Lab 4: Inheritance

CSCI 104
Inheritance

Reasoning behind it?

- Makes logical sense in the context of **object-oriented programming**
  - We define objects with classes; nice to say something “is” something else, ex. A square “is” a polygon
- This “is a” relationship isn’t just great logically, it’s great for our code!
  - Allows us to share and reuse code
  - Ex. Don’t have to define the variable “num_sides” twice for polygon example if a square can use the general polygon code
    - Code reuse is important on a very large scale
Association

- This is another term commonly used in object oriented programming that’s good to be familiar with
- Inheritance = “is a”, association = “has a”
- Association is for when classes are related, but can’t be described in terms of each other
- For example, a purse “is a” bag, and a purse “has a” wallet and “has a” chapstick
Polymorphism

- Easily confused with inheritance, since they generally rely on each other
- Inheritance is what’s used to create relationships between your classes
  - What logically connects things and allows for shared code
- Polymorphism is more based on the program, and how it decides to handle your classes’ relationships
  - Polymorphism is specifically the thing where you call a function that is defined at multiple levels of inheritance, and the correct definition of the function is called!
Polymorphism Example with Animals

- grandparent class Animal, parent class FarmhouseAnimal, child Pig
Polymorphism Example with Animals

- grandparent class Animal, parent class FarmhouseAnimal, child Pig

```cpp
int main() {
    Animal* an = new Pig("Annie", "Farmer Bridget", true);
    an->printNoise();
    delete an;
    return 0;
}
```

What will be outputted??

*(look back at previous slide)*
Polymorphism Example with Animals

- “Oink” will! This is because of polymorphism.
- Why isn’t it “Moo” from the FarmhouseAnimal class, or “Animal Noise”??
- Polymorphism allows parent classes to be able to reach down and access their children’s overridden definition of a function
- Wait... but how does C++ know how to do this? Is it always this easy?
Virtual Functions

- The reason the concept of polymorphism was able to be leveraged was because we used some special C++ keywords
- Use the keyword `virtual` before a function signature for polymorphism
- For children classes when overriding the parent function, put `override` at the end
- If you are the last child class (i.e. you won’t have any children that will override the function), then you don’t have to put `virtual` at the beginning

```cpp
virtual void printNoise() {
    cout << "Animal noise!" << endl;
}

virtual void printNoise() override {
    cout << "Moo" << endl;
}

void printNoise() override {
    cout << "Oink" << endl;
}
```
Pure Virtual Functions

- Okay, but no animal actually says “Animal noise!” except maybe humans and talented parrots
- We should leave this to our children classes to define, and not even attempt to define it ourselves
- Enter: **pure virtual functions**! Pure virtual functions is a way for a class to say: *I will NOT define this function, and it’s totally up to my children to do*
- When we have a pure virtual function in a class, we cannot instantiate (i.e. create an object of) the class, since not all of it’s functions are defined!
Pure Virtual Functions

- Better Animal printNoise()

- We can still have an Animal* pointer, we just wouldn’t be able to do something like:
  Animal* an = new Animal(“Aayushi”);

A final note on the use of virtual keywords:

- Make sure that if you are trying to use polymorphism, you define a virtual destructor! Otherwise, the compiler will give a warning. You don’t even have to put anything in it, it’s just so it can call the right one and delete everything.
Private, Protected, and Public

- Private, protected, and public are C++ keywords that define access levels for classes and how other classes can use them.

<table>
<thead>
<tr>
<th>Base class member access specifier</th>
<th>Type of Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Public</td>
</tr>
<tr>
<td>Public</td>
<td>Public</td>
</tr>
<tr>
<td>Protected</td>
<td>Not accessible (Hidden)</td>
</tr>
<tr>
<td>Private</td>
<td>Not accessible (Hidden)</td>
</tr>
</tbody>
</table>

- Why are these words on both axes of this table???
- They are both variable/function access specifiers and can define inheritance.
Private, Protected, and Public

<table>
<thead>
<tr>
<th>Base class member access specifier</th>
<th>Type of Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>Public</td>
</tr>
<tr>
<td>Protected</td>
<td>Protected</td>
</tr>
<tr>
<td>Private</td>
<td>Private</td>
</tr>
</tbody>
</table>

Keyword used to define type of inheritance

```cpp
class Pig : public FarmhouseAnimal {
  public:
  Pig(string n, string o, bool m) : FarmhouseAnimal(n, o) {
    likesMud = m;
  }

  ~Pig() {};

  void printNoise() override {
    cout << "Oink" << endl;
  }

  private:
  bool likesMud;
};
```

Keyword used to define member functions/vars
Private, Protected, and Public

- Public means any outside class (and itself) can access / use / call / etc.
- Protected means only children classes (and itself) can access / use / call / etc.
- Private means only ITSELF (not even children!!!) can access / use / call / etc.
TO DO:

- `git pull` the lab4 folder within the labs repo (should be familiar by now!)
- Read the write up on bytes
- Work on the lab in your **docker** environment
- Get checked off