

Inheritance in C++

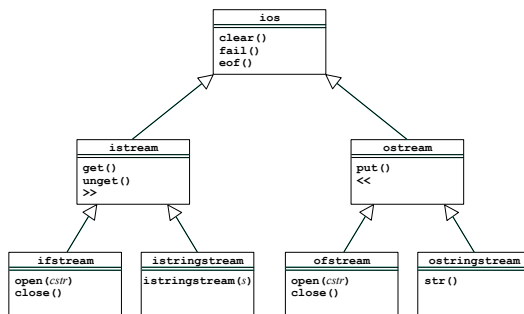
Inheritance in C++

Eric Roberts
CS 106B
February 27, 2015

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  
};
```

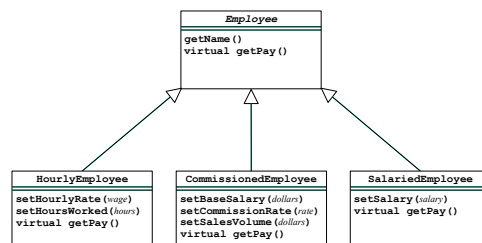
- You can use this feature to specify the types for a collection class, as in the following definition of `StringMap`:

```
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:

```
class Employee {
    virtual double getPay() = 0;
};

class HourlyEmployee : public Employee {
    virtual double getPay();
};

class CommissionedEmployee : public Employee {
    virtual double getPay();
};

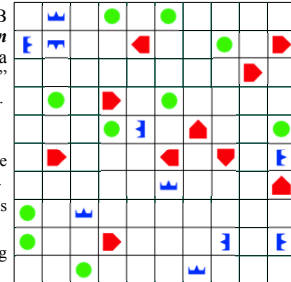
class SalariedEmployee : public Employee {
    virtual double getPay();
};
```

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:

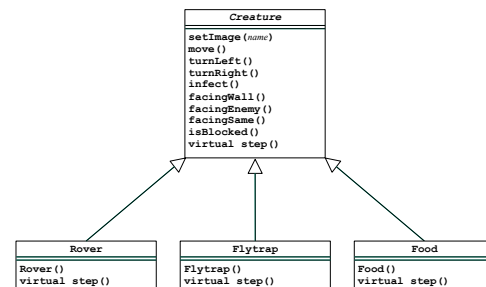
```
void Rover::step() {
    if (facingEnemy()) {
        infect();
    } else if (isBlocked()) {
        if (random()) {
            turnLeft();
        } else {
            turnRight();
        }
    } else {
        move();
    }
}
```

```
void Flytrap::step() {
    if (facingEnemy()) {
        infect();
    } else {
        turnLeft();
    }
}
```

```
void Food::step() {
    /* Empty */
}
```

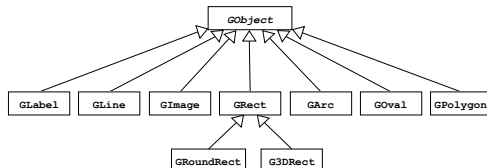
- Because the definition of **step** is different in each subclass, this method must be virtual.

The Creature Hierarchy



Representing Graphical Shapes

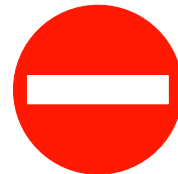
- In CS 106A, you learned how to use the **GObject** hierarchy in the **acm.graphics** package, which looks something like this:



- The **gobjects.h** interface includes all these classes. Chapter 19, however, implements just a few of them.
- In C++, the most important thing to keep in mind is that you have to use **pointers** to these objects.

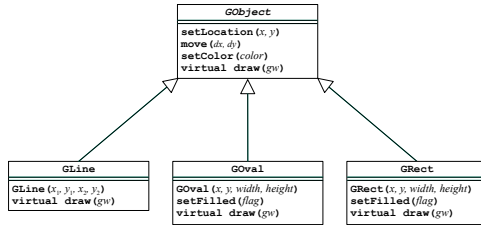
Exercise: Do Not Enter

- 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

```

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

#ifndef _gobjects_h
#define _gobjects_h

#include <string>
#include "gwindow.h"

/*
 * Class: GObject
 * -----
 * This class is 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 GObject {
public:

```

The gobjects.h Interface

```

/*
 * Method: setLocation
 * Usage: gobj->setLocation(x, y);
 * -----
 * Sets the x and y coordinates of gobj to the specified values.
 */

void setLocation(double x, double y);

/*
 * Method: move
 * Usage: gobj->move(dx, dy);
 * -----
 * Adds dx and dy to the coordinates of gobj.
 */

void move(double x, double y);

/*
 * Method: setColor
 * Usage: gobj->setColor(color);
 * -----
 * Sets the color of gobj.
 */

void setColor(std::string color);

```

The gobjects.h Interface

```

/*
 * Abstract method: draw
 * Usage: gobj->draw(gw);
 * -----
 * Draws the graphical object on the GraphicsWindow specified by gw.
 * This method is implemented by the specific GObject subclasses.
 */

virtual void draw(GWindow & gw) = 0;

protected:

/* The following methods and fields are available to the subclasses */

GObject(); // Superclass constructor */
std::string color; // The color of the object */
double x, y; // The coordinates of the object */

};

```

The gobjects.h Interface

```

/*
 * Subclass: GLine
 * -----
 * The GLine subclass represents a line segment on the window.
 */

class GLine : public GObject {
public:
/*
 * Constructor: GLine
 * Usage: GLine *lp = new GLine(x1, y1, x2, y2);
 * -----
 * Creates a line segment that extends from (x1, y1) to (x2, y2).
 */

GLine(double x1, double y1, double x2, double y2);

/* Prototypes for the overridden virtual methods */
virtual void draw(GWindow & gw);

private:
double dx; // Horizontal distance from x1 to x2 */
double dy; // Vertical distance from y1 to y2 */
};

```

The gobjects.h Interface

```

class GRect : public GObject {
public:
/*
 * Constructor: GRect
 * Usage: GRect *rp = new GRect(x, y, width, height);
 * -----
 * Creates a rectangle of the specified size and upper left corner at (x, y).
 */

GRect(double x, double y, double width, double height);

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

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

```
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

```
/*
 * Implementation notes: GObject class
 * -----
 * The constructor for the superclass sets all graphical objects to BLACK,
 * which is the default color.
 */
GObject::GObject() {
    setColor("BLACK");
}

void GObject::setLocation(double x, double y) {
    this->x = x;
    this->y = y;
}

void GObject::move(double dx, double dy) {
    x += dx;
    y += dy;
}

void GObject::setColor(string color) {
    this->color = color;
}
```

Implementation of the GLine Class

```
/*
 * Implementation notes: GLine class
 * -----
 * The constructor for the GLine class has to change the specification
 * of the line from the endpoints passed to the constructor to the
 * representation that uses a starting point along with dx/dy values.
 */
GLine::GLine(double x1, double y1, double x2, double y2) {
    this->x = x1;
    this->y = y1;
    this->dx = x2 - x1;
    this->dy = y2 - y1;
}

void GLine::draw(GWindow & gw) {
    gw.setColor(color);
    gw.drawLine(x, y, x + dx, y + dy);
}
```

Implementation of the GRect Class

```
GRect::GRect(double x, double y, double width, double height) {
    this->x = x;
    this->y = y;
    this->width = width;
    this->height = height;
    filled = false;
}

void GRect::setFilled(bool flag) {
    filled = flag;
}

void GRect::draw(GWindow & gw) {
    gw.setColor(color);
    if (filled) {
        gw.fillRect(x, y, width, height);
    } else {
        gw.drawRect(x, y, width, height);
    }
}
```

Implementation of the GOval Class

```
GOval::GOval(double x, double y, double width, double height) {
    this->x = x;
    this->y = y;
    this->width = width;
    this->height = height;
    filled = false;
}

void GOval::setFilled(bool flag) {
    filled = flag;
}

void GOval::draw(GWindow & gw) {
    gw.setColor(color);
    if (filled) {
        gw.fillOval(x, y, width, height);
    } else {
        gw.drawOval(x, y, width, height);
    }
}
```

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:

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