Week 5

Announcements

- HW 3 (due Tuesday 5/4)

Questions?

- anything?

Inheritance, Static vs Dynamic Typing

```
// re-use code, make things simpler, fewer bugs

Class Ant         { /* ... */ };
Class Grasshopper { /* ... */ };

Ant         ants[100]
Grasshopper ghs[100]

/*
there's no easy way for me to loop through all of my characters
```

```
I need to write a different loop for each class type, which can get annoying

loop through ants
  ants.talk

loop through grasshoppers
  grasshopper.talk
*/

// instead, we put the common functionality in a superclass, 'Insect'

class Insect: {
  public:
    Insect(string phrase) {
      myPhrase = phrase;
    }

    // all insects talk the same, so this isn't virtual
    // we pass myPhrase to the Insect constructor through our subclasses
    void talk() { std::cout << myPhrase << std::endl; }

    // with virtual, our subclasses are free to re-implement the function if
    // they want to do something different
    virtual void eat() { std::cout << "Eat nothing" << std::endl; };

    // note: don't overwrite a non-virtual function

  private:
    string myPhrase;
}

// now our Ant and Grasshopper classes inherit from class Insect
class Ant: public Insect {

  public:
    // we use insect to handle the phrase storage
    Ant(string phrase) :Insect(phrase) {}

    // we've overwritten eat() so that Ants have different behavior
    virtual void eat() { std::cout << "Do Something" << std::endl; };

  private:

}

class Grasshopper: public Insect {

  public:

    // do it for Grasshopper too
    Grasshopper(string phrase) :Insect(phrase) {}

    // Grasshopper doesn't overwrite eat, so when eat is called on a Grasshopper
    // it will call Insect's implementation
```

```
  private:

}

// and now we can store all of our different classes in the same array
// using the base class
Insect* arr[2]

// polymorphism
arr[0] = new Ant("hi");
arr[1] = new Grasshopper("Hey");

// will this work?
for (i = 0; i < 2; i++)
  arr[i].talk();

// this one?
for (i = 0; i < 2; i++)
  arr[i].eat();

// ***** Inheritance relationship summary *****

// super class, base class are what we inherit FROM
// subclass is what inherits the base/super class

// "is a"  - superclass/baseclass subclass
// "has a" - member variable

// ****************************************
```

Pure Virtual Functions

```
class Shape {
  public:
    // ...
    virtual float getArea() = 0;    // todo: make this ~pure~

  private:
    // ...
};

// will this compile?
// no, because Shape is now an ABC
Shape s;

// ABC - abstract base class - has at least 1 pure virtual function
```

Polymorphism

```
// using a class to simplify our code
class Shape {
  public:
    Shape(int sides) { m_sides = sides; }
    virtual ~Shape();                        // destructor has to be virtual
    int sides() { return m_sides; }
    // ...
  private:
    int m_sides;
}

class Square: public Shape {
  public:
    Square(): Shape(4) {
      m_idk = new int[5];
    }

    // todo: do we need a destructor?
    // yes, because we are handling a resource
    virtual ~Square() {
      delete m_idk [];
    }

  private:
      int* m_idk;
}

// if the argument is not a pointer or a reference, our object will get sliced
// which isn't good
void howManySides(Shape s) {    // should be howManySides(Shape* s)
  cout << s.sides() << endl;
}

// todo: is this ok?
// no, s1 gets sliced
Square s1;
howManySides(s1);

// todo: do our destructors look ok?
// after making ~Shape() virtual, this now works correctly
Shape* s = new Square();
delete s;

// todo: what about this code?
// this is just testing the Square desctructor
Square* sq = new Square();
delete sq;

/*****************************************************/
```

```
class Person {};
class Politician: public Person {};

// Are these valid uses of polymorphism?

// Scenario 1
// YES, assign base class to subclass pointer (definition of polymorphism)
Person *p;
Politician Chris;

p = &Chris;

// Scenario 2
// NO, cannot assign subclass to a base class pointer
Politician *p;
Person Dave;

p = &Dave;

// Inheritance vs Polymorphism
// Inheritance - having classes inherit other classes
// Polymorphism - when you use a base class pointer to point to a subclass object
```

Worksheet #2

```
// did this in the breakout room

#include <iostream>
using namespace std;

class Pet {
  public:
    Pet()  { cout << "Pet" << endl; }
    ~Pet() { cout << "~Pet" << endl; }
};

// contains a Pet as a data member.
class Dog : public Pet {
  public:
    Dog() { cout << "Woof" << endl; }
    ~Dog() { cout << "Dog ran away!" << endl; }
  private:
    Pet buddy;
};

// what does this output?
int main() {
  Pet* milo = new Dog;
```

```
  delete milo;
}
```

Recursion

```
// note: only use local variables, no globals or member vars
// be creative with how you use the return value, like in HW3

// fyi 0! = 1! = 1
// factorial(5) = 5 * 4 * 3 * 2 * 1 = 120

// todo - fill out
int factorial(int n) {

  // base case
  if (n == 0 && n == 1)
    return 1;

  // recursive call ("leap of faith")
  int subProblem = factorial(n-1);

  // do some work
  return n * subProblem;

}

// trace through small example
factorial(4) ---> 4 * 6 = 24
factorial(3) ---> 3 * 2
factorial(2) ---> 2 * 1
factorial(1)  --> 1

// similar example, but we're adding the elements of an array
int a = { 5, 3, 4, 7 }
addArr(a, 4)

int addArr(int* a, int n) {
  // each recursive call deals with the first element of the array passed to it

  // base case
  if (n == 1)
    return a[0];

  int result = addArr(a + 1, n-1);

  // do some work
  // use the result of the subproblem and combine it with the information we have
  // information we have = first element of the array
  return a[0] + result;
```

```
}

[ 5 3 4 7 ] 4 ---> 5 + 14 = 19
  [ 3 4 7 ] 3 ---> 3 + 11
    [ 4 7 ] 2 ---> 4 + 7
      [ 7 ] 1 ---> 7

// divide and conquer
// to sort the array, we keep splitting it in half
// until we reach one element

// Divide
[ 5 3 4 7]         // original array
[ 5 3 ] [ 4 7]
[ 5 ] [ 3 ] [ 4 ] [ 7]

// Conquer
[ 5 ] [ 3 ] [ 4 ] [ 7]
[ 3 5 ] [ 4 7 ]
[ 3 4 5 7]         // result

// pseudocode
array MergeSort(arr) {    // aka lazy person's sort

  // base case
  if (arr.size == 0 or 1)
    return arr

  // recursive call ("leap of faith")
  a1 = MergeSort(arr[: half])
  a2 = MergeSort(arr[half+1 :])

  // do some work (Merge algorithm)
  resultArr []
  take the smaller value: a1[0] a2[0]
  resultArr[nextIndex] = smaller value

  return resultArr
}
```

Palindrome (Recursion #2)

```
bool isPalindrome(string foo) {

  // base case
  // the simplest input is a string of length 1 or 0, which you can
  // immediately tell is a palindrome
  int l = foo.length;
  if(l =< 1)
    return true;
```

```
  // check if the outside two characters match
  if(foo[0] != foo[l-1])
    return false;

  // recursive call
  return isPalindrome(foo.substring(1, l-2));
}

// given examples
isPalindrome("kayak");           // true
isPalindrome("stanley yelnats"); // true
isPalindrome("LAs rock");        // false (but the sentiment is true)

// recursive stack
// if you're stuck, sometimes it helps to write these down
// you can think about how you're approaching your base case,
// what to return from your base case,
// and how to trickle that result back to the original function call

isPalindrome("kayak");
isPalindrome("aya");
isPalindrome("y");      // base case, returns true

isPalindrome("abba");
isPalindrome("bb");
isPalindrome("");       // base case, returns true
```