Inheritance in C++

Class Hierarchies
• Much of the power of modern object-oriented languages comes from the fact that they support class hierarchies. Any class can be designated as a subclass of some other class, which is called its superclass.
• Each subclass represents a specialization of its superclass. If you create an object that is an instance of a class, that object is also an instance of all other classes in the hierarchy above it in the superclass chain.
• When you define a new class in C++, that class automatically inherits the behavior of its superclass.
• Although C++ supports multiple inheritance in which a class can inherit behavior from more than one superclass, the vast majority of class hierarchies use single inheritance in which each class has a unique superclass. This convention means that class hierarchies tend to form trees rather than graphs.

Simplified View of the Stream Hierarchy

Representing Inheritance in C++
• The first step in creating a C++ subclass is to indicate the superclass on the header line, using the following syntax:

```
class subclass : public superclass {
    body of class definition
};
```
• You can use this feature to specify the types for a collection class, as in the following definition of StringMap:

```
class StringMap : public Map<string,string> {
    /* Empty */
};
```

Differences between Java and C++
• In Java, defining a subclass method automatically overrides the definition of that method in its superclass. In C++, you have to explicitly allow for overriding by marking the method prototype with the keyword virtual.
• In Java, all objects are allocated dynamically on the heap. In C++, objects live either on the heap or on the stack. Heap objects are created using the keyword new and are referred to by their address. Stack objects take a fixed amount of space determined by the number and size of the instance variables.
• In Java, it is always legal to assign an object of a subclass to a variable declared to be its superclass. While that operation is technically legal in C++, it rarely does what you want, because C++ throws away any fields in the assigned object that don’t fit into the superclass. This behavior is called slicing. By contrast, it is always legal to assign pointers to objects.

The Employee Hierarchy
In the Employee hierarchy, getPay is implemented differently in each subclass and must therefore be a virtual method.
Abstract Classes

- An **abstract class** is a class that is never created on its own but instead serves as a common superclass for **concrete classes** that correspond to actual objects.
- In C++, any method that is always implemented by a concrete subclass is indicated by including `= 0` before the semicolon on the prototype line, as follows:

```cpp
class Employee {
    virtual double getPay() = 0;
};
class HourlyEmployee : public Employee {
    virtual double getPay();
};
class CommissionedEmployee : public Employee {
    virtual double getPay();
};
class SalariedEmployee : public Employee {
    virtual double getPay();
};
```

The Darwin Simulation Game

Years ago, one of the 106B assignments was the **Darwin** game, which was played on a grid populated by “creatures” trying to “infect” other types.

The standard creatures were:

- **Rover**, which tries to move forward, turning if blocked.
- **Flytrap**, which simply spins to the left.
- **Food**, which does nothing except wait to be eaten.

Specifying Creature Behavior

- The creatures in the Darwin game have different behaviors, which are specified by defining a method called `step`. The definition of the `step` method is different for each subclass:

```cpp
void Rover::step() {
    if (facingEnemy()) {
        infect();
    } else if (isBlocked()) {
        turnLeft();
    } else {
        move();
    }
}
```

- Because the definition of `step` is different in each subclass, this method must be virtual.

The Creature Hierarchy

- `Rover`
- `Flytrap`
- `Food`

Representing Graphical Shapes

- In CS 106A, you learned how to use the `GObject` hierarchy in the `acm.graphics` package, which looks something like this:

```cpp
GObject
- GLine
- GRect
- GOval
- GRoundRect
- G3DRect
- GLabel
- GImage
- GArc
- GPolygon
```

- In C++, the most important thing to keep in mind is that you have to use pointers to these objects.

Exercise: Do Not Enter

- The British version of a “Do Not Enter” sign looks like this:

Write a program that uses the stripped-down version of the `gobjects.h` that displays this symbol at the center of the window. The sizes of the components are given as constants in the starter file.
The GObject Hierarchy

The gobjects.h Interface

```cpp
/* File: gobjects.h */
/* This file defines a simple hierarchy of graphical objects. */

#include "gobjects.h"
#include "gwindow.h"

/* include settings 
   include "gobjects.h"
*/

class GObject {
public:
    /* The GObject class represents the root of the hierarchy and encompasses all objects 
       that can be displayed in a window. Clients will use pointers to 
       a GObject rather than the GObject itself. */
};

class GLine : public GObject {
public:
    /* Subclass: GLine 
       The GLine subclass represents a line segment on the window. */
    GLine(double x1, double y1, double x2, double y2);
    virtual void draw(GWindow & gw);
private:
    double dx; /* Horizontal distance from x1 to x2 */
    double dy; /* Vertical distance from y1 to y2 */
};

class GRect : public GObject {
public:
    /* Subclass: GRect 
       The GRect subclass represents a rectangle on the window. */
    GRect(double x, double y, double width, double height);
    void setFilled(bool flag);
    virtual void draw(GWindow & gw);
private:
    double width, height; /* Dimensions of the rectangle */
    bool filled; /* True if the rectangle is filled */
};
```

The gobjects.h Interface

```cpp
/* */
/* Abstract method: draw 
 * Usage: gobj->draw(gw);
 */

virtual void draw(GWindow & gw) = 0;
```

The gobjects.h Interface

```cpp
class Ghost : public GObject {
public:
    /* Constructor: Ghost 
       Usage: Ghost *gp = new Ghost(x, y, width, height);
     */
    Ghost(double x, double y, double width, double height);
    /* Method: setFilled 
       Usage: gp->setFilled(flag);
     */
    void setFilled(bool flag);
    virtual void draw(GWindow & gw);
private:
    double width, height; /* Dimensions of the rectangle */
    bool filled; /* True if the rectangle is filled */
};
```
class GOval : public GObject {
public:
/* Constructor: GOval */
* Usage: GOval *op = new GOval(x, y, width, height);
* Creates an oval inscribed in the specified rectangle.
*/
GOval(double x, double y, double width, double height);

/* Method: setFilled */
* Usage: op->setFilled(flag);
* Indicates whether the oval is filled.
*/
void setFilled(bool flag);

virtual void draw(GWindow & gw);

private:
double width, height;
/* Dimensions of the bounding rectangle */
bool filled;
/* True if the oval is filled */
};

Implementation of the GObject Class

class GRect : public GObject {
public:
/* Constructor: GRect */
* Usage: GRect *op = new GRect(x, y, width, height);
* Creates a rectangle with the given coordinates.
*/
GRect(double x, double y, double width, double height);

/* Method: setFilled */
* Usage: op->setFilled(flag);
* Indicates whether the rectangle is filled.
*/
void setFilled(bool flag);

virtual void draw(GWindow & gw);

private:
double width, height;
/* Width and height of the rectangle */
bool filled;
/* True if the rectangle is filled */
};

Implementation of the GRect Class

class GLine : public GObject {
public:
/* Constructor: GLine */
* Usage: GLine *op = new GLine(x1, y1, x2, y2);
* Creates a line with the given endpoints.
*/
GLine(double x1, double y1, double x2, double y2);

/* Method: setFilled */
* Usage: op->setFilled(flag);
* Indicates whether the line is filled.
*/
void setFilled(bool flag);

virtual void draw(GWindow & gw);

private:
double x, y, dx, dy;
/* Coordinates and dx/dy values of the line */
};

Implementation of the GLine Class

class GOval : public GObject {
public:
/* Constructor: GOval */
* Usage: GOval *op = new GOval(x, y, width, height);
* Creates an oval inscribed in the specified rectangle.
*/
GOval(double x, double y, double width, double height);

/* Method: setFilled */
* Usage: op->setFilled(flag);
* Indicates whether the oval is filled.
*/
void setFilled(bool flag);

virtual void draw(GWindow & gw);

private:
double width, height;
/* Dimensions of the bounding rectangle */
bool filled;
/* True if the oval is filled */
};

Implementation of the GOval Class

Calling Superclass Constructors

• When you call the constructor for an object, the constructor ordinarily calls the default constructor for the superclass, which is the one that takes no arguments.

• You can call a different version of the superclass constructor by adding an initializer list to the constructor header. This list consists of a colon followed either by a call to the superclass constructor or initializers for its variables.

• As an example, the following definition creates a GSquare subclass whose constructor takes the coordinates of the upper left corner and the size of the square:

```cpp
class GSquare : public GRect {
public:
GSquare(double x, double y, double size) :
GRect(x, y, size, size) { /* Empty */ }
};
```