CSCI 200: Foundational Programming Concepts & Design Lecture 28

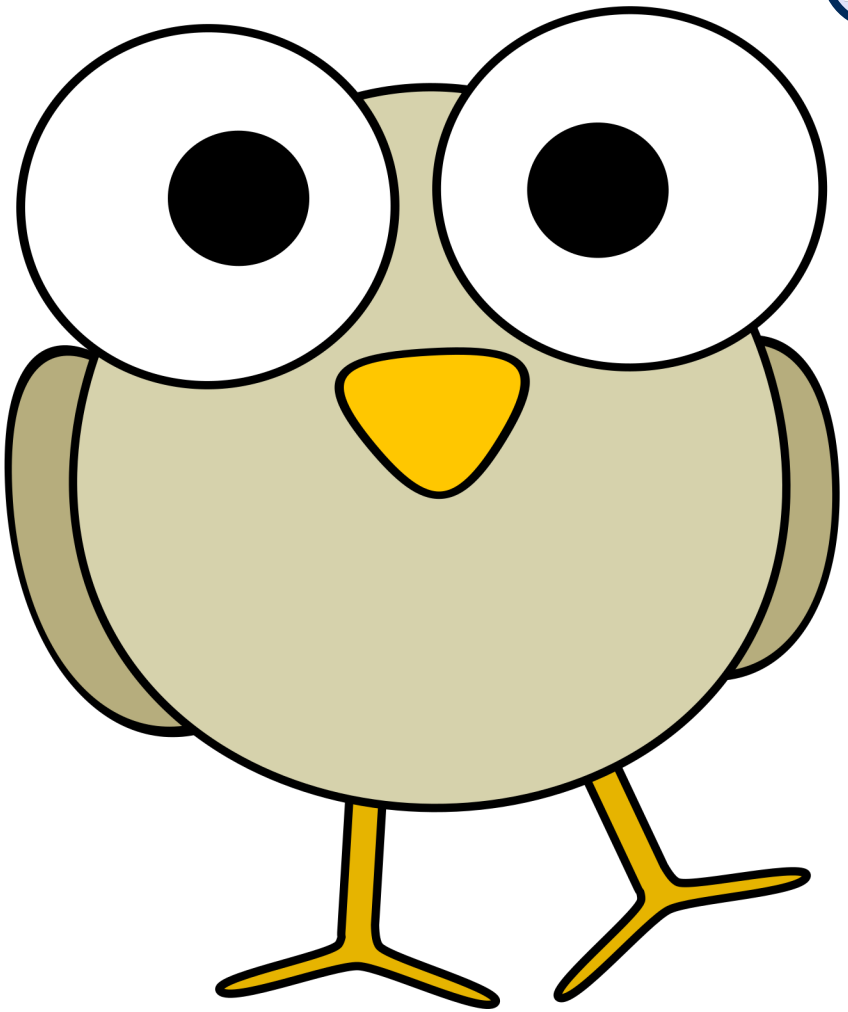
Object-Oriented Programming:

Inheritance & Compile-Time Polymorphism

Previously in CSCI 200

- Inheritance
  - Child class inherits members from Parent class

Questions?



Learning Outcomes For Today

- Discuss the concept of encapsulation
- Discuss what inheritance is and situations it should be used
- Draw a class diagram using UML to describe the structure of a class, its members, and its parents
- Create a child/derived class that inherits data members and member functions from a parent/base class
- Define polymorphism.
- Give examples of polymorphism at compile-time through ad-hoc polymorphism, parametric polymorphism, and subtype polymorphism.
- Discuss the dangers of subtype polymorphism.

On Tap For Today

- Polymorphism
  - Prior Usages
  - Inheritance
- Overriding Functions
- Practice

On Tap For Today

- Polymorphism
  - Prior Usages
  - Inheritance
- Overriding Functions
- Practice

poly·morph·ism

- *poly* – many
- *morph* – form / behavior
- *ism* – imitation of

- *polymorphism:*
  - having many forms
  - having many behaviors

On Tap For Today

- Polymorphism
  - Prior Usages
  - Inheritance
- Overriding Functions
- Practice

Ad-Hoc Polymorphism Example

- Overloaded functions

```
int remainder(int numerator, int denominator) {
   return numerator % denominator;
}


float remainder(float numerator, float denominator) {
   return (numerator / denominator) - (int)(numerator / denominator);
}


// ...
cout << remainder(9, 5) << endl;        // prints 4
cout << remainder(9.0f, 5.0f) << endl; // prints 0.8
```

- **`remainder`** has two forms

Parametric Polymorphism Example

- Templates (Classes and/or Functions)

```
template<typename T>
class LootBox {
public:
  LootBox() { _pLoot = nullptr; }
  void putIn(const T LOOT) { if(_pLoot != nullptr) _pLoot = new T(LOOT); }
  T takeOut() {
    T loot = *_pLoot;
    delete _pLoot; _pLoot = nullptr;
    return loot;
  }
private:
  T* _pLoot;
};

LootBox< string > wordBox;
wordBox.putIn( "polymorphism" );  // put in a string

LootBox< int > countBox;
countBox.putIn( 2 );              // put in an int
```

- **LootBox** has two behaviors

Polymorphism

- Overloaded Functions & Templates

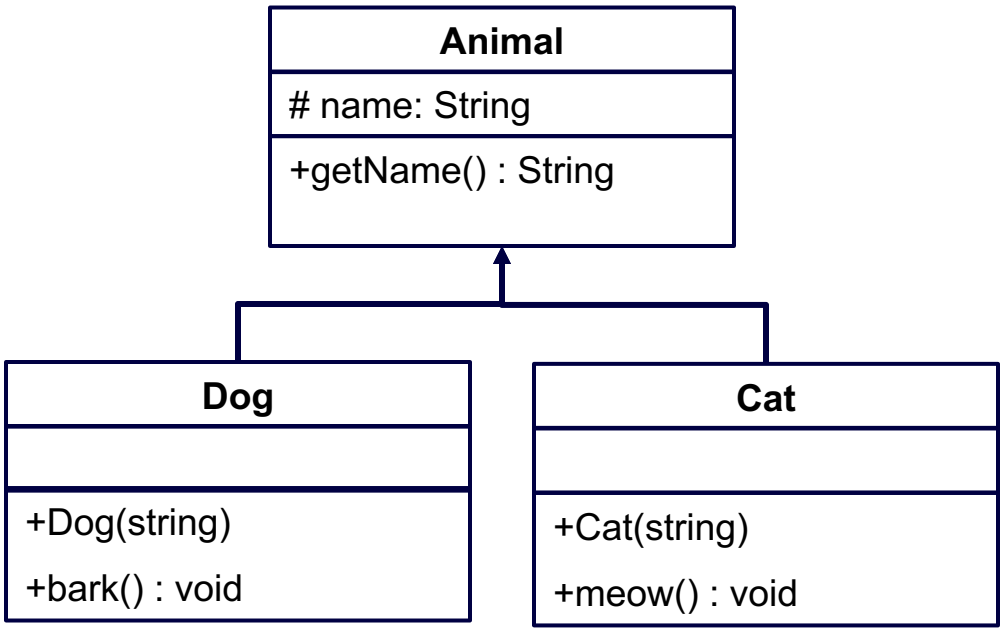
```
cout << remainder(9, 5) << endl;        // prints 4
cout << remainder(9.0f, 5.0f) << endl; // prints 0.8
// ...
wordBox.putIn( "polymorphism" );    // adds a string
countBox.putIn( 2 );                // adds an int
```

- At compile-time, which form to use is known

On Tap For Today

- Polymorphism
  - Prior Usages
  - Inheritance
- Overriding Functions
- Practice

Animal Hierarchy

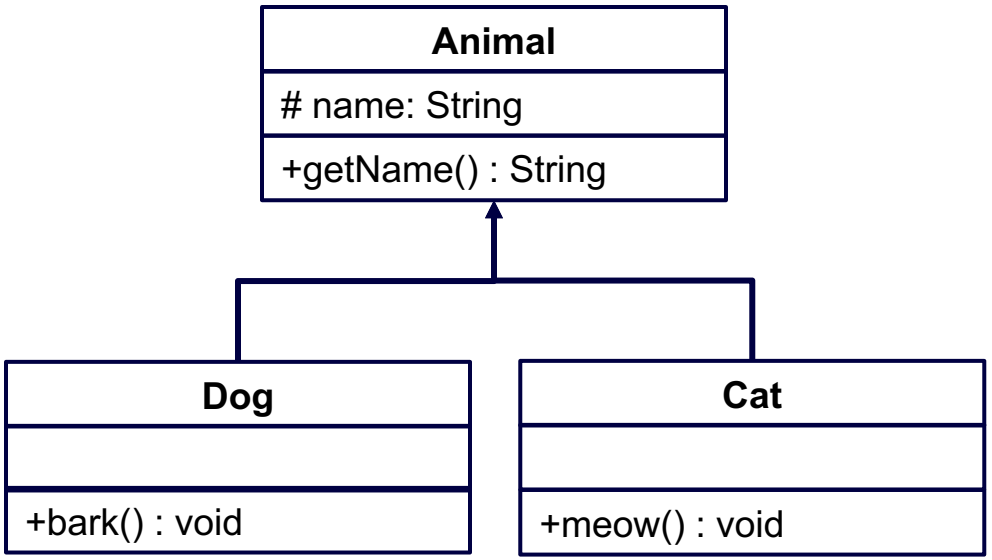


Polymorphism

```
Dog odie;
Cat garfield;

cout << odie.getName() << " ";
odie.bark();

cout << garfield.getName() << " ";
garfield.meow();
```



- odie is a Dog and an Animal
- garfield is a Cat and an Animal
  - Can exhibit behaviors of different types

Subtype Polymorphism

```
Dog odie;
cout << odie.getName() << " ";  // treat odie as an Animal
odie.bark();                    // treat odie as a Dog
```

- odie is a Dog and an Animal
  - Can exhibit behaviors of different types

- At compile-time, form & behavior is known

On Tap For Today

- Polymorphism
  - Prior Usages
  - Inheritance
- Overriding Functions
- Practice

Behavioral Similarities

```
class Dog : public Animal {
public:
  Dog() { cout << "Creating a dog" << endl; }
  ~Dog() { cout << "Destroying a dog" << endl; }
  void bark() const { cout << "Woof" << endl; }
private:
};
class Cat : public Animal {
public:
  Cat() { cout << "Creating a cat" << endl; }
  ~Cat() { cout << "Destroying a cat" << endl; }
  void meow() const { cout << "Meow" << endl; }
private:
};
```

Using The Classes

```
int main() {
  Animal anAnimal;  anAnimal.setName( "John" );
  Dog odie;         odie.setName( "Odie" );
  Cat garfield;     garfield.setName( "Garfield" );

  cout << "Animal " << anAnimal.getName() << " can't speak" << endl;

  cout << "Dog " << odie.getName() << " says ";
  dog.bark();

  cout << "Cat " << garfield.getName() << " says ";
  garfield.meow();

  return 0;
}
```

Overloading Functions

- Overloaded functions
  - Multiple functions have same identifier but different parameters

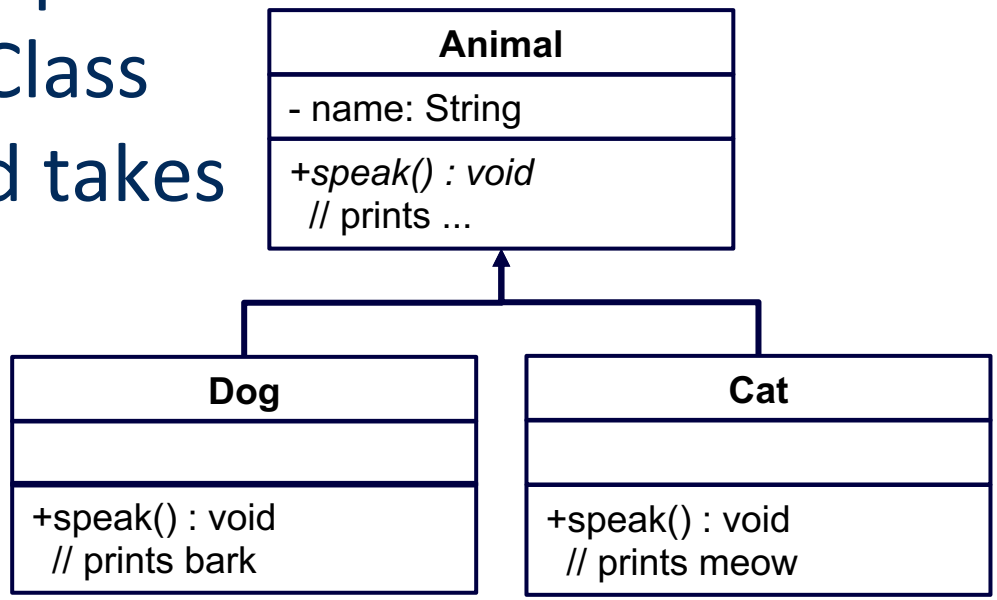
```
int remainder(int numerator, int denominator) {
   return numerator % denominator;
}


float remainder(float numerator, float denominator) {
   return (numerator / denominator) - (int)(numerator / denominator);
}


// ...
cout << remainder(9, 5) << endl;        // prints 4
cout << remainder(9.0f, 5.0f) << endl; // prints 0.8
```

Overriding Functions

- Overridden functions
  - Derived Class has member function with same function name and signature as Base Class
- The Derived Class implementation overrides the Base Class implementation and takes precedence
  - Resolve bottom-up



Virtual Functions

- Virtual Functions act as “function pointer”
  - Provide a default implementation
  - Subtype can override parent implementation

```
class Animal {
public:
  virtual void speak() const  { cout << "..." << endl;  }
};
class Dog : public Animal {
public:
  void speak() const override { cout << "bark" << endl; }
};
class Cat : public Animal {
public:
  void speak() const override { cout << "meow" << endl; }
};
```

  - **`override`** keyword checks that function signatures match AND parent function is **`virtual`** – can be overridden

First-Class

- First-Class in programming:
  - Any entity supports all operational properties inherent to other entities
  - Properties such as:
    - Created/deleted
    - Assigned
    - Tested for equality
    - Passed around as a function argument
    - Returned from a function
    - And others

First-Class Objects

- Variables!

```
T var1, var2, var3;
var2 = var1;
if(var1 == var3) {...}
void f(T param) {...}
T g() { T var; return var; }
```

Template Type T

- What data types do we have?
  - `int, bool, char, float, double`
  - `Pointers`
  - `string, vector`
  - `class`
  - Functions…

Functions

- What happens when you run

```
int f() { return 1; }
int main() {
  cout << f() << endl;
  cout << (void*)f << endl;
  return 0;
}
```

Template Type T

- What data types do we have?
  - `int, bool, char, float, double`
  - `Pointers`
  - `string, vector`
  - `struct`
  - `class`
  - Functions...as pointers

    --> Function Pointers

Function Pointer

- Function definition

```
int add(int x, int y) { return x + y; }
```

- Function declaration

```
int add(int, int);
```

- Function pointer (yes it's ugly)

```
int (*pfMyFunc)(int, int) = &add;
```

- Invoke Function pointer

```
(*pfMyFunc)(2, 3)// explicit
pfMyFunc(2, 3)   // implicit
```

Precedence	Operator	Associativity
1	Parenthesis: ( )	Innermost First
2	Scope Resolution: S::	Left to Right
3	Postfix Unary Operators: a++ a-- a[ ] a. f( ) p->	
4	Prefix Unary Operators: ++a --a +a -a !a (type)a &a *p new delete	Right to Left
5	Binary Operators: a*b a/b a%b	Left to Right
6	Binary Operators: a+b a-b	
7	Relational Operators: a<b a>b a<=b a>=b	
8	Relational Operators: a==b a!=b	
9	Logical Operators: a&&b	
10	Logical Operators: a||b	
11	Assignment Operators: a=b a+=b a-=b a*=b a/=b a%=b	Right to Left

Function Pointer Example

```
int f(int x) { return x + 1; }
int g(int x) { return x + 5; }

int main() {
  int (*pfMyFunction)(int) = nullptr;

  pfMyFunction = &f;
  cout << pfMyFunction(1) << endl; // prints 2

  pfMyFunction = &g;
  cout << pfMyFunction(1) << endl; // prints 6

  return 0;
}
```

Using The Classes

```
int main() {
  Animal anAnimal;  anAnimal.setName( "John" );
  Dog odie;         odie.setName( "Odie" );
  Cat garfield;     garfield.setName( "Garfield" );

  cout << "Animal " << anAnimal.getName() << " says ";
  anAnimal.speak(); // prints ...

  cout << "Dog " << odie.getName() << " says ";
  odie.speak();     // prints bark

  cout << "Cat " << garfield.getName() << " says ";
  garfield.speak(); // prints meow

  return 0;
}
```

Overridden Functions

```
class Animal {
public:
  virtual void speak() { cout << "..." << endl; }
};
class Dog : public Animal {
public:
  void speak() override { cout << "bark" << endl; }
};
// ...
Dog odie;
odie.speak();             // prints bark -- odie is a Dog
((Animal)odie).speak(); // prints ...  -- odie is an Animal
odie.Animal::speak();   // prints ...  -- odie is a Dog, explicitly call Animal
odie.Dog::speak();      // prints bark -- odie is a Dog, explicitly call Dog
```

- To call a specific form, either
  - Cast object type
  - Use scope resolution

Polymorphism In Action + Concerns

```
void hearAnimal(Animal animal) {
  animal.speak();
}

void hearDog(Dog dog) {
  dog.speak();
}
// ...
Dog odie;
hearDog(odie);            // prints bark -- odie is a Dog
hearAnimal(odie);         // prints ...  -- odie is an Animal
```

More Polymorphism Concerns

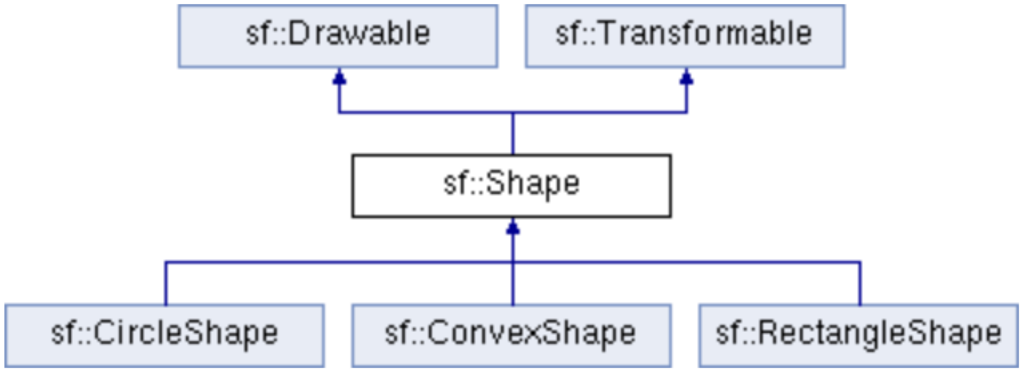
```
void hearAnimal(Animal animal) {
  animal.speak();
}

void hearDog(Dog dog) {
  dog.speak();
}
// ...
Cat garfield;
hearDog(garfield);            // error! -- garfield is a Cat, not a Dog
hearAnimal(garfield);         // prints ...  -- garfield is an Animal
```

- Class Cast Error → Compiler Error!
- Polymorphism checked at compile-time

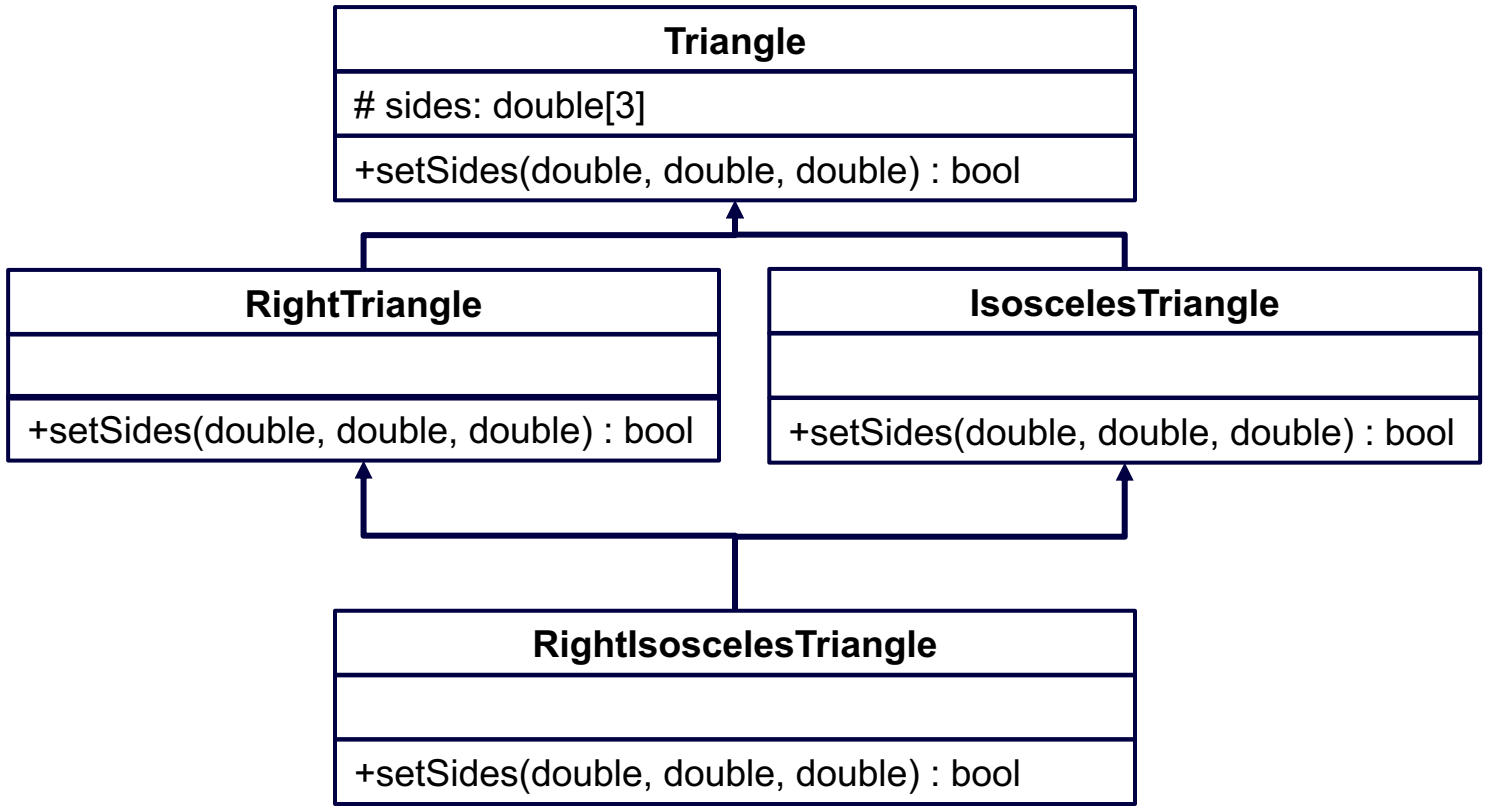
Multiple Inheritance

- Shape is both Drawable & Transformable
  - "Multiple Inheritance"
- Polymorphism in action!!



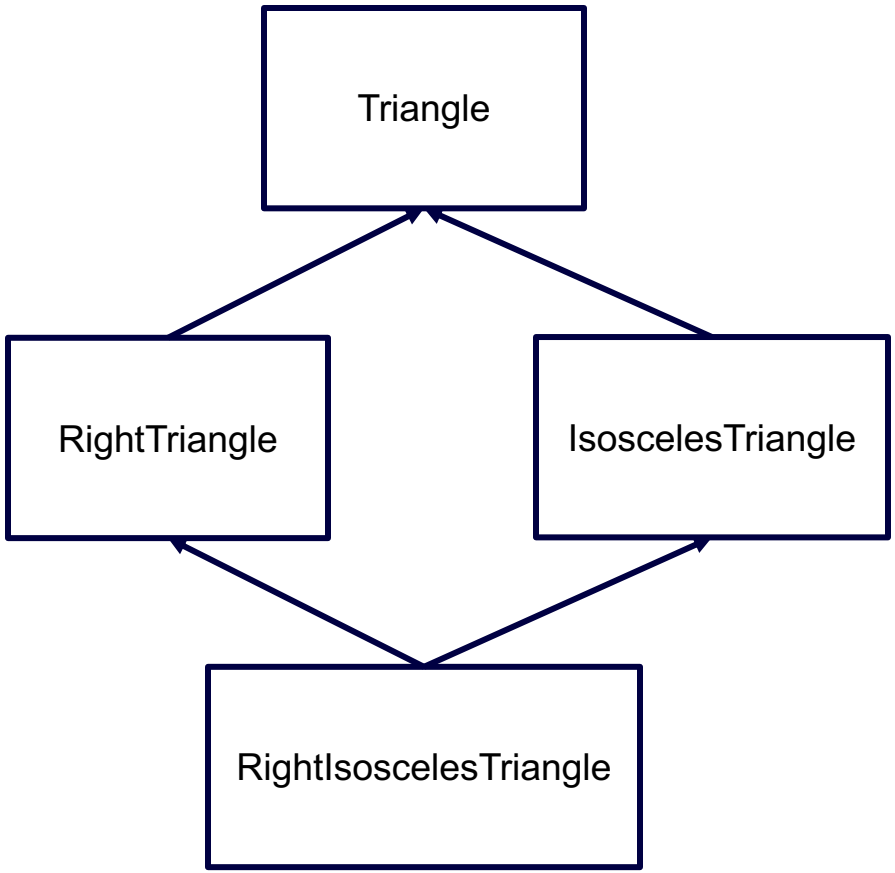
Multiple Inheritance Concern

- Diamond Problem



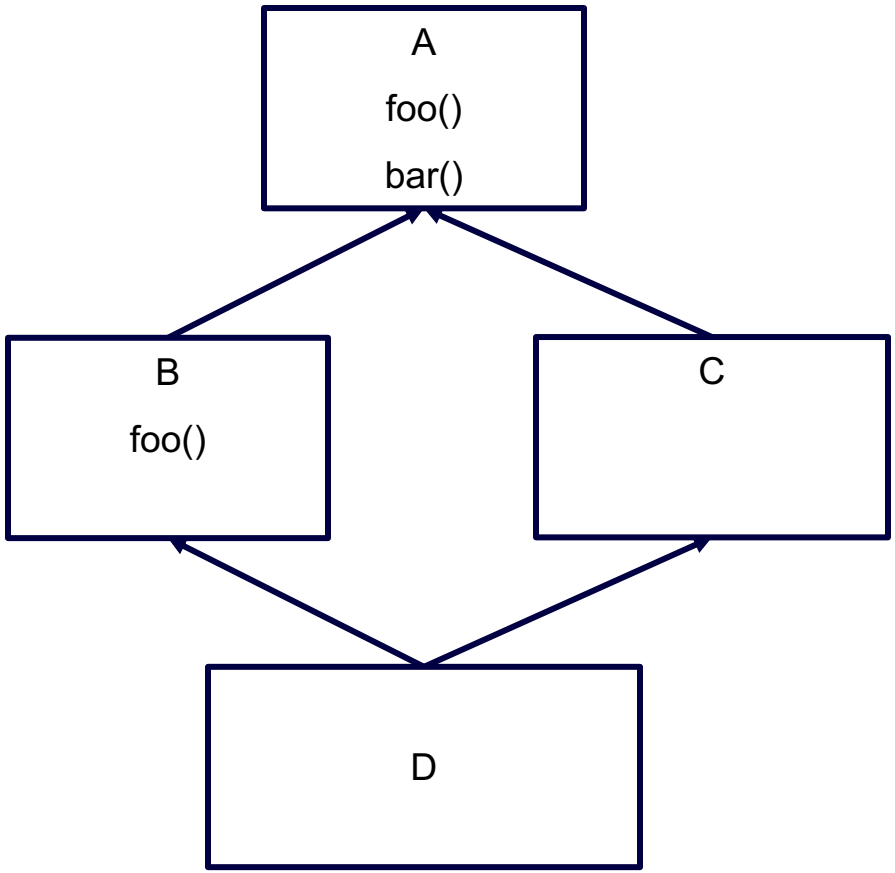
Multiple Inheritance Concern

- The Deadly Diamond of Death!

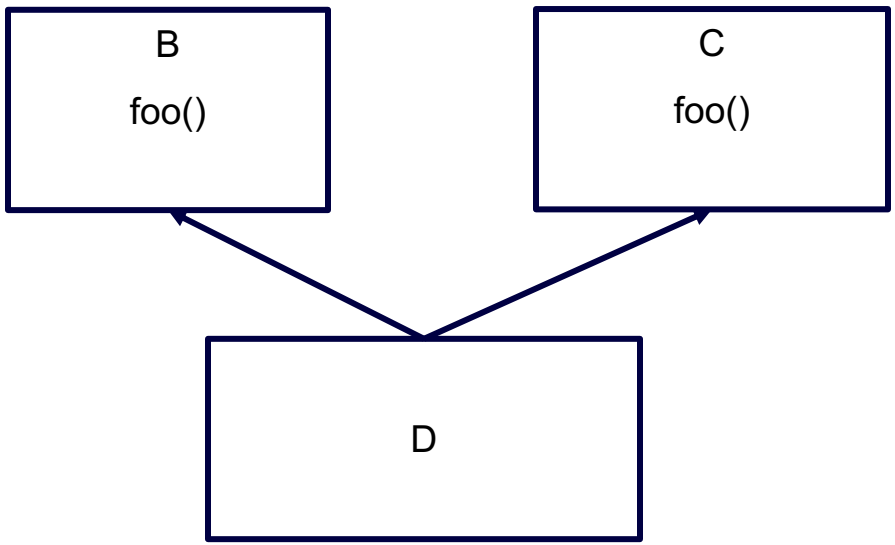


Multiple Inheritance Concern

- The Deadly Diamond of Death!



Multiple Inheritance Concern



On Tap For Today

- Polymorphism
  - Prior Usages
  - Inheritance
- Overriding Functions
- Practice

To Do For Next Time

- Set4 due tomorrow
- Set5 starts next time