Eric Roberts
CS106B
Handout #40
February 20, 2013

Section #6—Trees

For problems 1, 2, and 3, assume that **`BSTNode`** is defined as follows:

```
struct BSTNode {
   string key;
   BSTNode *left, *right;
};
```

1. Tracing binary tree insertion (Chapter 16, review question 9, page 724)

In the first example of binary search trees, the text uses the names of the dwarves from Walt Disney's 1937 classic animated film *Snow White and the Seven Dwarves*. Dwarves, of course, occur in other stories. In J. R. R. Tolkien's *The Hobbit*, for example, 13 dwarves arrive at the house of Bilbo Baggins in the following order:

Dwalin, Balin, Kili, Fili, Dori, Nori, Ori, Oin, Gloin, Bifur, Bofur, Bombur, Thorin

Diagram the binary search tree you get from inserting these names into an empty tree in this order. Once you have finished, answer the following questions about your diagram:

1a. What is the *height* (defined on page 691 as the number of nodes in the longest path from the root to a leaf) of the resulting tree?

1b. Which nodes are leaves?

1c. Which nodes, if any, are out of balance (in the sense that the subtree rooted at that node fails to meet the definition of balanced trees on page 706)?

1d. Which key comparisons are required to find the string **`"Gloin"`** in the tree?

2. Calculating the height of a binary tree (Chapter 16, exercise 6, page 730)

Write a function

```
int height(BSTNode *tree);
```

that takes a binary search tree and returns its height.

3. Checking whether a tree is balanced (Chapter 16, exercise 7, page 731)

Write a function

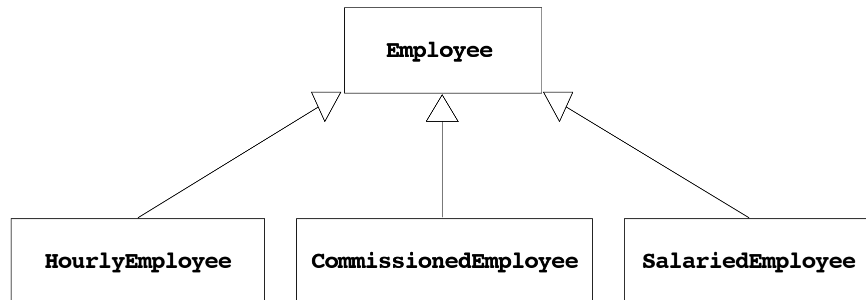
```
bool isBalanced(BSTNode *tree);
```

that determines whether a given tree is balanced according to the definition in the section on "Balanced trees." To solve this problem, all you really need to do is translate the definition of a balanced tree more or less directly into code. If you do so, however, the resulting implementation will be quite inefficient because it has to make several passes over the tree. The real challenge in this problem is to implement the **`isBalanced`** function so that it determines the result without looking at any node more than once.

Problem 4. Designing an interface for class hierarchies

One of the fundamental features of object-oriented languages is that classes form hierarchies through the inheritance chain. In C++, classes can inherit behavior from more than one class, so the notion of a superclass is a bit harder to pin down than it is in Java, where each class has exactly one superclass. In practice, however, most classes in C++ use a single inheritance chain, which means that the class hierarchy is structured in the form of a tree.

Consider, for example, the following diagram



In this diagram, **`HourlyEmployee`**, **`CommissionedEmployee`**, and **`SalariedEmployee`** are each subclasses of the more general **`Employee`** class.

Your job in this problem is to design and implement a class that can store the hierarchical relationships shown in these class diagrams. As the sample diagram indicates, each class has a name and a superclass, which will be **`NULL`** for the root of the tree. To illustrate the idea of inheritance, it is useful to include with each class a list of the methods that it exports directly. Given an object of that class, you could call any of these methods and any of the methods defined by a superclass above it in the hierarchy.

Figure 1 shows a piece of the class hierarchy for the **`GObject`** class hierarchy in the **`acm.graphics`** Java package. Inside the box for each class is a list of the common public methods it implements. Thus, given an object of type **`G3DRect`**, you can call **`setRaised`** because that is defined in the **`G3DRect`** class itself, **`setFilled`** because that is defined in the **`GRect`** class, and **`setColor`** because that is defined all the way back at the **`GObject`** level.

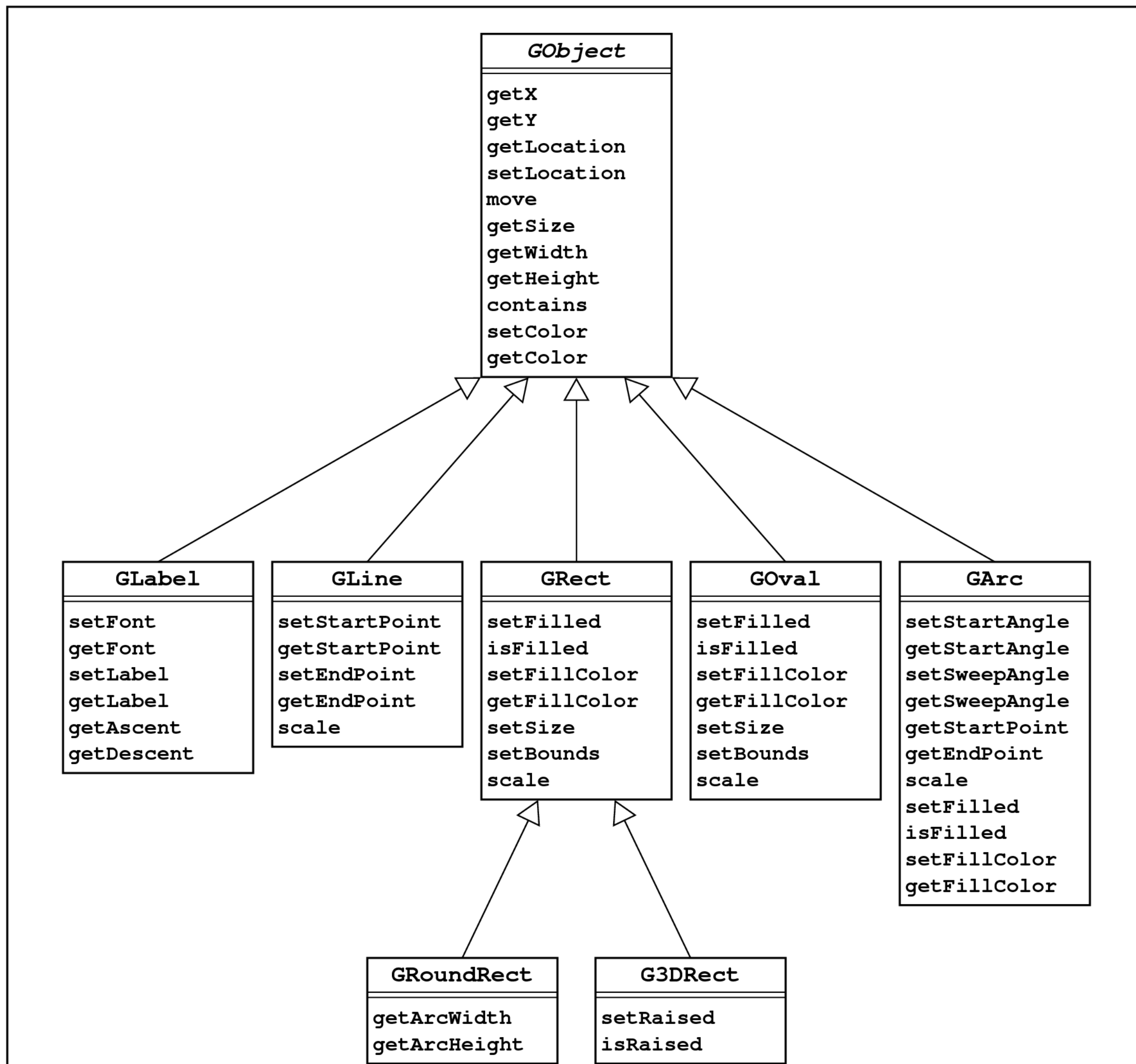
Task 4a: Design the interface for representing class hierarchies

Your first task in solving this problem is to define a structure that is capable of representing the classes and methods in a tree-structured class hierarchy. To do so, you need to write an interface **`class.h`** that exports a class named **`Class`**, which could represent any of the boxes in Figure 1. What methods does **`Class`** need to export? What does its constructor look like? What private instance variables will it need?

Task 4b: Implement the class you defined in the previous task

To complete the class definition, you need to write a file called **`class.cpp`** that supplies the implementation for **`Class`**.

Figure 1. Class diagram for a subset of the ACM graphics library



Task 4c: Write a function that displays all the methods available to a class

Now that you have a definition for `Class`, you can use it to generate some useful information. Working as a client of the `class.h` interface, write a function

```
void listAllMethods(Class *c);
```

that takes a pointer to a `Class` object and displays a list of all the methods you could apply to an object of that class along with the name of the class in which that method is defined. For example, if `classG3Rect` has been initialized to correspond to the `G3Rect` class shown in Figure 1, calling `listAllMethods(classG3Rect)` should produce the following output:

ClassTest

```
Enter class name: G3DRect
GObject::getX
GObject::getY
GObject::getLocation
GObject::setLocation
GObject::move
GObject::getSize
GObject::getWidth
GObject::getHeight
GObject::contains
GObject::setColor
GObject::getColor
GRect::setFilled
GRect::isFilled
GRect::setFillColor
GRect::getFillColor
GRect::setSize
GRect::setBounds
GRect::scale
G3DRect::setRaised
G3DRect::isRaised
```