Anagram: Using Normalizers

- How can we normalize an Anaword object differently?
  - Call normalize explicitly on all Anaword objects
  - Have Anaword objects normalize themselves
  - Advantages? Disadvantages?

- If Anaword objects normalize themselves, how can we experiment with different normalization techniques?
  - What about `cp anaword.cpp oldanaword.cpp`?
  - What about deciding at runtime on normalization?

- We need inheritance!
Normalizer hierarchy

- **Anaword objects normalize themselves**
  - Where does the normalizer come from?
    - Passed in at construction time
    - Obtained from normalizer factory
    - Other approaches?

  ➢ How is Normalizer used?

- **Normalizer is conceptually an interface**
  - Different implementations of the interface have different behavior (guts) but same skin (sort of)
Benefits of inheritance, interfaces

- Suppose you learn about a new class WebStream that conforms to the input stream interface (cin, ifstream, ...)
  - Read words, lines, chars from a web page?

```cpp
void readWords(istream& input) {
    string s, line;
    char ch;
    getline(input, line);
    input >> s;
    input.get(ch);
}
```

- We can pass cin, ifstream, WebStream, etc. to readWords
  - Why is this an advantage?
  - Inheritance and late/dynamic binding
### Why inheritance?

Adding new shapes easily without changing much code:
- `Shape * s1 = new Circle();`
- `Shape * s2 = new Square();`

**Abstract base class:**
- Interface or abstraction
- Pure virtual function

**Concrete subclass:**
- Implementation
- Provide a version of all pure functions

"is-a" view of inheritance:
- Substitutable for, usable in all cases as-a

User’s eye view: think and program with *abstractions*, realize different, but conforming *implementations*,

don’t commit to something concrete until as late as possible
Example of inheritance

- What is behavior of a shape?

```cpp
double doShape(Shape * s) {
    cout << s->area() << endl;
    cout << s->perimeter() << endl;
    s->expand(2.0);
    cout << s->area() << endl;
    cout << s->perimeter() << endl;
}

Shape * s1 = new Circle(2);
Shape * s2 = new Square(4);
Shape * s3 = new Rectangle(2,5);
doShape(s1); doShape(s2); doShape(s3);
```
Inheritance (language independent)

- **First view: exploit common interfaces in programming**
  - Streams in C++, iterators in Tapestry classes
    - Iterators in STL/C++ share interface by convention/templates
  - Implementation varies while interface stays the same

- **Second view: share code, factor code into parent class**
  - Code in parent class shared by subclasses
  - Subclasses can *override* inherited method
    - Can subclasses override and call?

- **Polymorphism/late(runtime) binding (compare: static)**
  - Actual function called determined when program runs, not when program is compiled
Inheritance Heuristics

- A base/parent class is an interface
  - Subclasses implement the interface
    - Behavior changes in subclasses, but there’s commonality
  - The base/parent class can supply some default behavior
    - Derived classes can use, override, both

- Push common behavior as high up as possible in an inheritance hierarchy

- If the subclasses aren’t used polymorphically (e.g., through a pointer to the base class) then the inheritance hierarchy is probably flawed
Normalizer details (see Anaword)

- What’s static? Why private? Static initialization?

```cpp
class Anaword
{
    public:
        // not shown
    private:
        void normalize();
        static Normalizer * ourNormalizer;
};

void Anaword::normalize()
// postcondition: mySortedWord is sorted version of myWord
{
    if (ourNormalizer == 0) {
        ourNormalizer = NormFactory::getNormalizer();
    }
    myNormalizedWord = ourNormalizer->normalize(myWord);
}
```
Where are the objects in Yahtzee?

- Similarities/differences in different scorecard entries?
  - Scoring?
  - Bonus?

- How do we play a game?
  - Roll dice (one, two, ...)
  - Make decision
  - Repeat
  - Decide where to score?

- Inheritance: leverage differences in common behavior
Objects, Classes, Tests

- **What classes have something to do with Dice?**
  - What’s the behavior of a Dice (sp) and a DiceGroup?
  - What about testing scoring behavior with randomness?
  - See DiceGroup, FixedDice, CupOfDice classes

- **What about ScoreCard and ScoreCardEntry?**
  - What behavior does a score card have?
  - Behavior of score card entry (small straight, chance, ...)

- **What about Game behavior?**
  - When do we stop playing?
What’s a “traditional” ScoreEntry?

class ScoreEntry
{
    public:
        enum Kind{
            ones, twos, threes, fours, fives, sixes,
            kind3, kind4, fullhouse, smallstraight,
            largestraight, yahtzee, chance
        };
        ScoreEntry(Kind kind);
        void score(const DiceGroup& dg);
        // other methods/member functions here
    private:
        int myScore;
        ScoreEntry::Kind myKind; // what am I
ScoreEntry code (non-OO version)

```cpp
if (myKind == kind3 || myKind == kind4) {
  // ..
}
else if (myKind == fullhouse) {
  // ..
}
else if (myKind == smallstraight) {
  // ..
}
```

- This kind of coding doesn’t scale (and isn’t OO)
  - Adding a new kind requires recompiling .cpp
  - Add new enum requires changing .h too
  - Duplication in score() and getDescription()
Inheritance and Yahtzee program

- **ScoreEntry** is a “kitchen-sink” class
  - Changing the .h requires recompiling all files that include it, either directly or indirectly
  - Consequences of large-scale recompiling? What about building large programs (word, XP, etc.)

- Changes made in several places in scoreentry.cpp as well
  - Code in different places, related, must be synchronized
  - Order of entries in enum makes a difference (hard to find bugs as a result)

- Inheritance helps keep related code together, avoids recompilation, facilitates extension, keeps each class simple
  - Consequence: many classes rather than one
Yahtzee with inheritance

- Base class behavior: some default, some left to subclasses
  - If subclass can change behavior use `virtual` keyword
  - If subclass must implement, use `= 0` aka abstract
    - This is a `pure virtual function`

```cpp
class ScoreEntry
{
public:
    ScoreEntry();
    virtual ~ScoreEntry();
    virtual int calcScore(const DiceGroup& dg) const = 0;
    virtual string getDescription() const = 0;
    virtual int getScore() const;
    virtual void scoreIt(const DiceGroup& dg);
};
```
Tradeoffs in per-class score entry

- **More classes: one per entry (above line entries are same)**
  - Proliferation of classes, harder to understand/grok?
  - Test classes in isolation, facilitates code/test/deploy

- **In creating a new score-card entry, do we modify existing header files? Existing .cpp files? Benefits?**
  - What must be recompiled when adding small straight to the ScoreCard class?
  - Somehow the ScoreCard class must have all entries.
Guidelines for using inheritance

- **Create a base/super/parent class that specifies the behavior that will be implemented in subclasses**
  - Most/All functions in base class may be virtual
    - Often pure virtual (= 0 syntax), subclasses *must* implement
  - Subclasses do not need to specify virtual, but good idea
    - May subclass further, show programmer what’s going on
  - Subclasses specify inheritance using: `public Base`
    - C++ has other kinds of inheritance, stay away from these
  - Must have virtual destructor in base class

- **Inheritance models “is-a” relationship, a subclass is-a parent-class, can be used-as-a, is substitutable-for**
  - Standard examples include animals and shapes
Inheritance guidelines/examples

- **Virtual function binding is determined at run-time**
  - Non-virtual function binding (which one is called) determined at compile time
  - Need compile-time, or *late*, or polymorphic binding
  - Small overhead for using virtual functions in terms of speed, design flexibility replaces need for speed
    - Contrast Java, all functions “virtual” by default
- **In a base class, make all functions virtual**
  - Allow design flexibility, if you need speed you’re wrong, or do it later
- **In C++, inheritance works only through pointer or reference**
  - If a copy is made, all bets are off, need the “real” object
class Student
{
    public:
        Student(const string & name);
        virtual ~Student();

        virtual void eat();
        virtual void work();
        virtual void sleep();
        virtual void live();

        bool isAlive() const;

    // more here
Implementation of behavior, .cpp file

```cpp
void Student::sleep()
{
    myEnergy += 10;
    cout << "Zzzzzzzzzzzzzz, resting sleep" << endl;
}

void Student::live()
{
    eat();
    work();
    sleep();
}
```
See students.cpp, school.cpp

- Base class student doesn’t have all functions virtual
  - What if subclass has different `name()` function?
    - `name()` bound at compile time, no change observed

- How do subclass objects call parent class code, see `DukeStudent` class in school.cpp
  - `class::function` syntax, must know name of parent class

- Why is base class data protected rather than private?
  - Must be accessed directly in subclasses, why?
  - Not ideal, try to avoid state in base/parent class: trouble
    - What if derived class doesn’t need data?
Inheritance Heuristics in C++

- Pure virtual (aka abstract) function makes a class abstract
  - Cannot be instantiated, but can be constructed (why?)
    - What do subclasses do?
  - Default in C++ is non-virtual or *monomorphic*
    - Unreasonable emphasis on efficiency, sacrifices generality
    - If you think subclassing will occur, all methods are virtual
  - Must have virtual destructor, the base class destructor (and constructor) will be called

- We use public inheritance, models *is-a* relationship
  - Private inheritance means is-implemented-in-terms-of
    - Implementation technique, not design technique
    - Not ubiquitous in other languages
Difference in behavior?

- What’s a field and what’s a method?
  - # tires on car?
  - # doors on car?
  - How student lives?

- Where does name of school belong? What about energy increment?

- What’s problem with hierarchy here?
  - NCState student?
Problems with inheritance

- Consider the student example and burrito eating
  - CosmicStudent is a subclass of DukeStudent
    - What behavior changes in the new subclass?
  - What about a UNCStudent eating cosmic cantina food?
    - Can we have CosmicDukeStudent and CosmicUNCStudent?
    - Problems with this approach?

- Alternative to inheritance: use delegation (aka layering, composition)
  - Just like myEnergy is a state variable with different values, make myEater a state variable with different values
  - Delegate behavior to another object rather than implementing it directly
Delegation with school/student

- If there's a class Eater, then what instance variable/field will a Student store to which eating behavior delegated?

```cpp
void Student::eat()
{
    myEater->doEat();
}
```

- How is the eater instance variable initialized?
- Could we adopt this approach for studying too?
- When is this approach better/worse?