CS 106X, Lecture 26
Inheritance and Polymorphism

reading:

Programming Abstractions in C++, Chapter 19

Plan For This Week

- Graphs: Topological Sort (HW8)
- **Classes: Inheritance and Polymorphism (HW8)**
- Sorting Algorithms

Plan For Today

- Inheritance
- Composition
- Announcements
- Polymorphism

- **Learning Goal:** understand how to create and use classes that build on each other's functionality.

Plan For Today

- **Inheritance**
- Composition
- Announcements
- Polymorphism

Example: Employees

- Imagine a company with the following **employee class**:
  - All employees keep track of the number of years they have been working.
  - All employees work 40 hours / week.
  - All employees keep track of their name.

Employee class

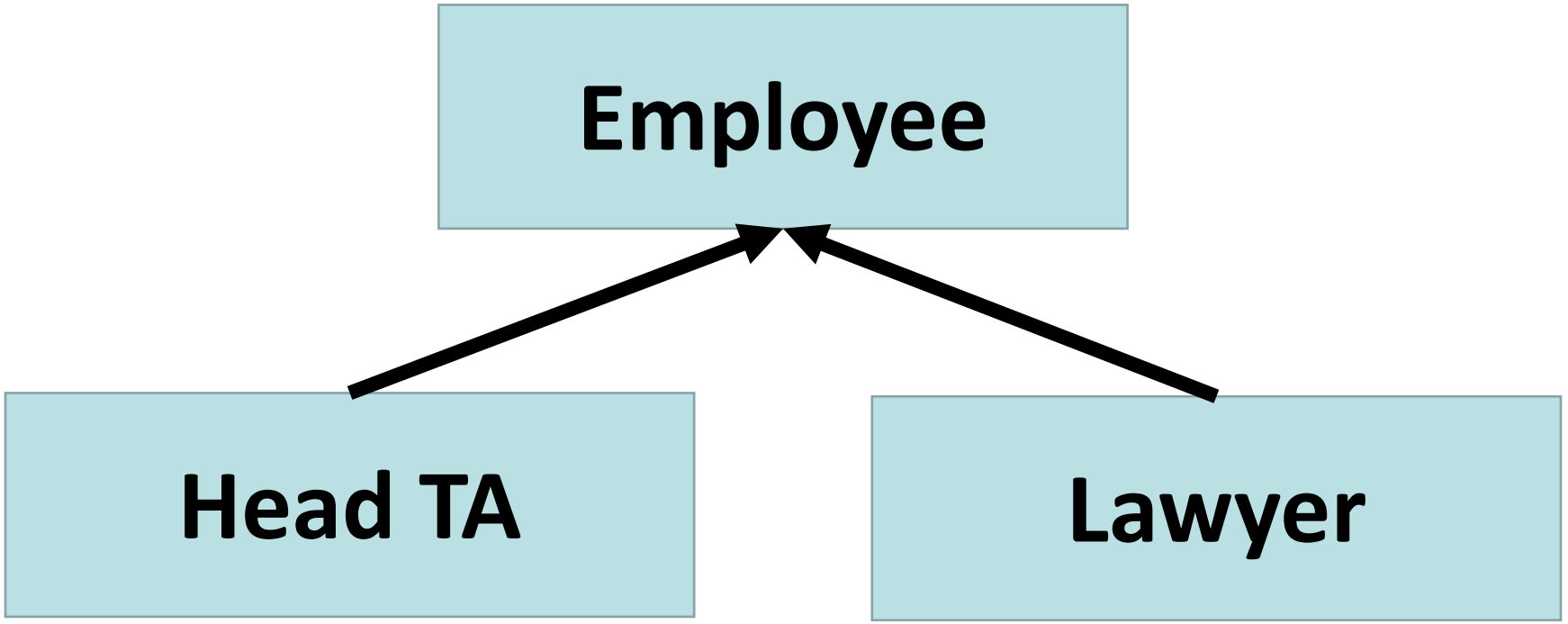
```
// Employee.h
class Employee {
public:
    Employee(string name,
             int yearsWorked);
    int getHoursWorkedPerWeek();
    string getName();
    int getYearsWorked();

private:
    string name;
    int yearsWorked;
};
```

```
// Employee.cpp
Employee::Employee(string name,
                   int yearsWorked) {
    this->name = name;
    this->yearsWorked = yearsWorked;
}

int Employee::getHoursWorkedPerWeek() {
    return 40;
}

string Employee::getName() {
    return name;
}


int Employee::getYearsWorked() {
    return yearsWorked;
}
```

Inheritance

Inheritance lets us relate our variable types to one another.

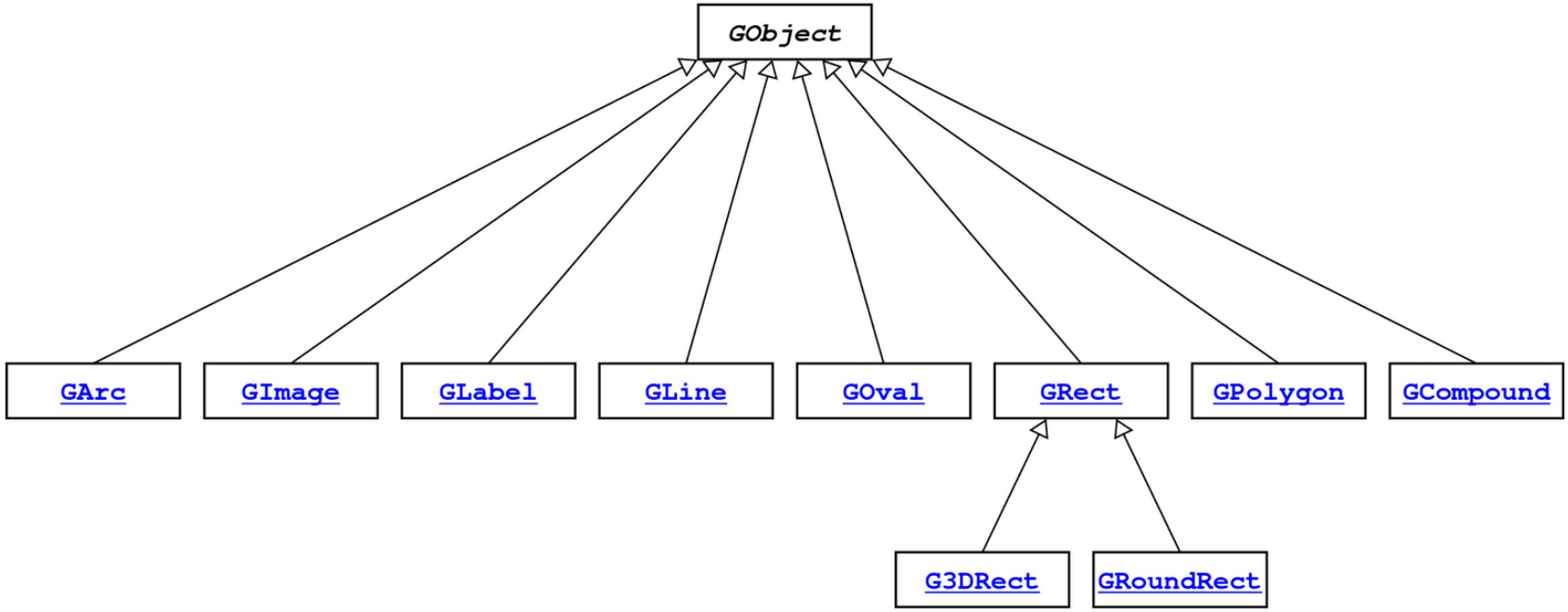
Inheritance



Variable types can seem to “inherit” from each other. We don’t want to have to duplicate code for each one!

Example: GObjects

- The Stanford library uses an inheritance hierarchy of graphical objects based on a common superclass named **`GObject`**.



Example: GObjects

- **`GObject`** defines the state and behavior common to all shapes:

```
contains(x, y)
getColor(), setColor(color)
getHeight(), getWidth(), getLocation(), setLocation(x, y)
getX(), getY(), setX(x), setY(y), move(dx, dy)
setVisible(visible), sendForward(), sendBackward()
toString()
```

- The subclasses add state and behavior unique to them:

`GLabel`	**`GLine`**	**`GPolygon`**
`get/setFont`	`get/setStartPoint`	`addEdge`
`get/setLabel`	`get/setEndPoint`	`addVertex`
		`get/setFillColor`
`...`	`...`	`..`

Using Inheritance

```
class Name : public Superclass {      // .h
```

– Example:

```
class Lawyer : public Employee {
    ...
}
```

- By extending Employee, this tells C++ that `Lawyer` can do **everything an Employee can do, plus more**.
- Lawyer automatically inherits all of the code from Employee!
- The **superclass** (or **base class**) is Employee, the **subclass** (or **derived class**) is Lawyer.

Example: Employees

- Lets implement **Lawyer**, that adds to the behavior of an **Employee** by keeping track of its clients (strings). You should be able to:
  - add a client for a Lawyer,
  - remove a client for a Lawyer,
  - get the number of clients.
  - Specify the law school when you create a Lawyer

Lawyer.h

```
class Lawyer : public Employee {
public:
    Lawyer(const string& name, int yearsWorked, const string&
lawSchool);
    void assignToClient(const string& clientName);
    void unassignToClient(const string& clientName);
    int getNumberOfClients() const;

private:
    int indexOfClient(const string& clientName) const;
    string lawSchool;
    Vector<string> clientNames;
};
```

Lawyer.cpp

```
void Lawyer::assignToClient(const string& clientName) {
	clientNames.add(clientName);
}

int Lawyer::getNumberOfClients() const {
	return clientNames.size();
}
```

Lawyer.cpp

```
int Lawyer::indexOfClient(const string& clientName) const {
	for (int i = 0; i < clientNames.size(); i++) {
		if (clientNames[i] == clientName) {
			return i;
		}
	}
	return -1;
}

void Lawyer::unassignToClient(const string& clientName) {
	int clientIndex = indexOfClient(clientName);
	if (clientIndex >= 0) {
		clientNames.remove(clientIndex);
	}
}
```

Call superclass c'tor

```
SubclassName::SubclassName(params)
        : SuperclassName(params) {
    statements;
}
```

- To call a superclass constructor from subclass constructor, use an *initialization list*, with a colon after the constructor declaration.
  - Example:

```
Lawyer::Lawyer(const string& name, int yearsWorked, const
string& lawSchool) : Employee(name, yearsWorked) {
    // calls Employee constructor first
    this->lawSchool = lawSchool;
}
```

Example: Employees

- Lets implement a **Head TA** class that adds to the behavior of an **Employee:**
- All employees work 40 hours / week.
  - Except for Head TAs who work half the hours (part-time)
- All employees report back their name.
  - Except for Head TAs who add “Head TA “ before it
- **Head TAs** have a favorite programming language.

Overriding

- **override**: To replace a superclass's member function by writing a new version of that function in a subclass.
- **virtual function**: One that is allowed to be overridden.
  - Must be declared with `virtual` keyword in superclass.

```
// Employee.h
virtual string getName();

// Employee.cpp
string Employee::getName() {
    return name;
}
```

```
// headta.h
string getName();

// headta.cpp
string HeadTA::getName() {
   // override!
}
```

Call superclass member

***SuperclassName*::*memberName*(*params*)**

- To call a superclass overridden member from subclass member.
  - Example:

```
int HeadTA::getHoursWorkedPerWeek() {      // part time
    return Employee::getHoursWorkedPerWeek() / 2;
}
```

  - Note: Subclass *cannot* access private members of the superclass.
  - Note: You only need to use this syntax when the superclass's member has been overridden.
    - If you just want to call one member from another, even if that member came from the superclass, you don't need to write `Superclass::` .

Pure virtual functions

```
virtual returntype name(params) = 0;
```

- **pure virtual function**: Declared in superclass's .h file and set to 0 (null). An absent function that has not been implemented.
  - *Must* be implemented by any subclass, or it cannot be used.
  - A way of forcing subclasses to add certain important behavior.

```
class Employee {
...
   virtual void work() = 0;  // every employee does
                             // some kind of work
};
```

  - FYI: In Java, this is called an *abstract method*.

Pure virtual base class

- **pure virtual base class**: One where every member function is declared as pure virtual. *(Also usually has no member variables.)*
  - Essentially not a superclass in terms of inheriting useful code.
  - But useful as a list of requirements for subclasses to implement.
  - Example: Demand that all shapes have an area, perimeter, # sides, ...

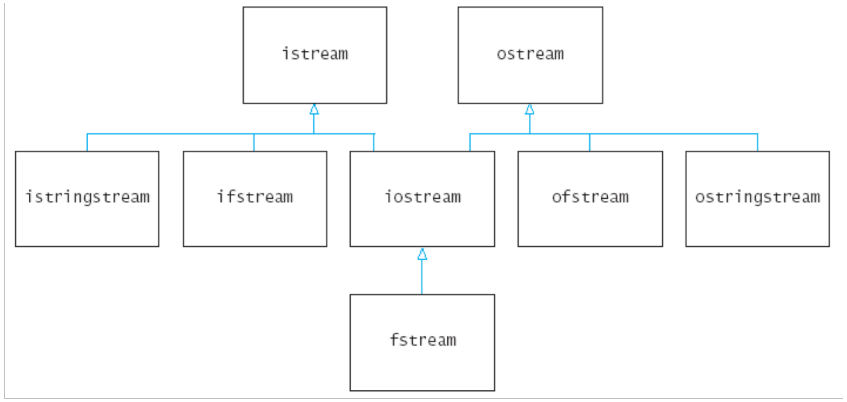
```
class Shape {   // pure virtual class; extend me!
    virtual double area() const = 0;
    virtual double perimeter() const = 0;
    virtual int sides() const = 0;
};
```

  - FYI: In Java, this is called an *interface*.

Multiple inheritance

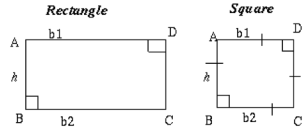
```
class Name : public Superclass1, public Superclass2, ...
```

- **multiple inheritance**: When one subclass has multiple superclasses.
  - Forbidden in many OO languages (e.g. Java) but allowed in C++.
  - Convenient because it allows code sharing from multiple sources.
  - Can be confusing or buggy, e.g. when both superclasses define a member with the same name.
  - Example: The C++ I/O streams use multiple inheritance:



Perils of inheritance

- Consider the following places you might use inheritance:
  - class **Point3D** extends Point2D and adds *z*-coordinate
  - class **Square** extends Rectangle (or vice versa?)
  - class **SortedVector** extends Vector, keeps it in sorted order



- What's wrong with these examples? Is inheritance good here?
  - Point2D's distance() function is wrong for 3D points
  - Rectangle supports operations a Square shouldn't (e.g. setWidth)
  - SortedVector might confuse client; they call insert at an index, then check that index, and the element they inserted is elsewhere!

Plan For Today

- Inheritance
- **Composition**
- Announcements
- Polymorphism

Composition

- **Composition** is an alternative to inheritance; instead of inheriting a class, you have an *instance* (or instances) of that class as an instance variable.
- **E.g.** SortedVector contains a Vector.
- **Is-a** vs. **Has-a**

Plan For Today

- Inheritance
- Composition
- **Announcements**
- Polymorphism

Announcements

- **HW8** (106XCell) goes out later today!
  - **No late submissions will be accepted**

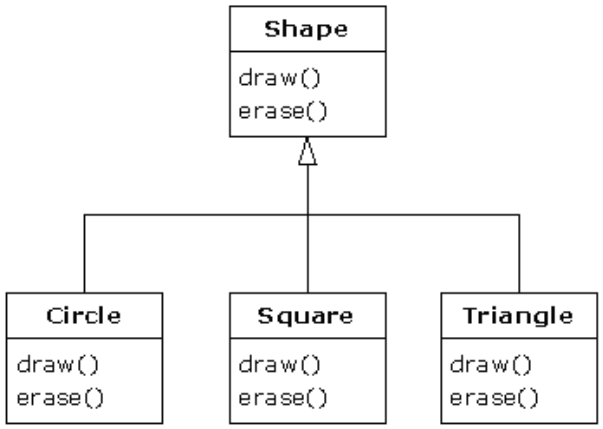
Plan For Today

- Inheritance
- Composition
- Announcements
- **Polymorphism**

Polymorphism

- **polymorphism**: Ability for the same code to be used with different types of objects and behave differently with each.
  - Templates provide *compile-time* polymorphism.
    Inheritance provides *run-time* polymorphism.

- *Idea:* Client code can call a method on different kinds of objects, and the resulting behavior will be different.



Poly. and pointers

- A pointer of type *T* can point to any subclass of *T*.

```
Employee*  edna  = new Lawyer("Edna", "Harvard", 5);
Secretary* steve = new LegalSecretary("Steve", 2);
World* world     = new WorldMap("map-stanford.txt");
```

  - When a member function is called on `edna`, it behaves as a `Lawyer`.
    - (This is because the employee functions are declared `virtual`.)
    - You can *not* call any `Lawyer`-only members on `edna` (e.g. `sue`).
      You can *not* call any `LegalSecretary`-only members on `steve` (e.g. `fileLegalBriefs`).

Polymorphism examples

- You can use the object's extra functionality by casting.

```
Employee* edna = new Lawyer("Edna", "Harvard", 5);
edna->getName();                                  // ok
edna->getNumberOfClients();                       // compiler error
((Lawyer*) edna)->getNumberOfClients();           // ok
```

- You should not cast a pointer into something that it is not.
  - It will compile, but the code will crash (or behave unpredictably) when you try to run it.

```
Employee* paul = new Programmer("Paul", 3);
paul->code();                                     // compiler error
((Programmer*) paul)->code();                     // ok
((Lawyer*) paul)->getNumberOfClients();           // crash!
```

Recap

- Inheritance
- Composition
- Announcements
- Polymorphism

- **Learning Goal:** understand how to create and use classes that build on each other's functionality.
- **Next time:** more Polymorphism; sorting