Anagrams/Jumbles

- How do humans solve puzzles like that at www.jumble.com
  - Is it important to get computers to solve similar puzzles? Reasons?
  - Should computers mimic humans in puzzle-solving, game playing, etc.? Lessons from chess?
- nelir, nelri, neilr, neirl, nerli, neril, nleir, nleri, niler, nilre, nilre, nier, nierl, niler, nilre, nilr, ... lenir, lenri, leinr, leirn, leirm, leiri, lerin, linr, liner
- What’s the problem here?

Brute force? permana.cpp

```cpp
void printAnagrams(string word, const tvector<string>& v)
  // post: print all values in v that are equal to word
  { 
    cout << "anagrams for " << word << " : ";
    sort(word.begin(), word.end()); // alphabetically first
    do{
      for(int k=0; k < v.size(); k++)
        if (word == v[k]) {
          cout << v[k] << endl;
          break;
        }
    } while (next_permutation(word.begin(), word.end()));
  }
```

Quantifying brute force for anagrams

- How big anagram/jumbles can we find if we can generate and test three letter words in 0.42, four-letter in 2.18, five letter in 13.27 using a 45,000 word dictionary? How long will six letter word take?
- We’re willing to do some pre-processing to make the time to find anagrams quicker
  - Often find that some initialization/up-front time or cost saves in the long run
  - What properties do words share that are anagrams?

Toward a faster anagram finder

- Words that are anagrams have the same letters; use a letter fingerprint or signature/histogram to help find anagrams
  - Count how many times each letter occurs:
    - "teacher" 1 0 1 0 2 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
    - "cheater" 1 0 1 0 2 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
- Store words, but use fingerprint for comparison when searching for an anagram
  - How to compare fingerprints using operator ==
  - How to compare fingerprints using operator <
- How do we make client programmers unaware of fingerprints? Should we do this?
Another anagram method

- Instead of fingerprint/histogram idea, use sorted form of word
  - “gable” and “bagel” both yield “abegl”
  - Anagrams share same sorted form
- Similarities/differences to histogram/fingerprint idea?
  - Both use canonical or normal/normalized form
  - Normalized form used for comparison, not for printing
  - When should this normal form be created?
- When is one method preferred over the other?
  - Big words, little words? Different alphabets? DNA vs English?

OO and C++

- We’ll use an adapter or wrapper class called Anaword instead of a string
  - Clients can treat Anaword objects like strings, but the objects are better suited for finding anagrams than strings
  - The Anaword for “bear” prints as “bear” but compares to other Anaword objects as 11001000000000000100000000
- C++ allows us to overload operators to help, not necessary but good cosmetically
  - Relational operators == and <
    - What about other operators: >, <=, >=, and !=
  - Stream operator <<

See simpleanagram.cpp

- The code does things simply, but isn’t very OO. Why is simple sometimes better? Why is it worse?

```cpp
void printAnagrams(const Ana& word, const tvector<Ana>& v)
{
  cout << "anagrams for " << word << " : ";
  for(int k=0; k < v.size(); k++){
    if (word == v[k]){ cout << v[k] << " ";
    }
  }
  cout << endl;
}
```

Overloaded operators

```cpp
struct Ana
{
  Ana(const string& s) : myWord(s), mySorted(s) {
    sort(mySorted.begin(), mySorted.end());
    string myWord, mySorted;
  }
  ostream& operator << (ostream& out, const Ana& a) {
    out << a.myWord;
    return out;
  }
  bool operator == (const Ana& lhs, const Ana& rhs) {
    return lhs.mySorted == rhs.mySorted;
  }
};
```
Overloaded operators implemented

- In C++ we can define what operator == and operator < mean for an object (and many other operators as well)
  - This is syntactically convenient when writing code
  - C++ details can be cumbersome (see Tapestry Howto E)
- In anaword.h we find operators <, ==, <, >
  - What about > and >=; what about !=; others?
  - How do we access private data for printing? Comparing?
- Overloaded operators are not necessary, syntactic sugar.

How to implement overloaded ops

- Typically operators need access to internal state of an object
  - Relational operators for Date, string, BigInt?
  - Where is “internal state”?
- For technical reasons sometimes operators should not be member functions:
  - BigInt b = enterBigValue();
  - if (b < 2) …
  - if (2 > b) …
  - We’d like to use both if statements, only the first can be implemented using BigInt::operator < (why?)
- Use helper member functions: equals, less, toString
  - Implement overloaded operators using helpers

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. Speed? Don’t do it
  - In C++, inheritance works only through pointer or reference
    - If a copy is made, all bets are off, need the “real” object
Student behavior/interface? .h file

class Student
{
public:
    Student(const string & name);
    virtual ~Student();
    virtual void eat();
    virtual void work();
    virtual void sleep();
    virtual void live();
    bool isAlive() const;
};

Implementation of behavior, .cpp file

void Student::sleep()
{
    myEnergy += 10;
    cout << "Zzzzzzzzzzzzz, 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?

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?

Anaword objects with options

- Can we use different canonical forms in different contexts?
  - Could have Anaword, FingerPrintAnaword, SortAnaword
  - What possible issues arise? What behavior is different in subclasses?
    - If there's no difference in behavior, don't have subclasses

- Alternative, make canonical/normalize method a class
  - Turn a function/idea into a class, then let the class vary to encapsulate different methods
  - Normalization done at construction time or later
  - Where is normalizer object created? When?

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)

**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);
}
```