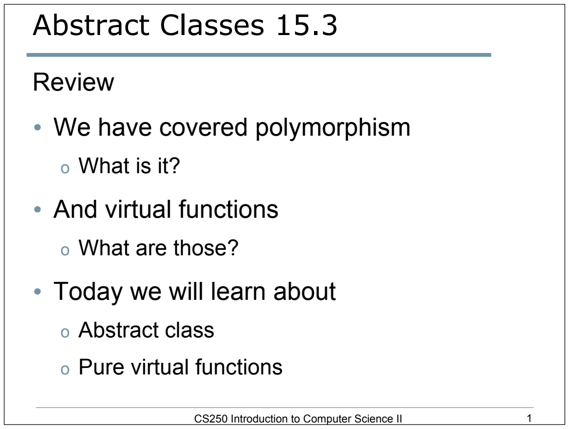
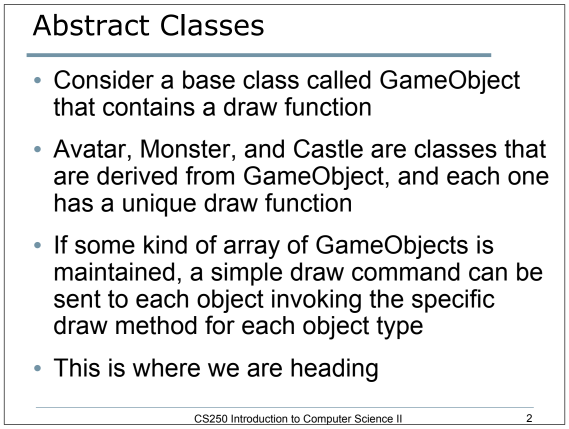


Abstract Classes 15.3

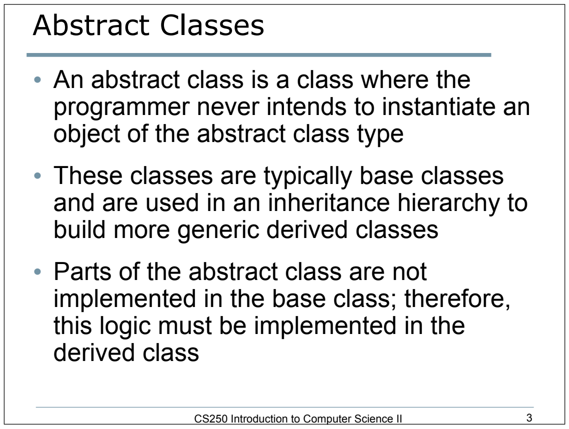
Review

- We have covered polymorphism
  - What is it?
- And virtual functions
  - What are those?
- Today we will learn about
  - Abstract class
  - Pure virtual functions



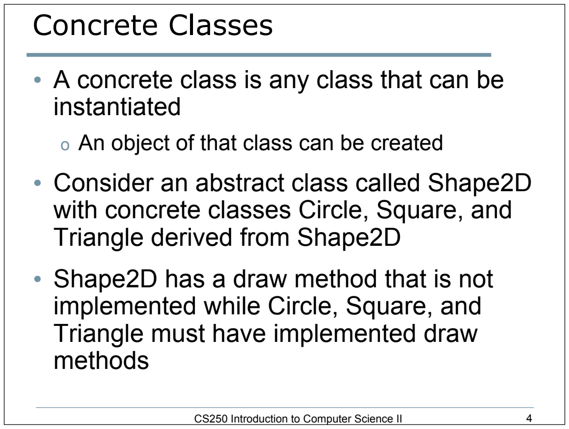
Abstract Classes

- Consider a base class called GameObject that contains a draw function
- Avatar, Monster, and Castle are classes that are derived from GameObject, and each one has a unique draw function
- If some kind of array of GameObjects is maintained, a simple draw command can be sent to each object invoking the specific draw method for each object type
- This is where we are heading



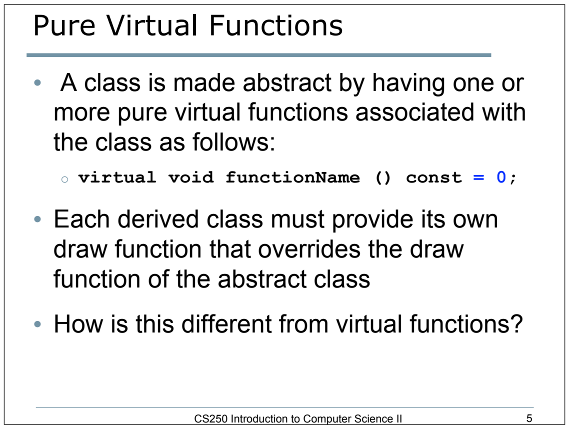
Abstract Classes

- An abstract class is a class where the programmer never intends to instantiate an object of the abstract class type
- These classes are typically base classes and are used in an inheritance hierarchy to build more generic derived classes
- Parts of the abstract class are not implemented in the base class; therefore, this logic must be implemented in the derived class



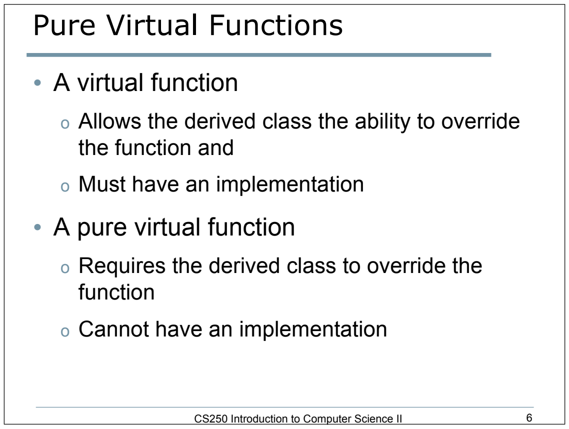
Concrete Classes

- A concrete class is any class that can be instantiated
  - An object of that class can be created
- Consider an abstract class called Shape2D with concrete classes Circle, Square, and Triangle derived from Shape2D
- Shape2D has a draw method that is not implemented while Circle, Square, and Triangle must have implemented draw methods



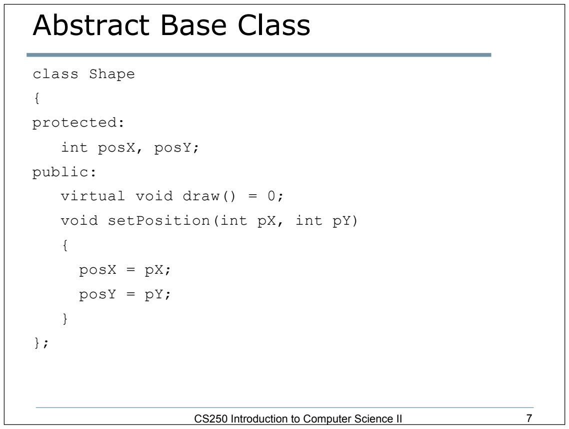
Pure Virtual Functions

- A class is made abstract by having one or more pure virtual functions associated with the class as follows:
  - `virtual void functionName () const = 0;`
- Each derived class must provide its own draw function that overrides the draw function of the abstract class
- How is this different from virtual functions?



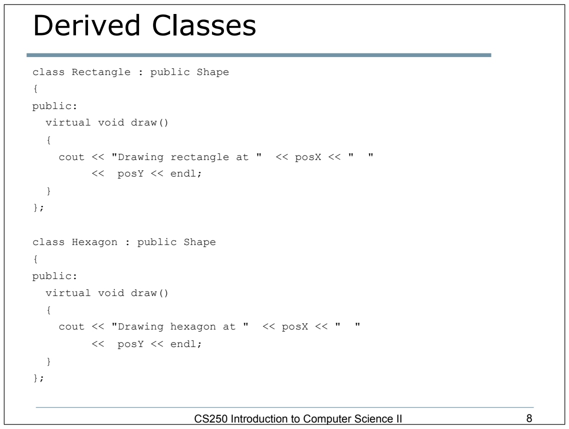
Pure Virtual Functions

- A virtual function
  - Allows the derived class the ability to override the function and
  - Must have an implementation
- A pure virtual function
  - Requires the derived class to override the function
  - Cannot have an implementation



Abstract Base Class

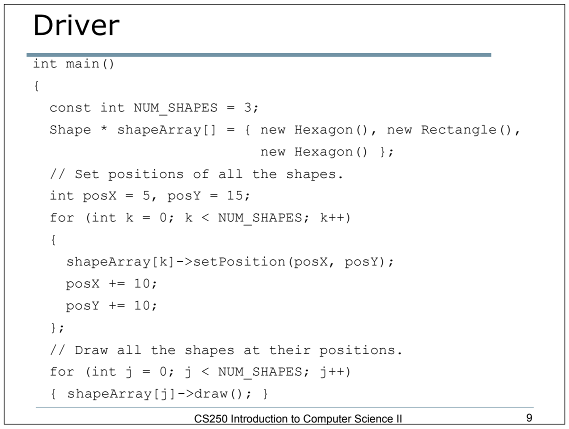
```
class Shape
{
protected:
   int posX, posY;
public:
   virtual void draw() = 0;
   void setPosition(int pX, int pY)
   {
     posX = pX;
     posY = pY;
   }
};
```



Derived Classes

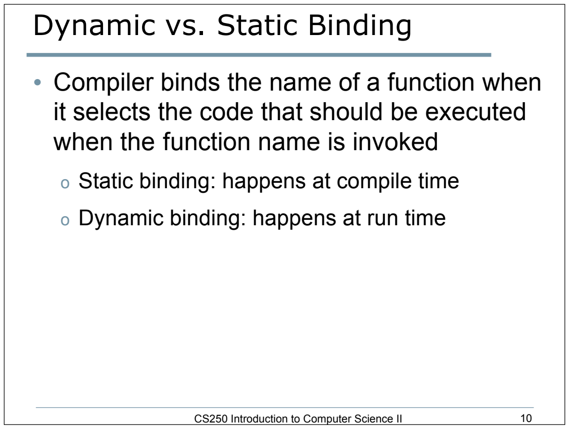
```
class Rectangle : public Shape
{
public:
  virtual void draw()
  {
    cout << "Drawing rectangle at "  << posX << "   "
         <<  posY << endl;
  }
};

class Hexagon : public Shape
{
public:
  virtual void draw()
  {
    cout << "Drawing hexagon at "  << posX << "   "
         <<  posY << endl;
  }
};
```



Driver

```
int main()
{
  const int NUM_SHAPES = 3;
  Shape * shapeArray[] = { new Hexagon(), new Rectangle(),
                           new Hexagon() };
  // Set positions of all the shapes.
  int posX = 5, posY = 15;
  for (int k = 0; k < NUM_SHAPES; k++)
  {
    shapeArray[k]->setPosition(posX, posY);
    posX += 10;
    posY += 10;
  };
  // Draw all the shapes at their positions.
  for (int j = 0; j < NUM_SHAPES; j++)
  { shapeArray[j]->draw(); }
```



Dynamic vs. Static Binding

- Compiler binds the name of a function when it selects the code that should be executed when the function name is invoked
  - Static binding: happens at compile time
  - Dynamic binding: happens at run time