Polymorphism 2

CMSC 202

Warmup

What is wrong with the following code?
What error will it produce? (Hint: it already compiles)

```
for (unsigned int i = 1; i < customers.size(); ++i)
{
  for (unsigned int j = i - 1; j >= 0; --j)
  {
     if (customers.at(j)->GetUsername()
          > customers.at(j+1)->GetUsername())
     {
        Customer* temp = customers.at(j);
        customers.at(j) = customers.at(j+1);
        customers.at(j+1) = temp;
     }
  }
}
```

Review

Polymorphism
- Ability to dynamically decide which method to call
- C++
  - Base class pointer
  - Derived class object
  - 'virtual' keyword
- Run-time
  - Call method on pointer
  - Runs method of the derived class

What about destructors?

Imagine the following:

```
class Animal
{
public:
    ~Animal();
};

class StarFish : public Animal
{
public:
    StarFish();
    ~StarFish();
    void RegrowArm(int i);
    void LoseArm(int i);
private:
    Arm* arms;
};
```

```
StarFish::StarFish()
{
    arms = new Arm[5];
}

StarFish::~StarFish()
{
    delete [] arms;
    arms = NULL;
}

int main()
{
    Animal* a = new StarFish();

    delete a;
    a = NULL;
}
```

Oh No! Only an Animal is destroyed!!!

What happens here?

Virtual Destructors

Remember

- Static binding means that we call the POINTER's method
- Dynamic binding means that we call the OBJECT's method
  - Requires the 'virtual' keyword

Rule of thumb?

- If a class has one or more virtual method – give it a virtual destructor!
  - Expect this class to be a base class, eventually

Let's rewrite that class…

Virtual Destructors

Imagine the following:

```
class Animal
{
public:
    virtual ~Animal();
};

class StarFish : public Animal
{
public:
    StarFish();
    ~StarFish();
    void RegrowArm(int i);
    void LoseArm(int i);
private:
    Arm* arms;
};
```

```
StarFish::StarFish()
{
    arms = new Arm[5];
}

StarFish::~StarFish()
{
    delete [] arms;
    arms = NULL;
}

int main()
{
    Animal* a = new StarFish();

    delete a;
    a = NULL;
}
```

Dynamic binding – a StarFish is destroyed!!!

What happens here?

Designing Inheritance

For Base Class
- Methods
  - Identify common operations of ALL derived classes
  - Identify which are type-dependent
    - These are (pure) virtual and will be overridden
  - Identify access level for each method
- Data Members
  - Identify common data of ALL derived classes
  - Identify access level for each data member

Aggregation Problem?

```
class Zoo
{
public:
    Zoo(const Zoo& zoo);
private:
    vector<Animal*> animals;
};

Zoo::Zoo(const Zoo& zoo)
{

    for (unsigned i = 0; i < zoo.animals.size(); ++i)
    {
        animals.push_back( _____________________________ );
    }
}
```

What goes here?
- new Animal(*zoo.animals.at(i))
- new Lion(*zoo.animals.at(i))
- new Gecko(*zoo.animals.at(i))
- new StarFish(*zoo.animals.at(i))
- new Penguin(*zoo.animals.at(i))

Aggregation Solution

Clone()
- Define a virtual method named Clone()
- Returns a pointer to current type
- Override in derived classes
- Might be pure virtual in base class

Example:
- virtual Animal* Clone() const = 0;
- virtual Lion* Clone();

Revised Animal Hierarchy

```
class Animal
{
public:
    virtual ~Animal();
    virtual Animal* Clone() const = 0;
};

class StarFish : public Animal
{
public:
    StarFish(const Starfish& s);
    StarFish* Clone() const;
};

class Lion : public Animal
{
public:
    Lion(const Lion& l);
    Lion* Clone() const;
};
```

How does this work? Aren't the signatures different??

```
StarFish* StarFish::Clone() const
{
    return new StarFish( *this );
}

Lion* Lion::Clone() const
{
    return new Lion( *this );
}
```

Revised Zoo

```
class Zoo
{
public:
    Zoo(const Zoo& zoo);
private:
    vector<Animal*> animals;
};

Zoo::Zoo(const Zoo& zoo)
{

    for (unsigned i = 0; i < zoo.animals.size(); ++i)
    {
        animals.push_back( zoo.animals.at(i)->Clone() );
    }
}
```

Inheritance in Real Life

Vehicles
Sports
Entertainment Products
Computers
Music
Writing Utensils
Food
Restaurants
Data Structures

Basically – anything that you can classify into a set of categories or hierarchy of categories!

Challenge

Pick an object from the world around you
Define an inheritance hierarchy for that item
Demonstrate the use of virtual destructors in your hierarchy
Demonstrate cloning in your hierarchy