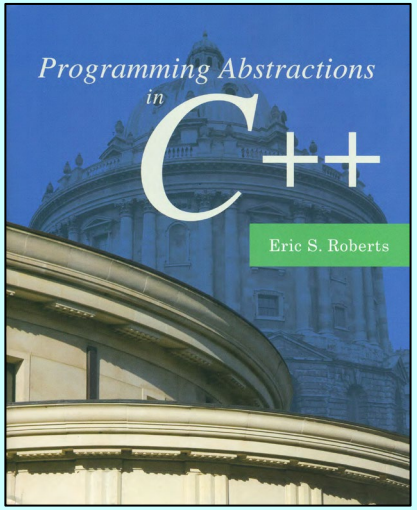
CHAPTER 19

Inheritance

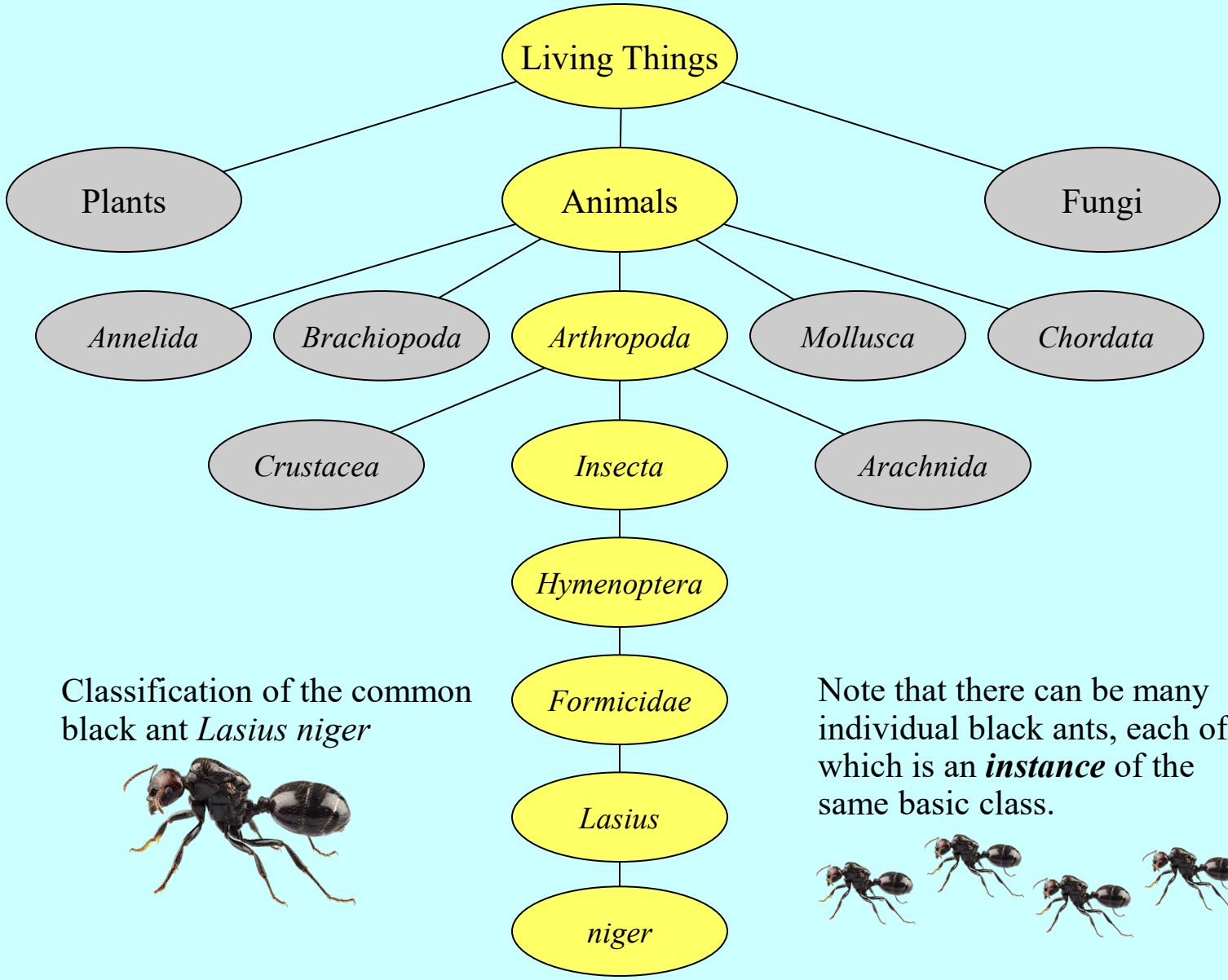
Beware how you trifle with your marvelous inheritance.

—Henry Cabot Lodge, "League of Nations" 1919



Flashback

Biological Class Hierarchy

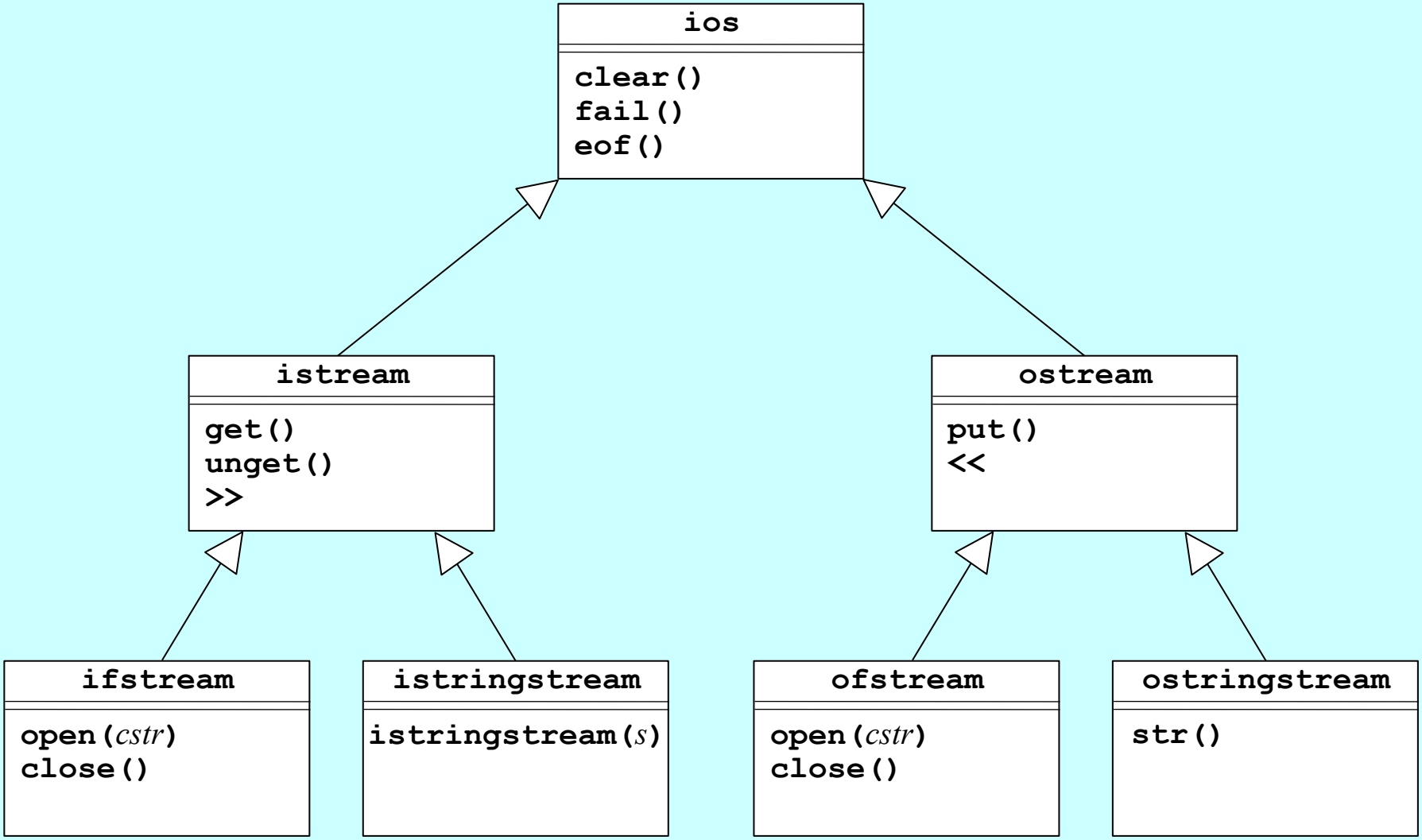


Classification of the common black ant *Lasius niger*

Note that there can be many individual black ants, each of which is an ***instance*** of the same basic class.

Simplified View of the Stream Hierarchy

Flashback



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*** (derived class) of some other class, which is called its ***superclass*** (base class).
- Each subclass represents a (usually more complicated) ***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.
- superclass : subclass ≠ superset : subset (e.g., rectangle : square)
- 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***.

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
};
```

can be **protected** *and* **private** *too.*

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

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

- This strategy is useful in the **graph** class, because it lets you define a **AirlineGraph** class with specific node and arc types:

```
class AirlineGraph : public Graph<City,Flight> {
    additional operations you want for your graph
};
```

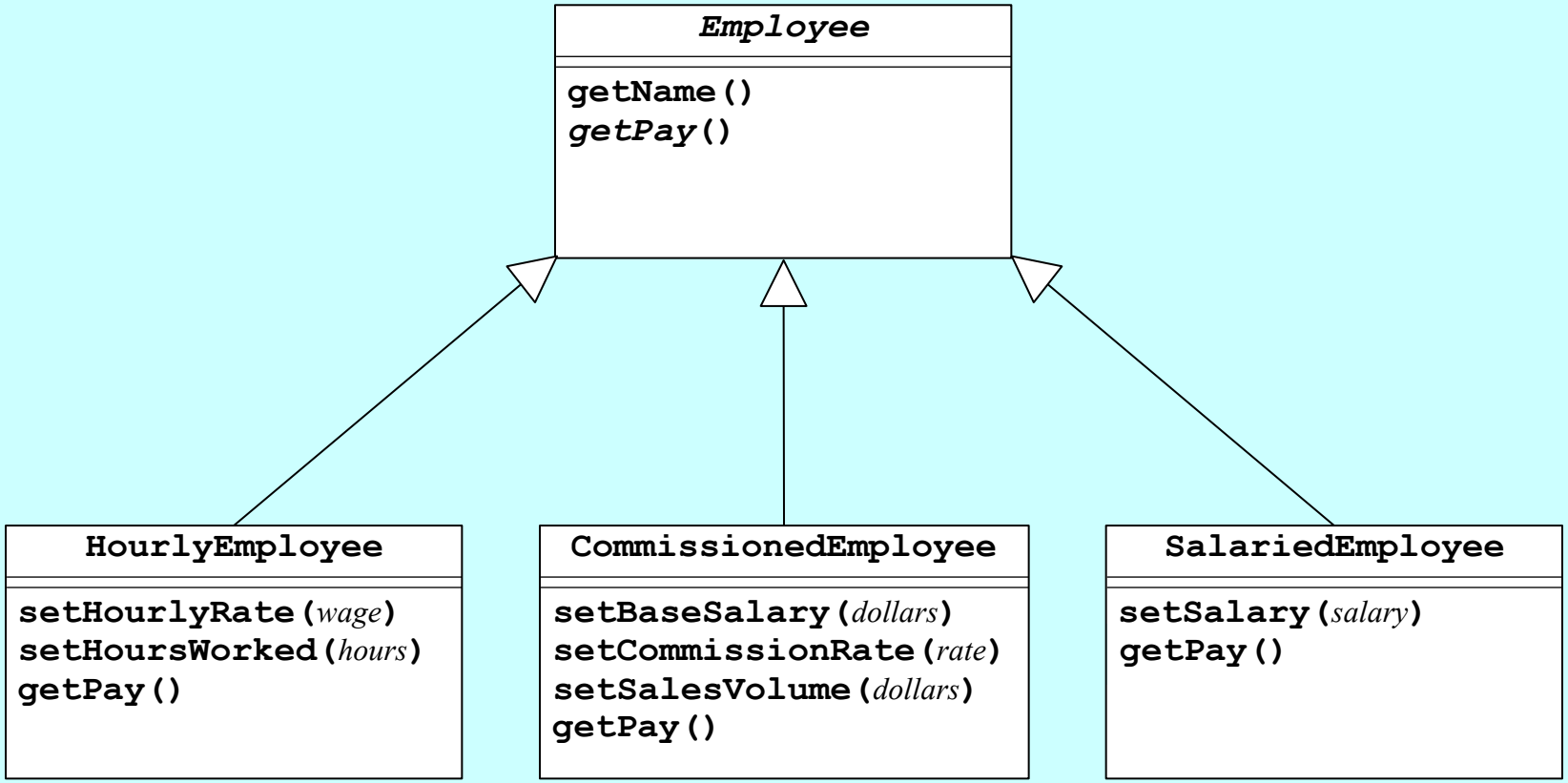
Representing Inheritance in C++

- Subclasses have access to the public and protected (but not private) members/methods in the superclass.
- ***public***, ***protected***, and ***private*** inheritance have the following features:

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

`protected` *and* **`private`** *inheritance is out of the scope of this course. We will mostly examine* **`public`** *inheritance.*

Example: the Employee Hierarchy



- In the **`Employee`** hierarchy, **`getPay`** is ***overridden*** differently in each subclass and therefore is a ***virtual method***.

Polymorphism: Overloading vs. Overriding

- ***Overloading*** is about multiple functions/operators/methods of the same name but different signatures and possibly different implementations.
- ***Overriding*** is about multiple methods of the same signature but different implementations defined in different classes connected through inheritance.
- Method ***overloading*** is an example of ***static*** (or ***compile time***) ***polymorphism***, while method ***overriding*** is an example of ***dynamic*** (or ***run time***) ***polymorphism***.

Virtual Methods

```
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();
};
```

- Once a method is declared as a virtual method, it becomes virtual in every subclass. In other words, it is not necessary to use the keyword virtual in the subclasses.
- Any method that is always implemented by a concrete subclass is indicated by including `= 0` before the semicolon on the prototype line, to mark the definition of a ***pure virtual method***.

Abstract Classes

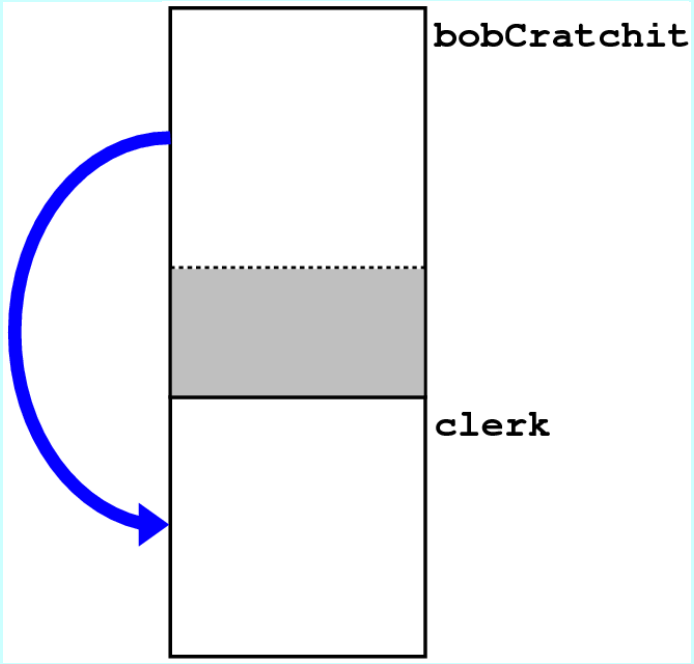
- Because of the pure virtual method **`getPay`**, the superclass **`Employee`** cannot be instantiated, and therefore is an ***abstract class*** that is never created on its own but instead serves as a common superclass for other ***concrete classes*** that correspond to actual objects. E.g., every **`Employee`** object must instead be constructed as an **`HourlyEmployee`**, a **`CommissionedEmployee`**, or a **`SalariedEmployee`**.
- You cannot declare an object of an abstract class because some of the methods may not be properly implemented (e.g., pure virtual methods). But you can declare a pointer of an abstract class, which can be later used to point to an object of a concrete subclass inheriting from the abstract superclass.

The limitation of subclassing in C++

- In Python, 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 a subclass always requires at least as much space as its superclass, and typically requires more. C++ throws away any fields in the assigned object that don't fit into the superclass by default. This behavior is called ***slicing***. (More serious a problem than shallow copy!)

```
HourlyEmployee bobCratchit;
Employee clerk;
clerk = bobCratchit;
```

If **Employee** *is an abstract class, it cannot be instantiated. Assuming it is not, slicing will happen. This is still incorrect.*



Avoid Slicing in C++

- To avoid slicing, one approach is to define private versions of the copy constructor and assignment operator so that copying objects in that inheritance hierarchy is prohibited (just like the stream class hierarchy in C++).
- Although this strategy eliminates the possibility of losing data through slicing, disallowing assignment makes it harder to embed objects in larger data structures.
- If you don't prohibit assignment, however, clients must then assume the responsibility for memory management. More often than not, clients do a terrible job in this regard.

The limitation of subclassing in C++

- It is also impossible to declare a collection of **`Employee`** and then **`add`** different types of objects of its subclasses (e.g., **`HourlyEmployee`**) into the collection because of slicing.
- By contrast, it is always legal to assign ***pointers*** to objects, and all pointers are the same size. Therefore, to avoid slicing, one way is to use pointers rather than the objects themselves.

```
Vector<Employee> payroll;
payroll.add(bobCratchit);
```

```
Vector<Employee *> payroll;
payroll.add(&bobCratchit);
for (Employee *ep : payroll) {
   cout << ep->getName() << ": " << ep->getPay() << endl;
}
```

- Question: Assuming **`Employee`** is not an abstract class, **`ep`** is a pointer of **`Employee`** pointing to **`HourlyEmployee`**, which **`getPay()`** method will be called by **`ep->getPay()`**?

The limitation of subclassing in C++

- In Python, defining a subclass method automatically overrides the definition of that method in its superclass (no explicit marking is needed). In C++, a pointer can be declared as the superclass but pointed to the subclass, you have to explicitly allow for overriding by marking the method prototype with the keyword **`virtual`**.
- More precisely, if you leave out the **`virtual`** keyword, the compiler determines which version of a method to call based on how the object is declared and not on how the object is constructed. If a pointer is declared as the superclass but pointed to the subclass, and a method is called using this pointer, the superclass method will be called if it's non-virtual, but the overridden method in the subclass will be called if it's marked as virtual in the superclass.

```
class A {
public:
    int a = 1;
    void display() { cout << 'A' << a << endl; }
};

class B: public A {
public:
    int b = 2;
    void display() {
        cout << 'B' << a << b << endl;
    }
};
class C: public B {
public:
    int c = 3;
    virtual void display() {
        cout << 'C' << a << b << c << endl;
    }
};
class D: public C {
public:
    int d = 4;
    void display() {
        cout << 'D' << a << b << c << d << endl;
    }
};
```

```
int main() {
    A oA;
    oA.display();
    B oB;
    oB.display();
    C oC;
    oC.display();
    D oD;
    oD.display();
    oA = oB;
    oA.display();
    oC = oD;
    oC.display();
    A* pA = &oB;
    pA->display();
    C* pC = &oD;
    pC->display();
}
```

```
A1
B12
C123
D1234
A1
C123
A1
D1234
```

Flashback

Three Strategies for Graph Abstraction

1. ***Use low-level structures***. This design uses the structure types `Node` and `Arc` to represent the components of a graph. This model gives clients complete freedom to extend these structures but offers no support for graph operations.
2. ***Adopt a hybrid strategy***. This design defines a `Graph` class as a parameterized class using templates so that it can use any structures or objects as the node and arc types. This strategy retains the flexibility of the low-level model and avoids the complexity associated with the class-based approach.
3. ***Define classes for each of the component types***. This design uses the `Node` and `Arc` classes to define the structure. In this model, clients define subclasses of the supplied types to particularize the graph data structure to their own application. More will be discussed later when we learn inheritance.

Three Strategies for Graph Abstraction

- In most object-oriented languages, the usual strategy to use in situations of this kind is ***subclassing***.
- Under this model, the graph package itself would define base classes for nodes and arcs that contain the information necessary to represent the graph structure. Clients needing to include additional data would do so by extending these base classes to create new subclasses with additional fields and methods.
- For reasons that you will have a chance to explore more fully in the extended discussion of inheritance in Chapter 19, this approach is not ideal for C++.
- The fundamental problem is that dynamic memory allocation and inheritance don't always work together in C++ as seamlessly as they do in other languages, to the point that it is worth adopting a different approach.

Representing Inheritance in C++

```
class Node;
class Arc;
class Node {
   string name;
   Set<Arc *> arcs;
};
class Arc {
   Node *start;
   Node *finish;
};
class Graph {
   Set<Node *> nodes;
   Set<Arc *> arcs;
   Map<std::string,Node *> nodeMap;
};
```

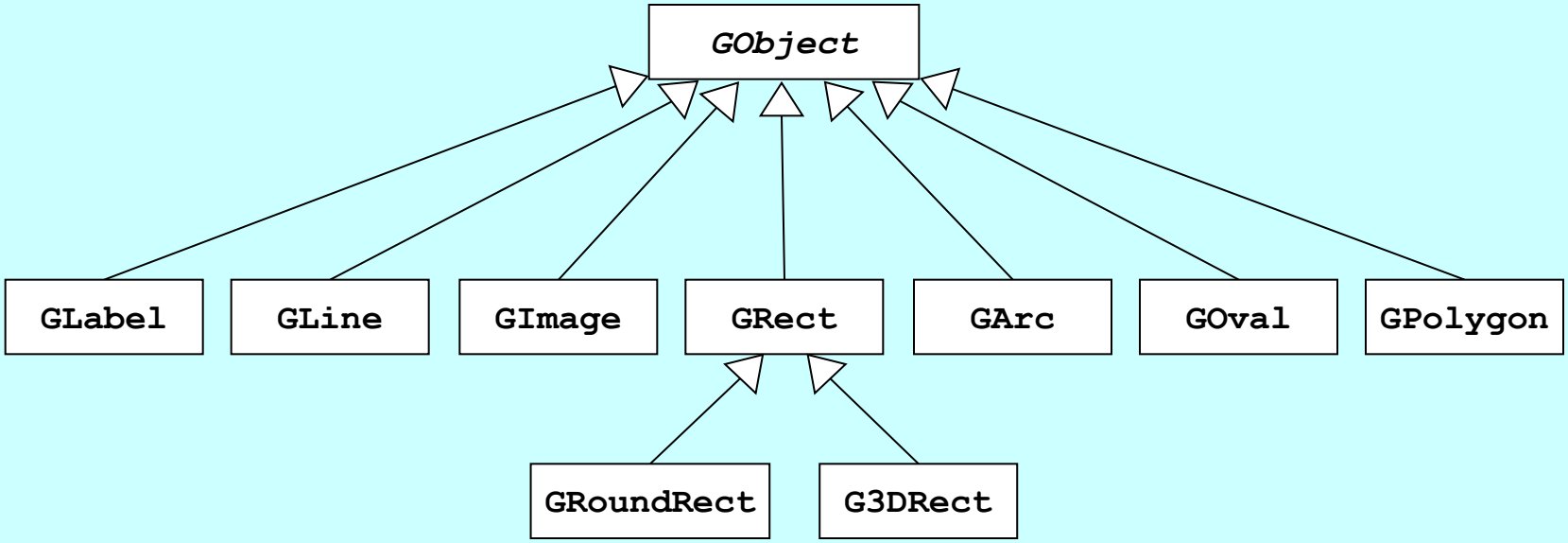
```
class City : public Node {
   additional members and operations you want for your graph node
};
class Flight : public Arc {
   additional members and operations you want for your graph arc
};
class AirlineGraph : public Graph {
   additional members and operations you want for your graph
}
```

The limitation of subclassing in C++

- Unfortunately, using pointers complicates the process of memory management, which is already a difficult challenge in C++. As long as the implementation takes care responsibility in the form of destructors, it is possible to keep the nightmares of memory management under control. As soon as you let pointers cross the boundary to the client side of the interface, the overall complexity skyrockets.
- The designers of the C++ libraries made different choices in different situations. The collection classes are implemented as independent classes that do not form an inheritance hierarchy. The stream classes, by contrast, form a sophisticated hierarchy but do not allow assignment. In each case, the designers chose to keep the details of memory management hidden from the client.
- Collections allow deep copying (by overloading assignment operator and copy constructor) while streams forbid copying.

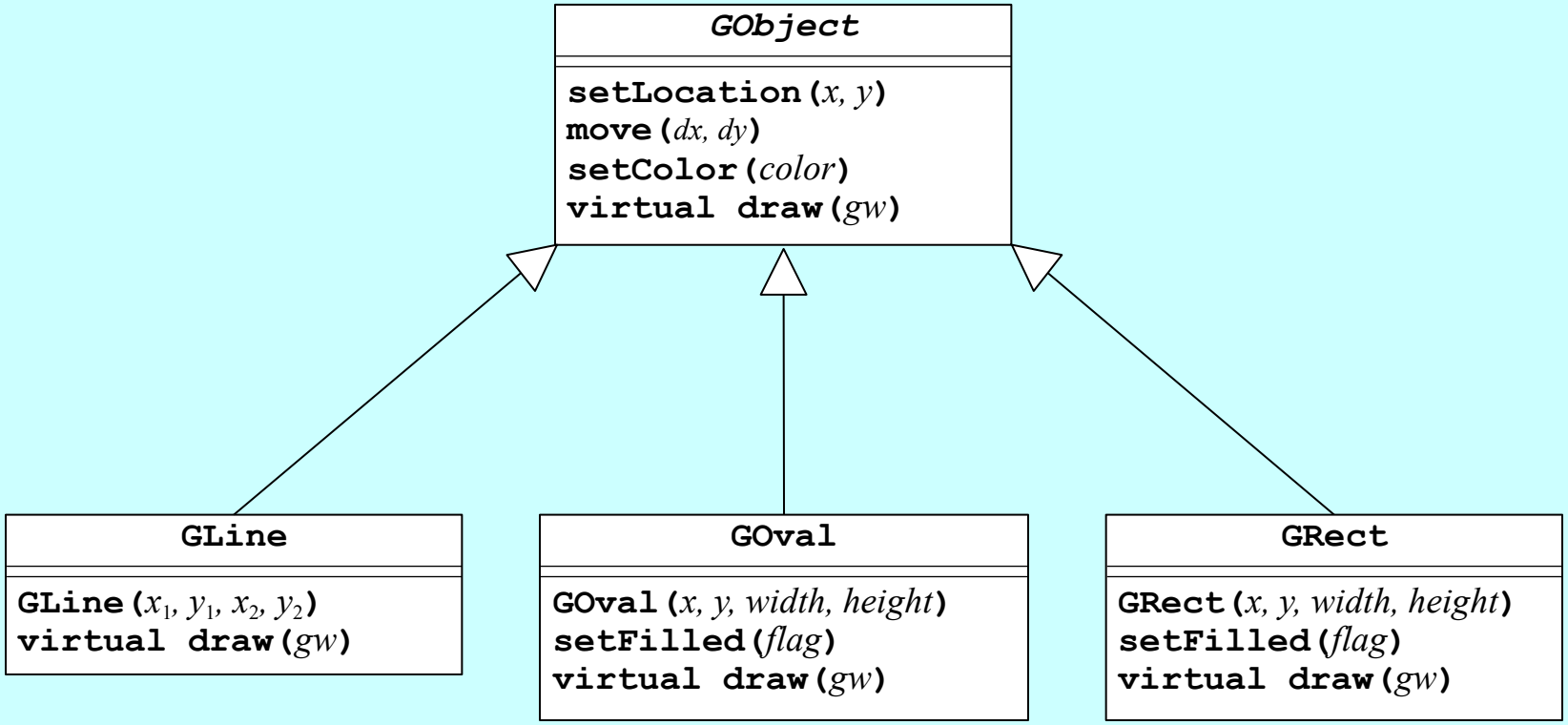
Representing Graphical Shapes

- In the Stanford C++ library, you can find the **`GObject`** hierarchy, which looks something like this:



- The **`gobjects.h`** interface includes all these classes. The text, 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.

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  */

};
```

Methods and fields in the ***protected*** *section are accessible to any subclasses but remain off-limits to clients.*

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

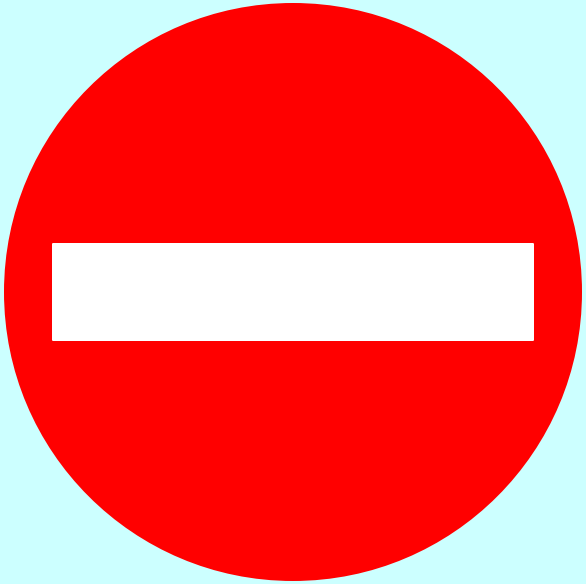
```
GRect::GRect(double x, double y, double width, double height) {
   this->x = x;
   this->y = y;
   this->width = width;
   this->height = height;
   filled = false;
}

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);
   }
}
```

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.

Initializer List

- An ***initializer list*** is sometimes used in initializing the data members of a class. The initializer list appears just before the brace that begins the body of the constructor and is set off from the parameter list by a colon. The elements of the initializer list should be in one of the following forms:
  - The name of a field in the class, followed by an initializer for that field enclosed in parentheses.
  - A superclass constructor (will be discussed later with ***inheritance***).

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 is the second form of the initializer list.
- 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 */
   }
};
```

This example is only used to illustrate initializer list. Generally square should not be a subclass of rectangle. Why?

Exercise: Construct `GCircle` Using `GOval`

```
class GCircle : public GOval {
   GCircle(double x, double y, double r)
                     : GOval(x-r, y-r, 2*r, 2*r) {
      /* Empty */
   }
};
```

Storing `GObject` pointers in a vector

- One of the advantages of defining a `GObject` class hierarchy is that doing so makes it possible to store different graphical shapes (e.g., `GLine`, `GRect`, `GOval`) in collection classes, as long as you remember to use pointers to the objects rather than the objects themselves.

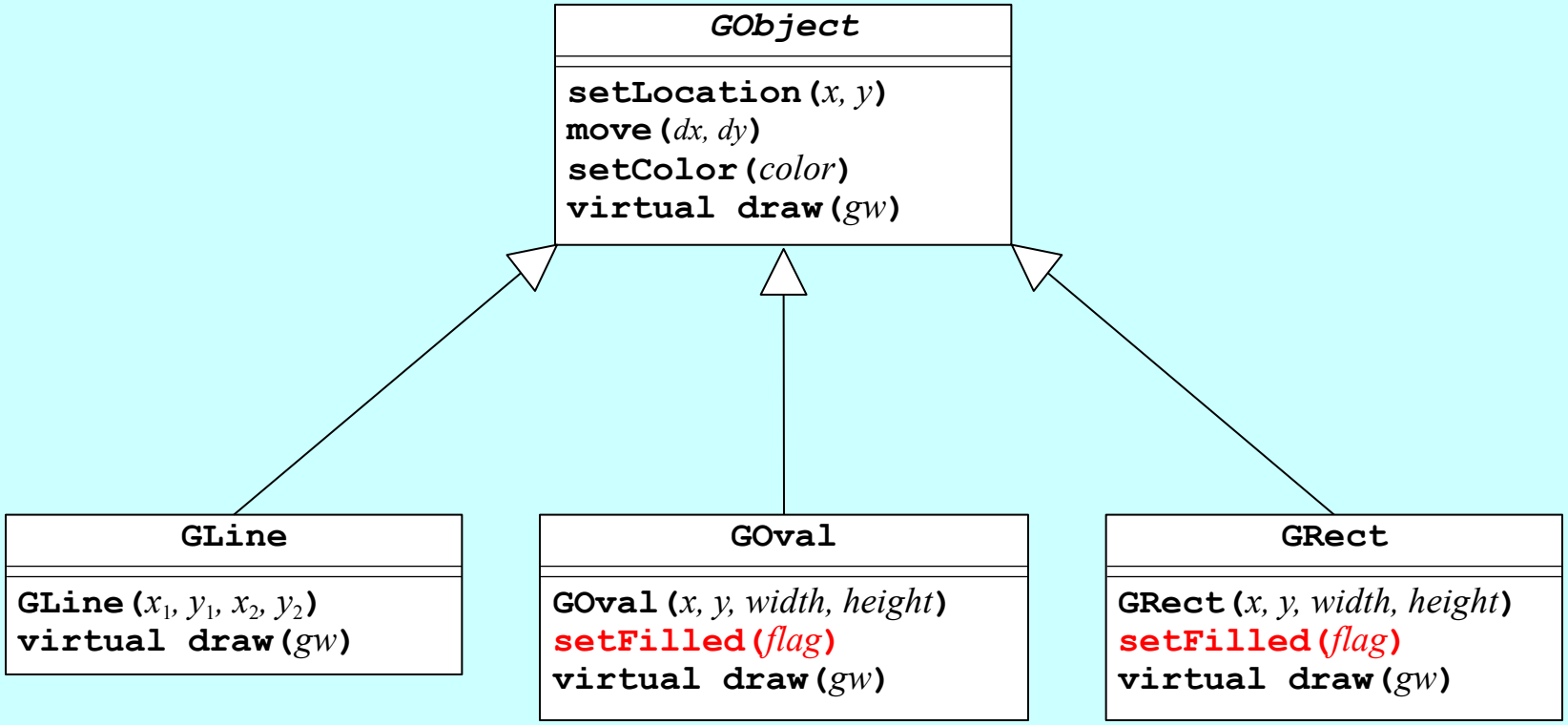
Storing `GObject` pointers in a vector

```
int main() {
   GWindow gw;
   double width = gw.getWidth();
   double height = gw.getHeight();
   GRect *rp = new GRect(width / 4, height / 4, width / 2, height / 2);
   GOval *op = new GOval(width / 4, height / 4, width / 2, height / 2);
   rp->setColor("BLUE");
   op->setColor("GRAY");
   Vector<GObject *> displayList;
   displayList.add(new GLine(0, height / 2, width / 2, 0));
   displayList.add(new GLine(width / 2, 0, width, height / 2));
   displayList.add(new GLine(width, height / 2, width / 2, height));
   displayList.add(new GLine(width / 2, height, 0, height / 2));
   displayList.add(rp);
   displayList.add(op);
   for (GObject *sp : displayList) {
      sp->draw(gw);
   }
   for (GObject *sp : displayList) {
      delete sp;
   }
   displayList.clear();
   return 0;
}
```

Multiple Inheritance

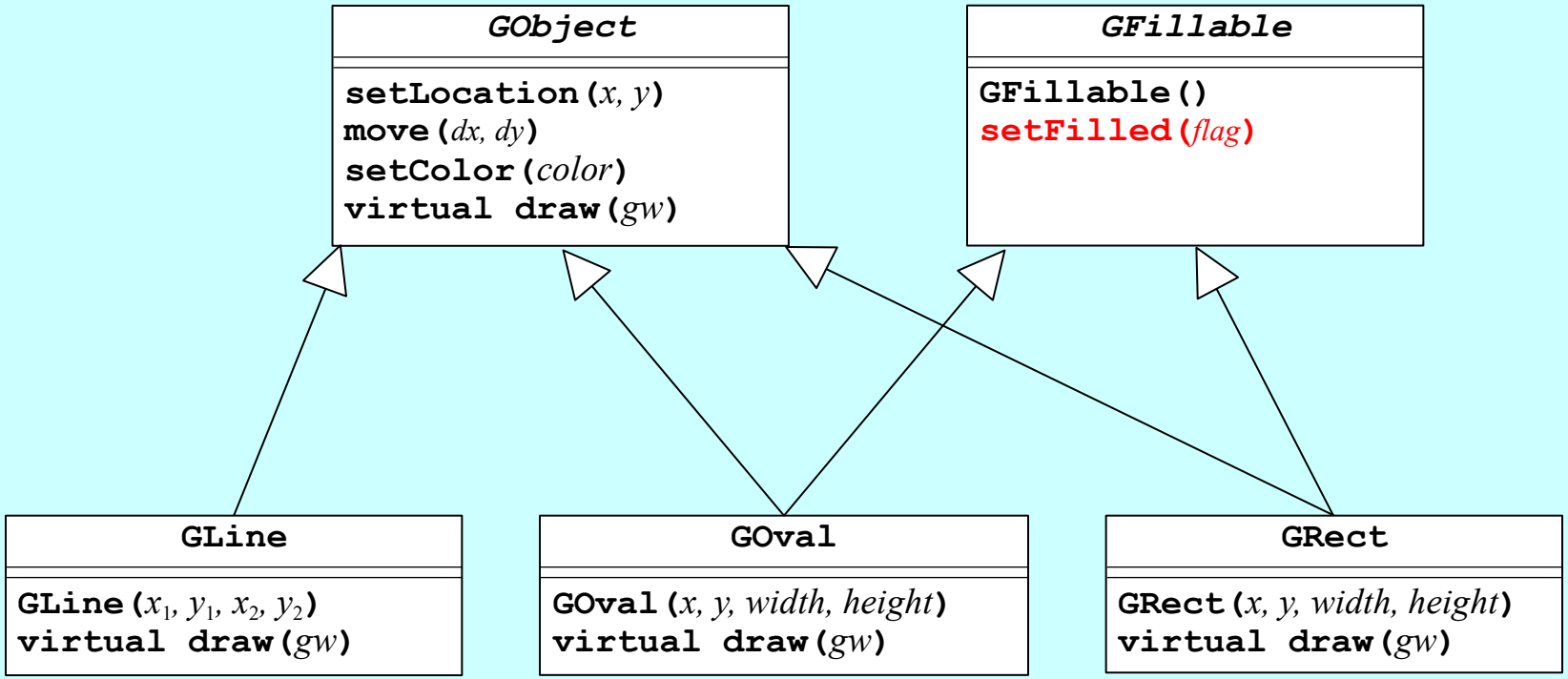
- Unlike many object-oriented languages, C++ allows one class to inherit behavior from more than one superclass. This technique is called ***multiple inheritance***.
- As an example of how multiple inheritance might prove useful in your own code, it is worth returning to the **`GObject`** hierarchy. As that class hierarchy currently exists, the **`GRect`** and **`GOval`** classes have duplicated **`setFilled`** methods that let clients specify whether a shape should be outlined or solid. By default, shapes are drawn as outlines, but clients can change that assumption by calling **`setFilled(true)`**.
- The interesting question is where to declare the **`setFilled`** method. It doesn't belong in the **`GObject`** class itself, because the **`GLine`** object can't be filled. On the other hand, declaring these methods in both the **`GRect`** and **`GOval`** classes requires duplicating those declarations. Multiple inheritance offers a way out of this dilemma.

The GObject Hierarchy



The GFillable Class

- If **GRect** and **GOval** also inherit from a class called **GFillable**, you can define the **setFilled** method there.
- The following UML diagram is a revised **GObject** hierarchy in which the **GOval** and **GRect** classes inherit from both the **GObject** class and a **GFillable** class that defines the behavior of fillable objects.

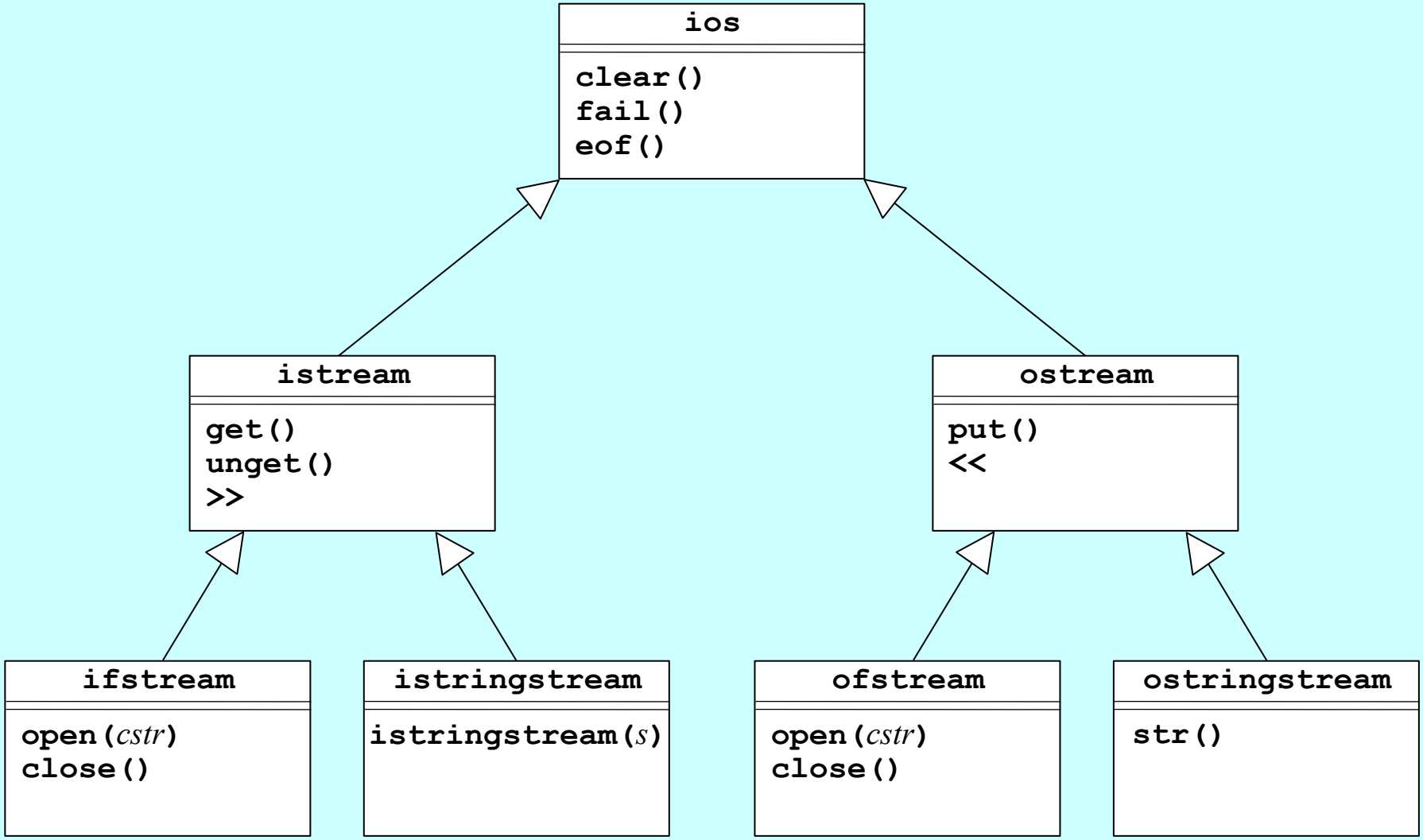


The `GFillable` Class

FIGURE 19-17 The `GFillable` class

```
class GFillable {

public:

/*
 * Constructor: GFillable
 * ----------------------
 * Ensures that shapes are created as outlines by default.
 */

   GFillable()     {
      fillFlag = false;
   }

/*
 * Method: setFilled
 * Usage: shape.setFilled(flag);
 * -----------------------------
 * Sets the fill status for shape, where false is outlined and true is filled.
 */

   void setFilled(bool flag) {
      fillFlag = flag;
   }

protected:

   bool fillFlag;        /* Flag is false for outline, true for solid fill */

};
```

Simplified View of the Stream Hierarchy



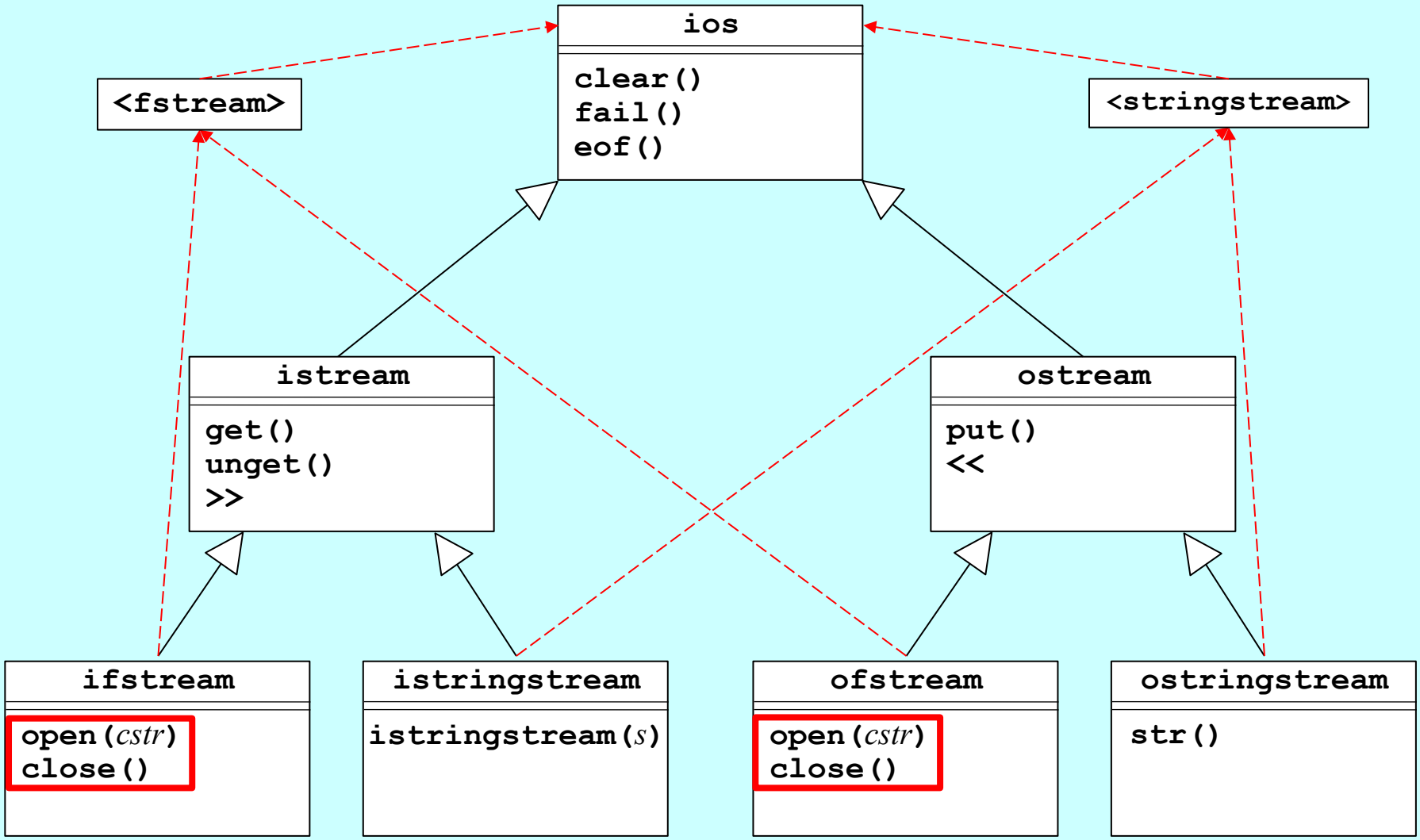
Copy Contents between Streams

```
int main() {
   ifstream infile;
   promptUserForFile(infile, "Input file: ");
   char ch;
   while (infile.get(ch)) {
      cout.put(ch);
   }
   infile.close();
   return 0;
}
void copyStream(istream & is, ostream & os) {
   char ch;
   while (is.get(ch)) {
      os.put(ch);
   }
}
void copyStream(ifstream & infile, ofstream & outfile) {
   char ch;
   while (infile.get(ch)) {
      outfile.put(ch);
   }
}
```

copyStream(infile, cout);

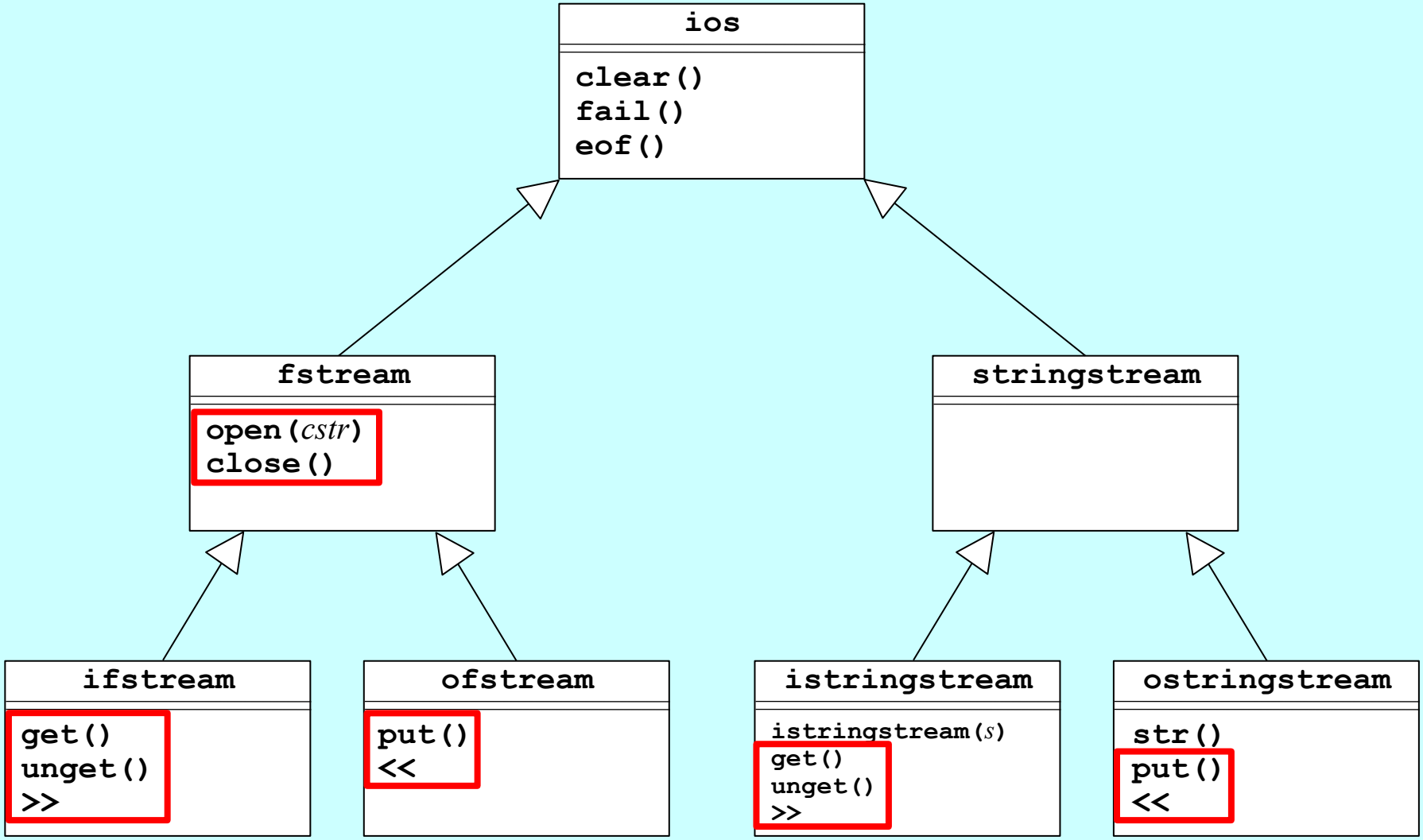
This version is obviously better because it can be directly used in the above program.

UML Diagram for the Stream Hierarchy



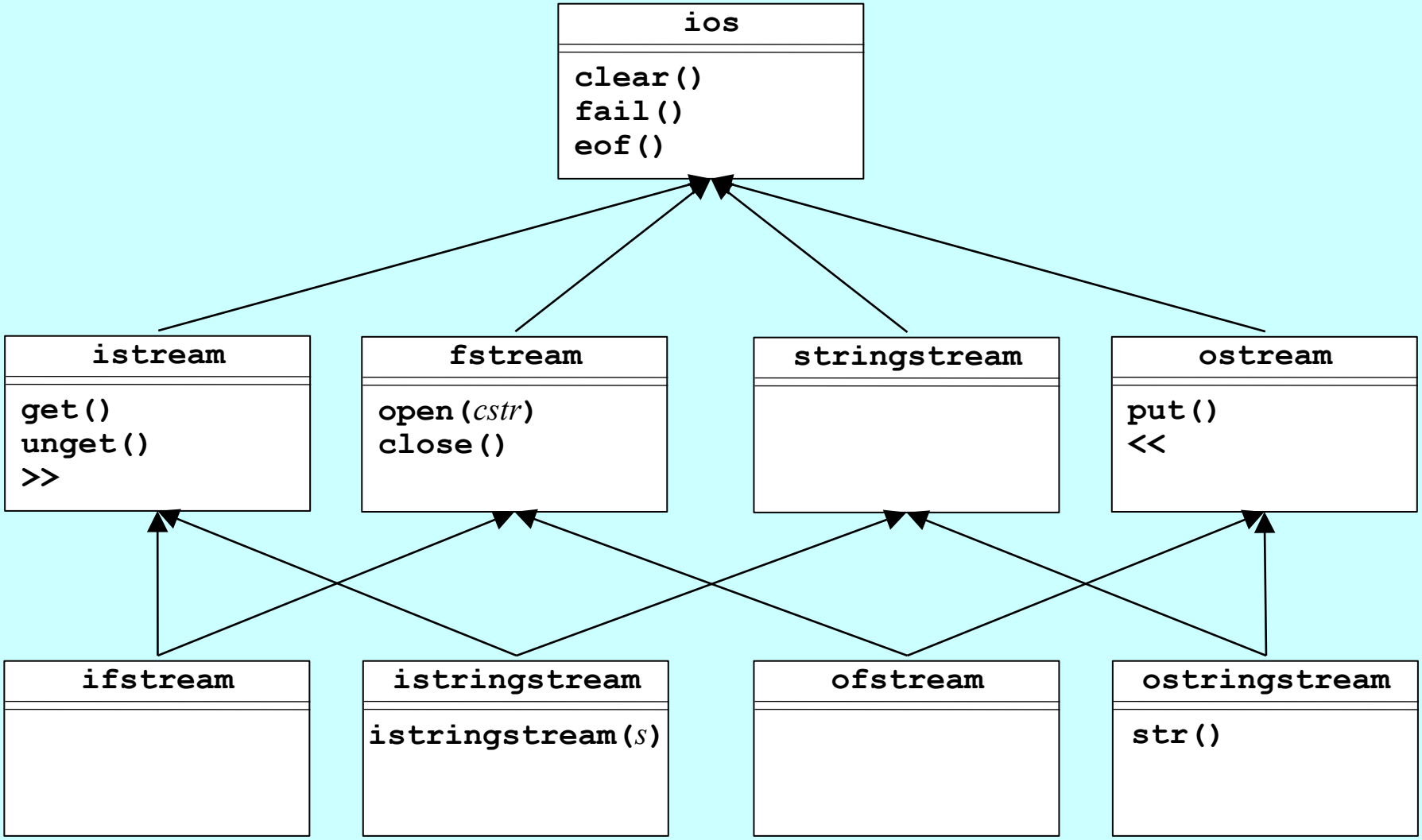
Question: Can we organize the hierarchy differently?

UML Diagram for an Alternative Hierarchy



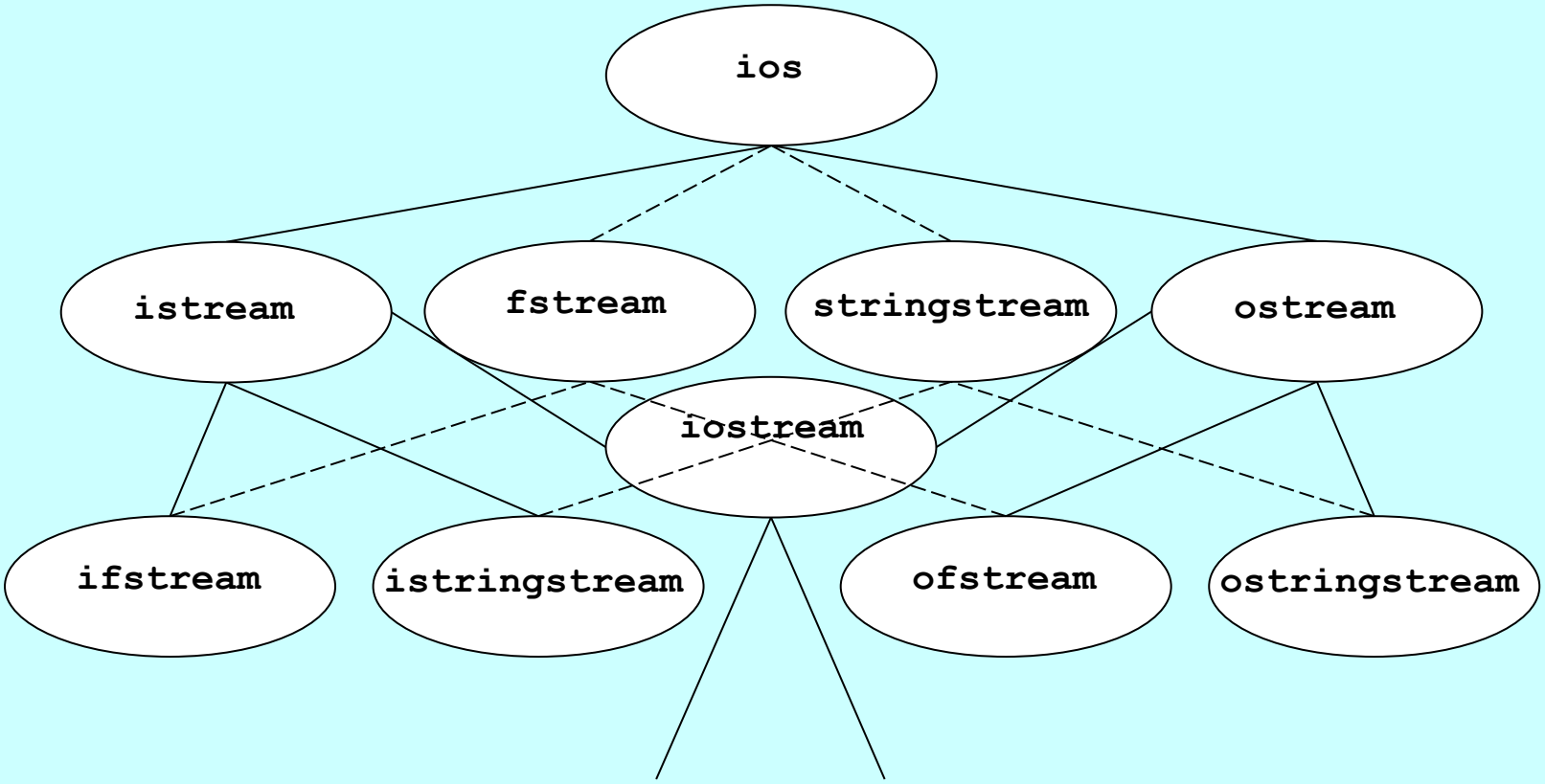
Question: Is this hierarchy a better one?

UML Diagram for Another Alternative



Question: Is this ***multiple inheritance*** design optimal?

Selected Classes in the Stream Hierarchy



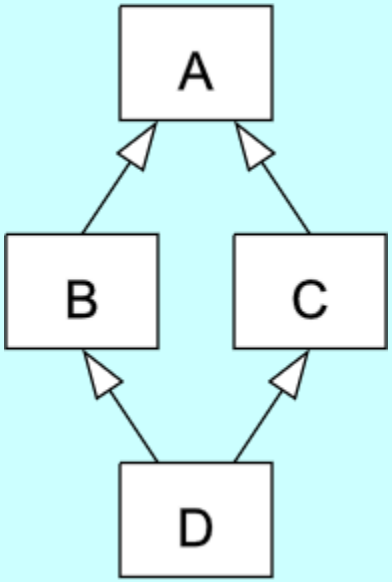
Multiple Inheritance in the stream libraries

- Chapter 4 introduces the stream class hierarchy, but stops short of describing those features that use multiple inheritance.
- As it happens, the stream libraries include classes that are both input and output streams (**`iostream`**). The previous slide updates the UML diagram from the previous one to show how these classes fit into the stream hierarchy as a whole.
- As an example of how you might use these bidirectional streams, write a function **`roundToSignificantDigits`** that takes a floating-point value **`x`** and rounds it to some specified number of significant digits.

```
double roundToSignificantDigits(double x, int nDigits) {
   stringstream ss;
   ss << setprecision(nDigits) << x;
   ss >> x;
   return x;
}
```

Multiple Inheritance

- Although multiple inheritance has its uses, it tends to create more problems than it solves.
- The "deadly diamond of death" is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?



```
class A {
public:
    int a = 1;
    void display() { cout << a << endl; }
};

class B: public A {
public:
    int b = 2;
    B() {a = 2;};
    void display() {
        cout << a << b << endl;
    }
};
class C: public A {
public:
    int c = 3;
    C() {a = 3;};
    void display() {
        cout << a << c << endl;
    }
};
class D: public B, public C {
public:
    int d = 4;
    void display() {
        cout << a << b << c << d << endl;
    }
};
```

'A' is an ambiguous base of 'D'

reference to 'a' is ambiguous

```
int main() {
    A oA;
    oA.display();
    B oB;
    oB.display();
    C oC;
    oC.display();
    D oD;
    oD.display();
    A* pA = &oD;
    pA->display();
    B* pB = &oD;
    pB->display();
    C* pC = &oD;
    pC->display();
}
```

```
1
22
33
234
22
33
```

Question: what's the output if **`display()`** is **`virtual`** in **`A`** or **`B`** or **`C`**?

Virtual Inheritance

- Different languages have different ways of dealing with these problems of multiple inheritance. C++ by default follows each inheritance path separately, so a D object would actually contain two separate A objects, and the uses of A's members have to be properly qualified.
- If the inheritance from A to B and the inheritance from A to C are both marked **`virtual`**, C++ takes special care to only create one A object, and the uses of A's members work correctly. E.g., in the stream hierarchy, **`ios`** is a virtual base of both **`istream`** and **`ostream`**.
- The values of A's members inherited by D are determined by the order of the constructor calls, which is from base to derived, and, in the case of multiple inheritance, from left to right as they appear in the class declaration.

```
class A {
public:
    int a = 1;
    void display() { cout << a << endl; }
};

class B: virtual public A {
public:
    int b = 2;
    B() {a = 2;};
    void display() {
        cout << a << b << endl;
    }
};
class C: virtual public A {
public:
    int c = 3;
    C() {a = 3;};
    void display() {
        cout << a << c << endl;
    }
};
class D: public B, public C {
public:
    int d = 4;
    void display() {
        cout << a << b << c << d << endl;
    }
};
```

```
int main() {
    A oA;
    oA.display();
    B oB;
    oB.display();
    C oC;
    oC.display();
    D oD;
    oD.display();
    A* pA = &oD;
    pA->display();
    B* pB = &oD;
    pB->display();
    C* pC = &oD;
    pC->display();
}
```

```
1
22
33
3234
3
32
33
```

Question: what's the output if **`display()`** is **`virtual`** in **`A`** or **`B`** or **`C`**?

The End