Topic: Inheritance
Outline

• Basic Concepts
• Subtypes and Polymorphism
• Inheritance in C++
• Type casting
• Virtual Methods
• Design Considerations
Inheritance Relationship

• Classes may “inherit” properties from others
• Ex. A dog is a mammal
• Inheritance defines a hierarchy of classes

Diagram:

```
mammal
  ∪
  \\
  dog  cat  whale
```
Inheritance Terminology

• Base, super- or parent class
  Refers to class inherited from

• Derived, sub-, or child class
  Refers to class which inherits
Subtypes

• We want inheritance to reflect a subtype relationship
• Examples:
  – A dog is a mammal
  – An integer is a real number
  – The numbers 0-9 are integers
• Subtypes share properties (operations) with supertype
Polymorphism and Subtypes

- A subtype can be used anywhere a supertype can be
  - Ex. An integer in the range 0-10 can be used anywhere an integer can be used
  - Ex. An integer can be used anywhere a real number can be used
- Defining a function on supertypes allows us to use it on the subtypes
- Limited form of polymorphism
Views of Inheritance

• Defining subtypes
  – Matches intuition
  – Good for polymorphism
  – Robust relationships over long-term

• Approach to code reuse
  – Sometimes not intuitive
  – Relationships can change
Inheritance in C++

class BaseClass {
   // declarations of class members
};

class DerivedClass : public BaseClass {
   // declarations of extra class members
};
An Inheritance Example

Hierarchy defines *roles of a person*
Base Class: Person

class Person {
    public:
        Person(const Name&, const Address&);
        Person(const Person&);
        void setName(const Name&);
        Name getName() const;
        void setAddress(const Address&);
        Address getAddress() const;
    private:
        Name _name;
        Address _address;
};
Derived Class: Student

class Student : public Person {
    public:
        Student(const Name& nm, const Address&, const string& mjr,
                 const StudentID&, const class_level&);
        void setMajor(const string&);
        string getMajor() const;
        StudentID getID() const;
        void setLevel(const class_level&);
        class_level getLevel() const;
    private:
        string _major;
        StudentID _id_number;
        class_level _level;
};
What Is Inherited?

• Student inherits from Person
  – Data fields – but not access privilege
  – Member functions

• Allows
  Student sue(nm, add, mjr, id, class);
  Sue.getName();  //base class method
  Sue.setAddress(newadd);  //base class method
Student Constructor

Student(const Name& nm, const Address& add, const string& mjr, const StudentID& id, const class_level& level) :
    Person(nm, add),
    _major(mjr),
    _id_number(id),
    _level(level)
{}
Student Copy Constructor

Student(const Student& s) :
    Person(s),
    _major(s._major),
    _id_number(s._id_number),
    _level(s._level)
{}
Student Default Constructor

```
Student() : Person(), _major(""""),
           _id_number(), _level(FRESHMAN)
{}  

OR

Student() :
           _major(""""), _id_number(), _level(FRESHMAN)
{}  
```
Derived Class: Employee

class Employee : public Person {
  public:
    Employee(const Person&, const string&);
    Employee(const Employee&);
    void setDept(const string&);
    string& getDept() const;
  private:
    string _dept;
};
Derived Class: Staff

class Staff : public Employee {

public:
    Staff(const Employee&, const float&);
    Staff(const Staff&);
    void setWage(const float&);
    float getWage() const;
    float grossIncome(short days) const;

private:
    float _hourly_wage;
};
Derived Class: Professor

class Professor : public Employee {
    public:
        Professor(const Employee&, const float&);
        Professor(const Professor&);
        void setSalary(const float&);
        float getSalary() const;
        float grossIncome(short days) const;
    private:
        float _salary;
};
Constructor Execution Order

- When Professor object created, constructors are executed in the order
  
  1. Person constructor
  
  2. Employee constructor
  
  3. Professor constructor
Access Rights

• Suppose wanted to write
  void Professor::giveRaise(const Policy& p)
• Suppose depends on the department of prof
• Is something like this OK?
  p.computeRaise(_dept, _salary)
• The identifier _dept cannot be used inside Professor class
Replacing Methods

- A method that is defined for both base and derived class
- Method in derived class hides base method

```cpp
class exA {
    public:
        void addOne() { a++; }
    private:
        int a;
};

class exB : public exA {
    public:
        void addOne() {
            b = (b+1) % 12;
        }
    private:
        short b;
};
```
Extending Methods

• Method with definitions for both base and derived classes, and, also
• Derived class version calls base class version

```cpp
class exA {
public:
    void addOne() { a++; }
private:
    int a;
};

class exB : public exA {
public:
    void addOne() { exA::addOne();
        b = (b+1) % 12; }
private:
    short b;
};
```
Invoking Replaced Methods

• For most inherited methods of parent class, a call uses the method from the parent class.
• If method is replaced (or extended) the replacement method is used
• Example
  – In the code
    ```cpp
exB b_obj;
b_obj.addOne();
```
  – The `exB::addOne` method is used
A Perverse Inheritance Example
Strange Professor Class

// use staff wage for salary
class Professor : public Staff {
    public:
        Professor(const Employee& e, const float& s) :
        Staff(e, s) {}
        void setSalary(const float& s) { setWage(s); }  
        float getSalary() const { return getWage(); }  
    };
Why “Perverse”? 

- Uses inheritance to avoid having to declare salary
- Still has inherited methods referring to wage, even though prof isn’t paid hourly
- Creates a relationship between staff and professor classes that is not natural
- Changes to staff imply changes to prof
Other Possible Strangeness

• Point as base class for Circle and Rectangle
  – A circle/rectangle is not a point
  – Both *have* points
• List as base class for Stack
  – A stack does not have all of the properties of a list
  – Would have to “hide” some list methods
• Aggregation better in both cases
Inheritance and Modification

- Inheritance makes modifications harder
- Changes to derived class – have to understand relationship to base class(es)
- Changes to base class – have to understand relationship to all derived classes
Inheritance and Testing

• Inheritance poses problems when testing
• Regression testing – tests done after changes
• Changes to base classes require regression testing for base class and all derived classes
Inheritance for Polymorphism

• Casting type to base class

• Virtual Methods

• Pure Virtual Methods
Type Casting

• Forced type conversion
• Used here to make an object of a derived class look like it belongs to base class
• Syntax:
  
  \texttt{static\_cast\<BaseClass\>(variablename)}

• C-style syntax is deprecated
Storage for Classes

Person

Employee

Staff

_name
_address
_dept

_name
_address

_name
_address
_dept
_wage
Assigning Derived to Base

Employee ellen(…);
Person p = ellen;

Only fields of Person copied

p
_name
_address

ellen
_name
_address

Rest is lost

_dept
Casting Type to Base Class

Use pointers (or references) to objects

\[
\text{Employee}\* \ e = \text{new Employee}(\text{e\_person}, \text{e\_dept}); \\
\text{Person}\* \ p = \text{static\_cast\<Person\*>}(e);
\]

Pointer \( p \) is now an alias for pointer \( e \)

However,

\[
\text{e\->getDept();} \quad // \text{OK, } e \text{ is an Employee}\* \\
\text{p\->getDept();} \quad // \text{Error, } p \text{ is a Person}\*
\]
Polymorphism via Casting

bool Search(const Name& f_nme, Person** a, int dim) {
    int mid = (dim - 1)/2;
    int left = 0; int right = dim-1;
    while ((a[mid]->getName() != f_nme) && (left < right))
        if (a[mid]->getName() < f_nme) {
            right = mid;  mid = (mid - left)/2;
        }
        else {
            left = mid; mid = (right - mid)/2;
        }
    return (a[mid]->getName()) != f_nme;
}
Polymorphism via Casting (2)

Person* a[4];
Professor* p = new Professor(e1, sal1);
Student* s = new Student(snm, sadd, mjr,...);
Staff* st1 = new Staff(e2, wage2);
Staff* st2 = new Staff(e3, wage3);
a[0] = static_cast<Person*>(p);
a[1] = static_cast<Person*>(s);
a[2] = static_cast<Person*>(st1);
a[3] = static_cast<Person*>(st2);
search(name, a, 4);
Notes on Poly via Casting

• Array can hold any derived class of Person
• Search uses getName() method of Person
• Polymorphism limited to Person hierarchy
• Cast makes compiler view derived object as a Person object
  – If Staff had replaced getName() method, the Person method would still be used
  – Casting does not allow us to use methods particular to derived classes
Static vs Dynamic Binding

- Method call is ordinarily statically bound to a method
  - Compiler uses pointer type to determine which
  - Same method used every time call is made
- Dynamic binding (or dispatch) allows method to be determined when call is made
- Dynamic binding is solution to problem of using methods from derived classes
Virtual methods

• Methods can be declared as virtual
• Sets up dynamic binding mechanism
  – Use pointer for base class to point to object of derived class (like before)
  – Method call for virtual method is dynamically bound to method of derived class
• Methods declared as virtual in base class are virtual for all derived classes
Example: Rectangles

• Classes to represent rectangles in a graphics program
  – Plain rectangles – display as lines
  – Filled rectangles – display with fill color

• Make (unfilled) Rectangle the base class
Rectangle without Virtual

class Rectangle {
  public:
    Rectangle(Location, Location);
    void draw(Canvas&) const;
    ...
};
Filled Rectangle without Virtual

class FilledRect : public Rectangle {
    public:
        FilledRect (Location, Location, Color);
        void draw(Canvas&) const;
        ...
};
Using Rectangle

Rectangle* a[2];
Rectangle* plain = new Rectangle(loc1, loc2);
FilledRect* red = new FilledRect (loc3,loc4,Color::red);
a[0] = plain;
a[1] = static_cast<Rectangle*>(red);
a[0]->draw(windowcanv);
a[1]->draw(windowcanv);
Display of Rectangles
Problem

• *Problem*: Method Rectangle::draw doesn’t know about fill colors
• *Solution*: make draw a virtual method
Rectangle with Virtual

class Rectangle {
    public:
    Rectangle(Location, Location);
    virtual void draw(Canvas&) const;
    ...
};

Note: Additional virtual modifier not needed in function definition.
Filled Rectangle with Virtual

class FilledRect : public Rectangle {
    public:
        FilledRect (Location, Location, Color);
        virtual void draw(Canvas&) const;
        ...  
    }

Note: method is virtual in all derived classes; modifier not needed
Using Rectangle

Rectangle* a[2];
Rectangle* plain = new Rectangle(loc1, loc2);
FilledRect* red = new FilledRect (loc3,loc4,Color::red);
a[0] = plain;
a[1] = static_cast<Rectangle*>(red);
a[0]->draw(windowcanv);
a[1]->draw(windowcanv);
Display of Rectangles (2)
Virtual Destructors

• Destructors of base classes should be declared virtual
• Ensures that they will be called when derived objects are destructed
Implicit Dynamic Binding

class Rectangle {
    public:
        virtual void draw(Canvas& ) const;
        void resize(int width, int height, Canvas& c);
        ...
};

void Rectangle::resize(int width, int height, Canvas& c) {
    // code to change lower right location
    draw(c); // really this->draw(c);
}
Implicit Dynamic Binding (cont)

For derived classes only need to redefine draw

class FilledRect : public Rectangle {
   public:
      virtual void draw(Canvas&) const;
      ...
};

If applied to FilledRect object, resize will use FilledRect::draw
Recognizing Dynamic Binding

- Nonvirtual methods always statically bound
- Virtual methods
  - Static when
    - Applied to object: `rect.draw(canv);`
    - Class explicitly named: `Rectangle::draw(canv)`
  - Dynamic when applied to
    - Pointer (see rectangle example)
    - Implicit object ("this" pointer)
Virtual Methods and Changes

• Virtual methods help with extensions
• Ex. Adding a “labeled” rectangle that contains a text label
• Only need to define new derived class
• No existing code needs to be changed for new class to be usable
Type Casting (again)

• Widening - convert from derived to base class
  – Always safe
  – Use static_cast<->
• Narrowing
  – Converting type from base to derived class
  – Requires run-time type check
  – Use dynamic_cast<->
  – Class must have at least one virtual method
Dynamic Casting

• Dynamic cast incorporates a type check
• Can write code like

```c++
Rectangle* s;
...
FilledRect* r = dynamic_cast<FilledRect*>(s);
if (r != NULL)
    // r equals s, points to FilledRect object
else
    // s points to object that is not a FilledRect
```
Pure Virtual Methods

• A *pure virtual method* has a null definition
  
  virtual void draw(Canvas&) = 0;

• *Abstract class* - class in which at least one method is pure virtual
  – Cannot be instantiated (no objects)
  – Can have fields and other methods

• *Pure abstract class* - defines an interface
Why Abstract Classes?

• Abstract class corresponds to general concept that is too general to actually have its own instances (must have derived class)

• Examples:
  – Geometric shape
  – Number
  – Mammal
Example: Geometric Shapes

• General class for geometric shapes

```cpp
class Shape {
    public:
        virtual void draw (Canvas&) const = 0;
        virtual void print (ostream&) const = 0;
        virtual void scale (Point center, double s);
        virtual void move (int x, int y);
    };
```

• Better style to not include data
Other C++ Inheritance

- Generally problematic
- Protected data
  - Derived class inherits
  - Decreases separation between implementations of base and derived classes
- Private and protected inheritance
  - Determines access to inherited public members
  - Confusing – done better with aggregation
- Multiple inheritance
Design Observations

• A base class should contain common operations and fields
  – Means you should not have to “hide” inherited methods
  – If you do, consider alternative designs
    • Using aggregation
    • A different hierarchy (Staff & Professor)
Design Observations (2)

• Use fields to keep track of state, methods to keep track of behavior
  – Ex. Base: Vehicle; Derived: Car, Truck
    Cars have speed limit of 65, trucks 55
  – Awkward if Car and Truck define own max_speed methods
  – Solution:
    • Add _max_speed, and max_speed accessor to Vehicle class
    • Car and Truck classes set value in constructor
Design Observations (3)

- Methods of derived-class must preserve assumptions of base-class
  - Should not change state of inherited data to violate assumptions of base class
  - Not a worry if using public inheritance
  - Could be a problem with protected inheritance
Design Observations (4)

- Objects of derived class should be preserved by inherited methods
  - Ex. Deriving Stack from List
    - List methods may insert anywhere in stack
    - Violates property of being a stack
  - Situation where would have to hide methods
Design Observations (5)

• Polymorphism should be used where you would use type information
  – Ex. Code of the form
    
    ```
    if (x is of type 1) do_this();
    else if (x is of type 2) do_that();
    ```
  – Can be replaced by virtual methods
  – Virtual methods easier to maintain
Design Observations (6)

• Move common behavior to the base class
  – Some methods may be slightly different for different derived classes
  – Try moving method to base class, but define it using simpler virtual methods that derived classes can define easily
  – Prevents having to define more complex methods for all derived classes
Design Observations (7)

• Don’t use protected data
  – Concern is that representation of class will be changed over time
  – If protected data changes, all derived classes change
  – If methods of base class maintain some property of implementation, better to not provide access to derived classes (ex. Sorted list)
Composition Strategies (P. Coad)

• Composition  = aggregation and association
• *Composition Strategy*: Use composition to extend responsibilities by delegating work to other classes.
• Prefer composition over inheritance
Inheritance Strategy (Coad)

- Inheritance is used to extend attributes and methods
- Use should be restricted, because the relationship between base and derived classes leads is a weak form of encapsulation
When to Use Inheritance

Inheritance relationship must satisfy:

1. Represents “is a special kind of”, and not “is a role of”
2. An object of one class in hierarchy never needs to transmute to another class
3. Derived class extends rather than overriding or nullifying base class
4. Does not derive for the purpose of copying useful capabilities
5. If classes from problem domain, represents special kinds of roles, transactions or devices
Coad’s View of Person Example

Revisit Person/Student/Employee

1. Represents “is a special kind of”
   No, a student is a role of a person

2. No transmutation
   No, a student could become an employee

3. Only extends --- Yes!

4. Not deriving “useful” methods --- Yes!

5. PD classes represent kinds --- No (roles)
New Person Hierarchy

Person

Name
Address

PersonRole

1

n

Student

Employee

Professor

Staff
Evaluate New Person Example

1. Represents “is a special kind of” --- Yes
2. No transmutation
   *OK, person may have many roles*
3. Only extends --- Yes
4. Not deriving “useful” methods --- Yes
5. PD classes represent kinds
   *Yes, has special kinds of roles*
Inheritance Overview

• Useful for coding polymorphism
  – Virtual methods
  – Containers of heterogeneous objects
• Design so that derived class objects are a “special kind of” base class object
• Need to use multiple inheritance is rare
• Consider association and aggregation first