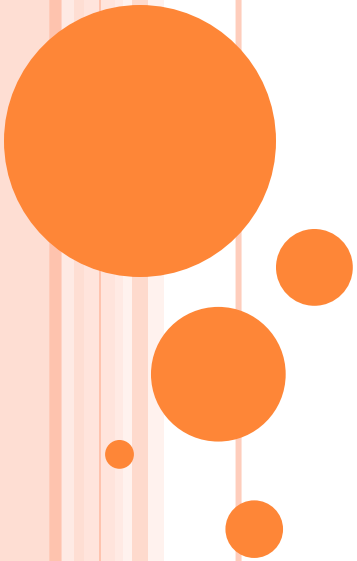

Virtual Functions

CS A150 - C++ Programming 1

Virtual Functions Basics

- **Polymorphism**
  - Associating many meanings to one function
    - Values of different data types handled by using a uniform interface
  - Fundamental principle of **object-oriented programming**
  - Virtual functions provide this capability
- **Virtual**
  - Existing in "essence" though not in fact
- **Virtual Function**
  - Can be "used" before it is "defined"

Virtual Functions

- Classes for several kinds of figures
  - Rectangles, circles, ovals, etc.
  - Each figure is an object of a different class
    - **Rectangle data:** height, width, center point
    - **Circle data:** radius, center point
- All derive from one parent class → Shape
- Require function for all classes: `draw()`
  - Different instructions for each figure

Virtual Functions (cont.)

- Each class needs a different *draw* function
- Can be called "draw" in each class, so:

```
Rectangle r;
Circle c;
r.draw();  //calls Rectangle class's draw
c.draw();  //calls Circle class's draw
```

- Nothing new here yet…

Virtual Functions (cont.)

```
class Shape
{
public:
     Shape();
     void draw() const;
     void center() const;

     ~Shape();
}
```

Function **center** moves a shape to the center of the screen.
First erases what is on the screen, and then re-draws the shape using the function **draw**.
All children will **inherit** the function **center**.

Complications: Which **draw** function to use? From which class?

```
class Circle: public Shape
{
public:
     Circle();
     void draw() const;
     ~Circle();
private:
     double radius;
}
```

```
class Rectangle: public Shape
{
public:
     Rectangle();
     void draw() const;
     ~Rectangle();
private:
     double height;
     double width;
}
```

Virtual Functions (cont.)

- Consider a new kind of figure comes along:

  `Triangle class`

  derived from `Figure` class
- Function `center()` inherited from `Figure`
  - Will it work for triangles?
  - It uses `draw()`, which is different for each figure!
  - It will use `Figure::draw()` → will ***not*** work for triangles
- Want inherited function `center()` to use function `Triangle::draw()` *not* function `Figure::draw()`
  - But class `Triangle` was not even written when `Figure::center()` was! Does not know "triangles"!

Virtual Functions (cont.)

- **Virtual functions** are the answer
- Tell compiler:
  - "Don't know how function is implemented"
  - "Wait until used in program"
  - "Then get implementation from object instance"
- Called **late binding** or **dynamic binding**
  - Virtual functions implement late binding

Virtual Functions - Examples

- These examples have walk-through explanations that are easy to follow in the project instead of having them in the slides:
  - Virtual_1
  - Virtual_2
  - Virtual_3
  - Virtual_4
  - Virtual_5

Overriding

- When a **virtual function *definition*** is changed in a **derived class**
  - We say it is been "**overridden**"
  - Similar to ***redefined***
- So:
  - Virtual functions are ***overridden***
  - **Non**-virtual functions are ***redefined***

Virtual Functions: Why Not All?

- Clear advantages to virtual functions as we have seen
- One major ***disadvantage***: **overhead**
  - Uses ***more*** storage
  - Late binding is "on the fly", so programs run slower.
- So if virtual functions not needed, should not be used.

Pure Virtual Functions

- Base class might not have "meaningful" definition
  - Its purpose solely for others to derive from
- Recall class **`Shape`**
  - All figures are objects of derived classes
    - Rectangles, circles, triangles, etc.
  - Class Shape has no idea how to draw!
- Make it a ***pure* virtual function**:

```
virtual void draw() = 0;
```

Abstract Base Classes

- **Pure virtual functions** require **no** definition
  - Forces all derived classes to define "their own" version
- Class with one or more pure virtual functions is an **abstract base class**
  - Can ***only*** be used as base class
  - No objects can ever be created from it
    - Since it does not have complete "definitions" of all its members

Virtual Destructors

- Recall:
  - **Destructors** are automatically executed when the class object goes out of scope
- Now consider:
  - If we pass the **derived** object to the **non-member** function print as type **base** class, when the object is destroyed, the **destructor** of the **base** class executes regardless of whether the derived class object is passed by reference or by value.
  - Logically, you would think that the **destructor** of the **derived** class is also executed when the class object goes out of scope.
    - Correct?

Virtual Destructors (cont.)

- No, it is not correct. The **destructor** of the derived class will ***not*** be executed.
- To correct the problem:
  - The **destructor** of the base class must be **virtual**.
  - The virtual destructor of a base class automatically makes the destructor of a derived class be virtual so that it can also be executed when the object is out of scope.
    - The derived class destructor will be executed first, then the base class destructor will be executed.

Virtual Destructors

- Any class that includes ***at least one*** **virtual member function** should define a **virtual destructor**

- If you are using inheritance, it is a good idea to have the **destructor** of the base class declared as **virtual**

Summary

- **Late binding** delays decision of which member function is called until **runtime**
  - In C++, virtual functions use **late binding**
- **Pure virtual functions** have no definition
  - Classes with at least one are **abstract**
  - **No** objects can be created from abstract class
  - Used ***strictly*** as base for others to derive
- Make **all destructors virtual**
  - Good programming practice
  - Ensures memory correctly de-allocated

Virtual Functions (end)