

Data Structures with C++ : CS189

Lecture 2-1: Polymorphism

Recap

Inheritance

- ISA
 - If you have 2 classes and you can say "a ISA b", then you can connect them
 - Why?
 - Remove duplicate code
 - Easier to split to many programmers
 - Easier to add new leaves without touching base
 - Easier to debug
 - Easier to maintain
 - If you have a bunch of leaf classes, identifying dupes and pulling functionality up is "bottom up"

Polymorphism

- Starting with a class that is the parent of everything and then specializing it is "top down" inheritance
- Bottom up: "I have Dog and Cat classes and both have code for breathing. I'll make an Animal class!"
- Top down: "I have an Animal class, but it is too generic to bark. I'll make Cat and Dog!"
- Both inheritance, just different perspectives

First, why?

- I want to write a program for a Zoo that will track if each animal has been fed
 - I really don't care what kind of animals they are. Just that they are animals
 - What each animal eats is different, so we will want classes still
 - Cow ISA Animal and Elephant ISA Animal
 - I can now get everybody in to one big list
 - STL summary should have been last semester
- ```
class Animal
{};
class Cow : public Animal
{};
class Elephant
 : public Animal
{};
list< Animal * > animals;
```

## And how

- You are a Student. Student ISA Mammal
  - Imagine me pointing at you right now
    - I say "You are a Student"
  - I keep pointing
    - Now I say "You are a Mammal"
  - You didn't magically change type
  - So if I have a dog, two cats, a badger, and a horse, I can say "I have 5 animals"
  - Polymorphism is all about abstracting a group of classes into a base class
    - Last week we were more dealing with the leaves
- Same thing

- Extendibility: If a new animal is made, that loop doesn't change
  - Cohesion: Only general Animal code is in Animal.  
Dove stuff is in Dove
- ```
Animal *tArray[3];
tArray[0] = new Dove();
tArray[1] = new Cat();
tArray[2] = new Monkey();
Hippopotamus tHippo;

for( int i = 0; i < 3; i++ )
    cout << tArray[i]->CanEat(&tHippo);

// Whatever you are, can you eat a hippo?
```

Where Polymorphism Differs

- Last week, even after pulling mammal code out of dog we still had a dog and could call dog methods
- If you only have the baseclass pointer, how can you call anything?
 - Sheep *A; calls Sheep methods
 - Dog *B; calls Dog methods
 - Animal *; ...calls Animal methods? What if I want something in mammal or dog?

Virtual

- I can only see methods at the level I have a pointer to
- Plus, an object always knows what it was originally
- Plus, if you call a method that doesn't exist where you are looking, it tries the next class up
- Actually, these are three huge points.
- I'll pause

SO, Virtual

- Flags a method as "Please look for this at leaf level"
 - Then the "can't find it look upwards" rule takes over
 - Animal has a virtual method and goes to Dog
 - Dog doesn't have it and goes to Mammal
 - Mammal has an override and stops
 - Exact same as last week except we started at the top first and then tried to find the method
- ```
class Animal {
 virtual bool Func();
};

class Mammal
: public Animal {
 virtual bool Func();
};

class Dog
: public Mammal {
};
```

## Virtual Destructor

- Small point that deserves its own slide:
  - If any method in your hierarchy is virtual, the Destructor must be virtual
  - ~ is a method like any other. So if I have an Animal pointer and try to delete the object, it will only delete the Animal part
    - Rest of the object is now memory leak

## Pure Virtual

```
class Animal {
 virtual void Eat() = 0;
};
```

- We have lots of Animal pointers, but don't want Animal objects
- "pure virtual" adds on to the virtual definition
  - Please start at the leaf please... and there's nothing here.
- "Nothing" doesn't mean empty method, it means nothing is there
  - Syntax is "= 0" as if it were wetting the method to null in a way

## Abstract Class

- That Animal class cannot be instantiated
  - "Pure virtual" method makes it an "Abstract Class"
- As long as either Dog or Mammal implements the missing "Eat" method, they can be instantiated
  - Really just "every method must be somewhere"
- Other languages have keywords for this
  - Java has "abstract" if you've seen it

## Anti - Polymorphism

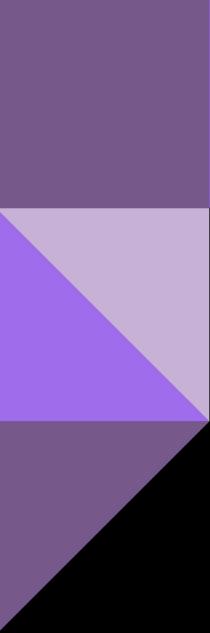
- With regular inheritance, we'd be fine if we only used leaf classes
- Without polymorphism, if I made a Dog, and then referred to it with a Mammal pointer, I could only call Mammal and Animal methods
  - Virtual is what lets you go down to your leaf class

```
class Mammal
: public Animal {
 bool Func();
};
```

```
class Dog
: public Mammal {
 bool Func();
};
Mammal *ptr = new Dog
ptr->Func(); // calls Mammal
```

## Summary

- Basic inheritance is putting methods and properties at the correct level to remove duplication
- Polymorphism uses inheritance and virtual together to enable code to deal only with the baseclass



# End

Tomorrow is just some polish. Start the homework so you can ask questions