data types

Big O
- efficiency of algorithms
- "how fast is that algorithm"

How many iterations in terms of N
Big O: compare algorithms for a given size input

The concept:
- gross number of steps it requires to process an input of size N
- Simple functions like log(n), n, n^2, nlog(n), n^3, etc

e.g. Nested double for loop 0-n, 0-n:
- O(n^2)
- Steps
    - Compute f(n)
    - keep most significant term of the function
    - remove any constant multiplier from the function
    - Now, you have big O

When you have stuff divided by 2, you have log(n)

ugly math stuff
i < n
j < i (replace i with n for max value, so…)
⇒ O(n^2)

x < n
y < x*x (x can be as big as n, so replace x with n, so…)
⇒ O(n^3)

Use both variables if there are 2 variables to depend on

STL and Big O
do Big O under the assumption that the variables are the highest they can be

Selection sort
- select lowest, swap w i

Insertion Sort

- grab last element, insert into proper place

Bubble sort

- swap all the way through

Check STL challenge

Big O of STL things

Quicksort, Mergesort?

- generally work as follows:
    - divide elements into 2 groups
    - sort each group
    - combine the sorted groups
- Recursion

Quicksort algorithm

- if array has 0 or 1 element, return (base case)
- select an arbitrary element P from the array (typically the first element)
- Move all elements less than or equal to P to the left of the array and all elements greater than P to the right (partitioning)
- Usually O(nlogn) unless ordered, in which case it is O(n^2)

Mergesort

- first, know the merge algorithm: takes 2 pre-sorted arrays as inputs and outputs a combined, sorted array
- has to allocate a new array every time
- if array has 1 element, return
- split array into 2 equal
- mergesort left, mergesort right
- merge 2 halves together