CS1020E: Data Structures and Algorithms I

Tutorial 2 – Advanced OO

(Week 4, starting 29 August 2016)

1. *Advanced OO*

Old McDonald has a farm with some animals.

Each animal has a name, and makes a sound. Some animals are flyers. These animals can fly – once they start to fly, their sound is “flap flap” till they stop. Some flyers are gliders. These animals can glide – if they are flying, once they start to glide, their movement is “whoosh” till they stop.

Your junior has written this code to describe the animals in the farm. However, his code cannot be compiled, as there are 4 errors within these 3 classes. (Only 1 of these errors affect compilation)

```
class Animal {
     string _name; // e.g. Cow
     string _sound; // e.g. moo
  public:
     Animal(string name, string sound) { _name = name; _sound = sound; }
     string getName() { return _name; }
     void makeSound() { cout << _name << " goes " << _sound << endl;}
};
class Flyer : public Animal {
  protected:
     string _name;
     string _sound;
     bool _isFlying;
  public:
     Flyer(string name, string sound)
          : _name(name), _sound(sound), _isFlying(false) {}
     void makeSound() {
          if(_isFlying) cout << getName() << " goes flap flap" << endl;
          else Animal::makeSound();
     }
     void fly() { _isFlying = true; }
     void stop() { _isFlying = false; }
};
class Glider : Flyer {
     bool _isGliding;
  public:
     Glider(string name, string sound)
          : Flyer(name, sound), _isGliding(false) {}
     void glide() { if(_isFlying) _isGliding = true; }
     void stop() { _isFlying = false; _isGliding = false; }
     void makeSound() {
          if(_isGliding) cout << getName() << " goes whoosh" << endl;
          else makeSound();
     }
};
```

(a) What does the `protected` keyword mean? In this example, how is it useful?

(b) Within a member function in the Flyer class, why can `getName()` be invoked?

(c) How is overriding demonstrated here, and how is it useful?

(d) Identify and rectify the 4 errors.

Tip: Try to compile your code! Once the compilation errors are rectified, instantiate objects and test.

Related Concepts

- Member inheritance
- Overriding, scope resolution
- Member variables **NOT** overridden
- Constructors chained, **NOT** inherited
- public access vs public inheritance

Answer

(a)
`protected` is an access modifier, in between `public` and `private`. Protected access means the **member is accessible from subclasses, but not other** classes. This allows subclasses to use the superclass' data and functionality, while protecting them from the outside world. The keyword can be applied to both member variables and member functions.

In this example, we want to encapsulate whether a Flyer object is currently flying, from the outside world, but allow a Glider object (which IS also a Flyer) to read and modify this data.

(b)
Protected and public **members are inherited** from the superclass Animal into the subclass Flyer, as with member variables. This means that a Flyer object uses the `getName()` **functionality implemented within the superclass** Animal. In this example, `getName()` is equivalent to `Animal::getName()` or `this->Animal::getName()`.

Similarly, the protected member variable `_isFlying` can be used within a member function in Glider class because it is inherited. Such use is equivalent to `this->Flyer::_isFlying`.

(c)
Flyer and Glider have their own implementations of `makeSound()` and `stop()`. A Glider object would **choose to use its own implementation of member function** `Glider::stop()` instead of `Flyer::stop()`, because Glider's `stop()` is said to override Flyer's `stop()`.

Through overriding, for the same functionality, we can selectively make subclasses have different behavior from that of the superclass.

(d)

1 – Member **variables DO NOT override** variables with the same name from the superclass, unlike member functions. Therefore, a Flyer object has two names, `Animal::_name` (hidden from the Flyer class because it is private) and `Flyer::_name`!

Lesson: In a subclass, avoid declaring member variable that has the same name as that of a superclass member variable. You will confuse yourself.

If `Animal::_name` is `protected` or `public`, it will be shadowed (hidden) by `Flyer::_name`.

Solution: As an Animal already has a name and sound, we **do not want to duplicate these data** for Flyers. We can use the `getName()` and `makeSound()` inherited from Animal to achieve the needed functionality. Therefore, within the Flyer class, strings `_name` and `_sound` should not be declared.

2 – Within `Glider::makeSound()`, there is a call to `makeSound()`. As there are 3 implementations of `makeSound()` in Glider and its superclasses, the **lowest in the inheritance hierarchy will be chosen**, `Glider::makeSound()`. This means that we have a function that repeatedly invokes itself. After resolving all compilation errors, a call to this function will result in infinite recursion (infinite loop).

Solution: After overriding `Flyer::makeSound()` with `Glider::makeSound()`, we want to use Flyer's `makeSound()` functionality. Use the **scope resolution operator** (`::`) to specify which implementation of `makeSound()` you want to use, i.e. `Flyer::makeSound()`.

3 – For compilation errors, **learn to read error messages** and identify the problem(s). At the very least, you may be able to identify where the error occurred, based on the **line number**.

<file name>:**<line>**:**<character>**:**<message>**

```
mcd.cpp:4:9: error: 'std::string Animal::_name' is private
  string _name; // e.g. Cow
         ^
```

Here, we are trying to access a private attribute `Animal::_name` from the initialization list of the constructor of subclass Flyer, in an attempt to set Animal's attributes.

Solution: We have to **invoke the superclass constructor** directly in the **initialization list**. The only way we can set the name and sound of an Animal is through Animal's constructor (when it is created). Attempting to invoke `Animal(…)` within Flyer's constructor body fails too.

Call immediate superclass constructor in subclass' initialization list, not in constructor body.

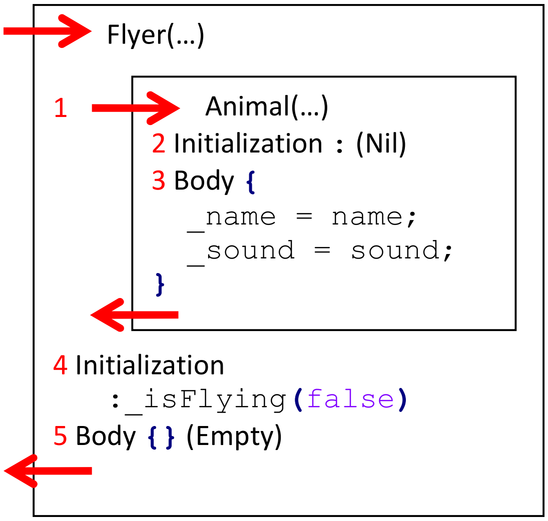
Correct syntax

```
Flyer(…) : Animal(name,sound),…
```

This error highlights 2 important points: **constructors are not inherited** – you cannot otherwise use `Animal(…)` functionality within Flyer; constructor **execution is chained**.

When the subclass' constructor is called, its initialization list and body are executed last. The topmost constructor in the inheritance hierarchy completes execution first.

As we start to initialize `Flyer::_isFlying`, Animal would have been completely initialized, while name and sound attributes would already have been assigned values through Animal's constructor body.



Flyer(…)

1 Animal(…)

2 Initialization : (Nil)

3 Body {
```
_name = name;
_sound = sound;
}
```

4 Initialization
```
:_isFlying(false)
```

5 Body { } (Empty)

4 – Glider's inheritance of Flyer is not `public`. There will be problems accessing members from outside the Glider class.

Try instantiating a Glider object and calling its inherited member function `getName()`, which is supposed to be a public inherited member function. Without public inheritance, is `getName()` accessible from the outside world?

If you are interested to find out more, read up this post and the first two replies:

http://stackoverflow.com/questions/860339/
difference-between-private-public-and-protected-inheritance

Corrected Code Snippet

```
class Animal {
     string _name;
     string _sound;
  public:
     Animal(string name, string sound) { _name = name; _sound = sound; }
     string getName() { return _name; }
     void makeSound() { cout << _name << " goes " << _sound << endl;}
};
class Flyer : public Animal {
  protected:
     bool _isFlying; // Error 1: Shadowed variables removed
  public:
     Flyer(string name, string sound)
          : Animal(name, sound), _isFlying(false) {} //Err 3: C'tor call
     void makeSound() {
          if(_isFlying) cout << getName() << " goes flap flap" << endl;
          else Animal::makeSound();
     }
     void fly() { _isFlying = true; }
     void stop() { _isFlying = false; }
};
```

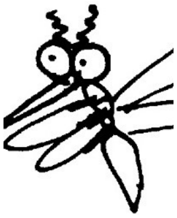
```
class Glider : public Flyer { // Error 4: Inheritance made public
     bool _isGliding;
  public:
     Glider(string name, string sound)
          : Flyer(name, sound), _isGliding(false) {}
     void glide() { if(_isFlying) _isGliding = true; }
     void stop() { _isFlying = false; _isGliding = false; }
     void makeSound() {
          if(_isGliding) cout << getName() << " goes whoosh" << endl;
          else Flyer::makeSound();// Error 2: Scope resolution used
     }
};
```

2. *Inheritance & Polymorphism*

Related to Q1, Old McDonald still has a farm with some animals. You want to **make use of polymorphism** to allow 5 animals in the farm to `makeSound()`, without having to concern yourself about what type of Animal each is. To simplify things, let's ignore Gliders. There are only Flyers and non-Flyers.

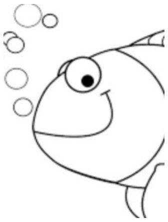
Parrot
"squak"
Perched

Cow
"moo"



Mosquito
"buzz"
Flying

Sheep
"mehh"



Fish
"blurp"

Picture credits: clipartpanda.com

```
class Animal { ... }; // Rectify the problem in (c)
class Flyer : public Animal { ... };
class OldMcDonald {
  private:
     Animal** _farm; // Old McDonald had a farm (still has now)
     const int _size; // Fixed farm size of 5
  public:
     OldMcDonald() {
          /* TODO: Create your farm, an array of Animal* elements */
     }
     ~OldMcDonald() {
          /* TODO: Old McDonald has no (more) farm... */
     }
     void makeSomeNoise() {
          /* TODO: Make sound(s) without looking out for Flyers...! */
     }

...
```

```
    void fillThisFarm() {
        _farm[0] = new Flyer("Parrot”, "squak");
        _farm[1] = new Animal("Cow", "moo");
        _farm[2] = new Flyer("Mosquito", "buzz");
        ((Flyer*)_farm[2])->fly();
        _farm[3] = new Animal("Sheep", "mehh");
        _farm[4] = new Animal("Fish", "blurp");
    }
};
```

(a) What is the datatype of `_farm[0]`? Why can a pointer to Flyer be assigned to `_farm[0]`?

(b) Why can't `((Flyer*)_farm[2])->fly()` be replaced with `_farm[2]->fly()`?

(c) With Animal and Flyer classes from Q1, why will polymorphism not work? Make the necessary change.

(d) Solve the problem, ensuring that the sounds output by each animal are correct. The output of `makeSomeNoise()` should be:

Parrot goes **squak**

Cow goes moo

Mosquito goes **flap flap**

Sheep goes mehh

Fish goes blurp

Related Concepts

- Substitutability principle
- Function must exist in *(pointer type)
- Polymorphic functions only with `virtual`

Answer

(a)

Datatype of `_farm[0]` is `Animal*`, or **Animal pointer**. This means it stores the address of Animal object. `_farm` is an `Animal**` typed variable, so after dereferencing (`[]`), `_farm[0]`'s type is `Animal*`.

In C++, we can **substitute a superclass-typed object with a subclass-typed object** when assigning to a pointer. In other words, since we expect the address of an Animal to be in `_farm[0]`, we allow the Animal pointer to store the address of a Flyer, because a Flyer IS an Animal.

The **new** keyword creates an object on the heap and returns the address of that object. A Flyer pointer is returned and assigned to the Animal pointer `_farm[0]`.

(b)

As `_farm[2]` is an Animal pointer, (`_farm[2]->`) **dereferences to an Animal**, which does not have the member function `fly()`. Although this Animal is a Flyer, the compiler does not know that. Therefore, we first have to **explicitly downcast the pointer** from an Animal pointer to a Flyer pointer, and then dereference the pointer to get a Flyer object, which has `fly()`.

Remember, we are casting the pointer, not the object, from one type to another!

(c)

The `virtual` keyword is missing in `Animal::makeSound()`. **Without the keyword**, static binding is used. The function executed will **follow the implementation in *(pointer type)**, regardless of the object type. The flying `Mosquito goes buzz` because `_farm[2]->makeSound()` uses `Animal::makeSound()`.

On the other hand, **with the keyword** applied to a function in the *(pointer type) or above, dynamic binding occurs. The function executed will **follow the implementation in the actual object type**, allowing the program to behave in a polymorphic manner (one pointer type, many different function implementations). The flying `Mosquito goes flap flap` because `_farm[2]->makeSound()` uses `Flyer::makeSound()`.

(d) After `Animal::makeSound()` is made `virtual`:

```
OldMcDonald() : _size(5) { _farm = new Animal* [_size]; }
~OldMcDonald() {
     for (int i = 0; i< _size; i++) delete _farm[i];
     delete [] _farm;
}
void makeSomeNoise() {
     for (int i = 0; i < _size; i++) _farm[i]->makeSound();
}
```

Again, remember to **return memory allocated on the heap**, by using the **`delete`** keyword *on the pointers*, and **`delete []`** on pointers to arrays.

- Practise consistently ☻ -

Revise a concept
Test understanding
Code to confirm