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
  ```
  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
    //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
  – Public - all public members made private in derived class
  – Protected - all public members made protected
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

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
    
    ```java
    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 change 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
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

<table>
<thead>
<tr>
<th>Person Data</th>
<th>Student Data</th>
<th>Teaching Assistant Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major, level, ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Person Data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Department</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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

- pointer
- Major, level, ID
- pointer
- Department

\{ Student Data \}

\{ Employee Data \}

\{ Teaching Assistant Data \}

Casting & Virtual Inheritance

- Ordinarily casting pointers “does nothing”
- The cast
  \texttt{static\_cast\!<\!Person\*>\!} (studentp)
  follows pointer
- The cast
  \texttt{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

A Problem with Methods

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

```cpp
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

- Teaching Assistant
  - id() is Student::id()

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