CS 162
Intro to Programming II

Polymorphism II

Virtual Functions

- Which is which, for changing the behavior of inherited functions?
  - Overriding refers to doing this change to a virtual function
  - Redefining refers to doing this change to a non-virtual function
  - Overloading refers to the definition of different functions within the same class with the same name and different parameter lists.

Abstract Classes

```
/* Shape.hpp */
class Shape {
public:
   Shape();
   virtual void print() = 0;
   virtual double getArea() = 0;
};

/* Shape.cpp */
   Shape::Shape() {
}
```

Abstract Classes

```
/* Shape.hpp */
class Shape {
public:
   Shape();
   virtual void print() = 0;
   virtual double getArea() = 0;
};
```

- These are pure virtual functions
- Cannot create a shape object- abstract class

Abstract Classes

- Why abstract classes?
  - It has the common characteristics
  - Details are left to specific subclasses
  - In this case we want each shape to draw itself and compute its area
  - Details will vary for each shape
  - You do not even need to know what shapes may be used later!

Abstract Classes

- A derived class will also be abstract unless you:
  1. You provide definitions for the inherited pure virtual functions
  2. You do not add any pure virtual functions
- If these conditions are met you can create an object of the derived class

Abstract Classes

- How does this work?
- We will create a derived class for a Rectangle and a Triangle
- Both will define the pure virtural functions
- Both will add member variables and functions (but no pure virtual functions)

Abstract Classes

```
/* Rectangle.hpp */
class Rectangle : public Shape {
public:
   Rectangle(double height, double
   width);
   void print();
   double getArea();
private:
   double height;
   double width;
};
```

Abstract Classes

```
/* Inside Rectangle.cpp. Constructor omitted
due of lack of space */

void Rectangle::print() {
   for( int i = 0; i < height; i++ ) {
      for( int j = 0; j < width; j++ ) {
         std::cout << "*";
      }
      std::cout << std::endl;
   }
}

double Rectangle::getArea() {
   return (height*width);
}
```

Abstract Classes

```
/* Triangle.hpp */
class Rectangle : public Shape {
public:
   Triangle(double height, double
   width);
   void print();
   double getArea();
private:
   double height;
   double width;
};
```

Abstract Classes

```
/* Inside Triangle.cpp. Constructor omitted
due of lack of space */

void Triangle::print() {
   for( int i = 0; i < height; i++ ) {
      for( int j = 0; j <= i; j++ ) {
         std::cout << "*";
      }
      std::cout << std::endl;
   }
}

double Rectangle::getArea() {
   return (0.5*height*width);
}
```

Slicing Problem

```
/* In Bunny.hpp */
class Bunny {
public:
   Bunny(std::string);
   virtual ~Bunny();
   virtual void print() const;
private:
   std::string *name;
};
```

Slicing Problem

```
/* In MutantBunny.hpp */
class MutantBunny : public Bunny {
public:
   MutantBunny(std::string name);
   ~MutantBunny();
   void print() const;
   void addNameOfBunnyEaten(std::string
   name);
private:
   std::vector<std::string>
   *namesOfBunniesEaten;
};
```

Slicing Problem

```
 /* In Bunny.cpp */
void Bunny::print() const {
    std::cout << "Name: " << name << std::endl;
}

 /* In MutantBunny.cpp*/
void MutantBunny::print() const {
    Bunny::print();
    std::cout << "Bunnies eaten: " << std::endl;
    for( int i = 0; i < namesOfBunniesEaten->size();i++ ) {
    std::cout << "\t" << (*namesOfBunniesEaten)[i] <<
          std::endl;
    }
}
```

Slicing Problem

```
MutantBunny mb("Fluffy");
Bunny b = mb;
```

- This assignment of objects works
- The copy does not include any variables in MutantBunny but not in Bunny
- It slices off those data members
- This code will not compile as b is a Bunny:

```
b.addNameOfBunnyEaten(“Bugs”);
```

Slicing Problem

```
MutantBunny *mb = new MutantBunny("Fluffy");
mb->addNameOfBunnyEaten("Bugs");
Bunny *b = mb;
b->print();
```

- This assignment of pointers works
- `print()` calls the version in MutantBunny but not in Bunny
- The output is:

```
Name: Fluffy
Bunnies eaten: Bugs
```

Virtual Destructors

- In Bunny.hpp we had:

```
virtual ~Bunny();
```

- The definitions look like this:

```
/* In Bunny.cpp */
Bunny::~Bunny() {
   delete name;
}

/* In MutantBunny.cpp */
MutantBunny::~MutantBunny() {
   delete namesOfBunniesEaten;
}
```

Virtual Destructors

- If the destructor is not virtual then this code:

```
Bunny *b = new MutantBunny("Bob");
delete b;
```

- Will clean up the name member in Bunny
- But not the namesOfBunniesEaten in MutantBunny
- If the base class destructor is virtual then all derived class destructors are virtual and will be called