CS2704

Topic:
Inheritance

Outline

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

Inheritance Relationship

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

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++

```cpp
class BaseClass {
  // declarations of class members
};

class DerivedClass : public BaseClass {
  // declarations of extra class members
};
```

An Inheritance Example

Hierarchy defines roles of a person

Derived Class: Student

```cpp
class Student : public Person {
  public:
    Student(const Name& nm, const Address& addr, const strings& mjr,: 
      const StudentID, const class_level&);
    void setName(const Name&);
    string getMajor() const;
    StudentID getID() const;
    void setID(const StudentID&);
    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.setName(); // base class method
  Sue.setAddress(new add); // base class method
Student Constructor

Student(const Name &nm, const Address &addr, 
    const string &mj, const StudentID &id, 
    const class_level &level) :
    Person(nm, addr), 
    _major(mj), 
    _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 &p, const string &s);
    Employee(const Employee &e);
    void setDept(const string &s);
    string &getDept() const;
    private:
        string _dept;
};

Derived Class: Staff

class Staff : public Employee {
public:
    Staff(const Employee &e, const float &f);
    Staff(const Staff &s);
    void setWage(const float &f);
    float getWage() const;
    float grossIncome(short days) const;
    private:
        float _hourly_wage;
};

Derived Class: Professor

class Professor : public Employee {
public:
    Professor(const Employee &e, const float &f);
    Professor(const Professor &p);
    void setSalary(const float &s);
    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

Extending Methods

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

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
    exB b_obj;
    b_obj.addOne();
  - The exB::addOne method is used

A Perverse Inheritance Example
Strange Professor Class

```cpp
// 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: `static_cast<BaseClass> variableName`
- C-style syntax is deprecated

Storage for Classes

<table>
<thead>
<tr>
<th>Person</th>
<th>Employee</th>
<th>Staff</th>
</tr>
</thead>
<tbody>
<tr>
<td>_name</td>
<td>_name</td>
<td>_name</td>
</tr>
<tr>
<td>_address</td>
<td>_address</td>
<td>_address</td>
</tr>
<tr>
<td>_dept</td>
<td>_dept</td>
<td>_wage</td>
</tr>
</tbody>
</table>

Assigning Derived to Base

Employee ellen(...);
Person p = ellen;

Only fields of Person copied

p ellen

_rest is lost

_cname

_Caddress

Casting Type to Base Class

Use pointers (or references) to objects

Employee* e = new Employee(e_person, e_dept);
Person* p = static_cast<Person*>(e);

Pointer p is now an alias for pointer e

However,
e->getDept(); // OK, e is an Employee*
p->getDept(); // Error, p 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)) {
        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, sall1);
Student* s = new Student(snm, addm, ...);
Staff* s1 = new Staff(e2, wage2);
Staff* s2 = new Staff(e3, wage3);
a[0] = static_cast<Person*>( p);
a[1] = static_cast<Person*>( s);
a[2] = static_cast<Person*>( s1);
a[3] = static_cast<Person*>( s2);
search(f_nme, 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

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

Filled Rectangle without Virtual

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

```cpp
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(windowCanvas);
a[1]->draw(windowCanvas);
```

Display of Rectangles

Problem

- **Problem:** Method `Rectangle::draw` doesn't know about fill colors
- **Solution:** make `draw` a virtual method

Rectangle with Virtual

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

Note: Additional virtual modifier not needed in function definition.

Filled Rectangle with Virtual

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

Using Rectangle

```cpp
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(windowCanvas);
a[1]->draw(windowCanvas);
```

Note: method is virtual in all derived classes; modifier not needed.
Virtual Destructors

- Destructors of base classes should be declared virtual
- Ensures that they will be called when derived objects are destructed

Implicit Dynamic Binding

```cpp
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`

```cpp
class FilledRect : public Rectangle {
public:
    virtual void draw(Canvas& c) const;
    
    void FilledRect::draw(Canvas& c) {
        // draw with additional steps
    }
};
```

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
  ```cpp
  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
  ```cpp
  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

Protected Data

- If want data to be accessible to derived class, can declare as protected.
  ```cpp
  class BaseClass {
  public:
    // inherited, visible from outside
    protected:
    // inherited, private from outside
    private:
    // private in base class and outside
  }
  ```
Private & Protected Inheritance

- Have used public inheritance
  class DerivedClass : public BaseClass
- Can replace public with private or protected
  - Private - all public members made private in derived class
  - Protected - all public members made protected

Templates and Inheritance

- Several possible ways of using inheritance with Templates:
  - Template class inheriting from template
  - Class inheriting from template
  - Template inheriting from class

Inheriting From Template

```cpp
template <class T> class Queue {
    public:
    Queue();
    void Insert(T item);
    T Remove();
    ~Queue();
    private:
    T buffer[100];
    int head, tail, count;
};
```

Inheriting from Template (2)

```cpp
template <class Item>
    classInspectableQueue : public Queue<Item> {
    public:
   InspectableQueue();
    Item Inspect(); // return first element
    ~InspectableQueue();
    }
```

Templates vs Inheritance

- Templates - static form of polymorphism
  - Containers of objects (homogeneous)
  - Best when want to use in different ways in several projects
- Inheritance - dynamic form
  - Objects with similar behavior
  - Best when want polymorphic functions or heterogeneous containers

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_things();
    else if (x is of type 2) do_something();
    ```
  - 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 to 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

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
Multiple Inheritance

class TA : public Student, public Employee {
  ...
};

Objects and Multiple Inheritance

Student Data

Employee Data

Teaching Assistant Data

A Problem with Data

- Don’t forget rest of inheritance hierarchy

Person Data
  Major, level, ID
  Department

- A TA could have two names!

C++ Solution

- Change declarations of base classes
  class Student : virtual public Person {...};
  class Employee : virtual public Person {...};

- If inherit both
  class TA : public Student, public Employee {...};

- TA object contains pointers to Person object(s)

Virtual Inheritance

Hypothetical data layout - compiler may do something else

Person Data
  Major, level, ID
  Department

Student Data

Employee Data

Teaching Assistant Data

Casting & Virtual Inheritance

- Ordinarily casting pointers “does nothing”
- The cast
  static_cast<Person*>(studentp)
  follows pointer

- The cast
  dynamic_cast<Student*>(st_person)
  works if virtual base class has virtual function (usually destructor)
Methods and Multiple Inheritance

- Both Parents have a move method

```
TextBox
  move(Location)
```

```
Canvas
  move(Location)
```

```
DrawOnText
```

A Problem with Methods

- Can’t tell which base class to get method from

```
DrawOnText y;
y.move(lowerleft);
```

- Could be `TextBox::move(-)` or `Canvas::move(-)`
- Redefine methods with name clash

Method Dominance

For virtual inheritance

```
Person
  id() Returns SSN

Employee
  id() Returns Student ID

Student
  id() is Student::id()

Teaching Assistant
```

Ambiguous in non-virtual case

On Multiple Inheritance

- Much disagreement on whether multiple inheritance necessary
- Difficult to come up with examples that couldn’t be done equally well or better some other way
- Multiple inheritance increases complexity

When Multiple Inheritance?

- Disjoint base classes
  - No common base class
  - No method name clashes
- Want to enforce some protocol
  - Ex. MFC persistence functionality
- Conclusion: don’t go out of your way to use it, but recognize could be useful

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