Designing and Implementing a class

- Consider a simple class to implement sets (no duplicates) or multisets (duplicates allowed)
  - Initially we’ll store strings, but we’d like to store anything, how is this done in C++?
  - How do we design: behavior or implementation first?
  - How do we test/implement, one method at a time or all at once?

- Tapestry book shows string set implemented with singly linked list and header node
  - Templated set class implemented after string set done
  - Detailed discussion of implementation provided

- We’ll look at a doubly-linked list implementation of a MultiSet
Behavior of MultiSet: methods?

- **What accessor functions do we need (these are const)?**
  - Size of set? Printing set? Element in set?
  - What does Print generalize to?
    - For Iterator class, see book, requires *friend* class

- **What mutator functions do we need (non const)?**
  - Insert an element into the set
  - Make a set empty
  - Erase an element (all occurrences or one?)

- **What constructor(s) and other, similar standard methods are needed**
  - Copy constructor, assignment operator, destructor
First draft of multiset behavior

class MultiSet
{

public:
MultiSet();                          // construct a wordlist
virtual ~MultiSet();                // destruct
// mutators
virtual void insert(const string & word);
virtual void clear();
// accessor
virtual void apply(MSApplicant & app) const;
virtual int occurrences(const string & key) const;

private:
};

● What’s missing? Why the word virtual? What is apply( . . )
From Behavior to Implementation

- **We’ll use a linked list**
  - New nodes added to front, but this will change in other versions: add to back, move to front, ...
  - Use a header node (not in set), last node (in set)
  - To print, count, etc. we’ll use an *internal iterator*
    - Pass to the set the operation you want to perform on all set elements
    - What’s good/bad about this compared to iterators?

- **Use doubly-linked list, where is Node declaration found?**
  - Private section means clients cannot access, ok?
  - We’ll need to use protected for subclasses, inheritance
What’s easy, hard, first?

- **Searching the list is straightforward, visit all nodes**
  - We need to search for `insert` and for `occurrences`, how can we factor out common code?
  - If we implemented remove/erase, we’d need to search too.

- **Once we implement insert, how do we test?**
  - We need to print, we should implement `print()` even if not general, we’ll toss it later, why is this ok?

- **What’s after insert? We’ll look to the accessor functions**
  - `occurrences` is simple, why?
  - What is `apply(..)` about?
Iterators and alternatives

```cpp
MultiSet ms;
ms.insert("bad");
ms.insert("good");
ms.insert("ugly");
ms.insert("good");

MultiSetIterator it(ms);
for(it.Init(); it.HasMore(); it.Next())
{
    cout << it.Current() << endl;
}
```

● What's printed? How might this work?
  ➤ What does iterator class have access to?
  ➤ What happens when `MultiSet` passed to iterator? Stored?
Iterators and alternatives, continued

- Iterators require class tightly coupled to container class
  - Friend class often required
  - Const can be a problem, but surmountable

- Alternative, pass function into `MultiSet`, function is applied to every element of the set
  - How can we pass a function?
  - What’s potential difference compared to `Iterator` class?

- To pass a function, we’ll put the function in a class and pass an object
  - Must adhere to naming conventions since written in client code, called in `MultiSet` code
Using a common interface

- We’ll use a function named `apply(…)`
  - It will be called for each string, count in a `MultiSet`
  - Encapsulated in a class, other functions can exist in the class, e.g., to access data [can use struct too]
  - What do functions below do, where are they written?

```cpp
void apply(const string& s, int count) const
{
    cout << count << "\t" << s << endl;
}
// alternative
void apply(const string& s, int count) const
{
    myTotal += count;
}
```
Interfaces and Inheritance

- Programming to a common interface is a good idea
  - I have an iterator named `FooIterator`, how is it used?
    - Convention enforces function names, not the compiler
  - I have an `ostream` object named `out`, how is it used?
    - Inheritance enforces function names, compiler checks

- Design a good interface and write client programs to use it
  - Change the implementation without changing client code
  - Example: read a stream, what’s the source of the stream?
    - file, cin, string, web, ...

- The syntax of inheritance is cumbersome in C++, but the idea is simple:
  - Design an interface and make classes use the interface
  - Client code knows interface only, not implementation
Multisets, function objects, inheritance

- Interface used by MultiSet objects, apply function to every object in the set
  - string and count of a set element are passed to \texttt{apply(...)}

```cpp
class MSApplicant
{
  public:
    virtual ~MSApplicant() {}
    virtual void apply(const string & word, int count) = 0;
};
```

- Virtual means “inheritance works”, function called determined at run-time, not compile-time
  - The \texttt{=0} syntax means this that subclasses must implement the function --- subclass implements the interface
What is a function object?

- Encapsulate a function in a class, enforce interface using inheritance or templates
  - Class has state, functions don’t (really)
  - Sorting using different comparison criteria as in extra-credit for Anagram assignment

- In C++ it’s possible to pass a function, actually use pointer to function
  - Syntax is awkward and ugly
  - Functions can’t have state accessible outside the function (how would we count elements in a set, for example)?
  - Limited since return type and parameters fixed, in classes can add other functions
How does interface inheritance help?

- MultiSet code uses interface only to process all set elements

```cpp
void MultiSet::apply(MSApplicant & app) const
// postcondition: app.apply called for all elements in the set
{
    Node * current = myFirst->next; // skip header
    while (current != NULL)
    {
        app.apply(current->myKey, current->myCount);
        current = current->next;
    }
}
```

- How do we count # elements in a set? # distinct elements?
Counting unique words in MultiSet

class MSCounter : public MSApplicant
{
    public:
        MSCounter();
        virtual void apply(const string & word, int count);
        int count() const;  // access the count
    private:
        int myCount;
};

void MSCounter::apply(const string & word, int count)
// postcondition: internal count incremented by 1
{
    myCount++;
}

● If we use list of strings instead of strings/counts, how does apply change? What does body of MSCounter::count() look like? What changes if we count all set elements, not just unique?
Why inheritance?

- Add new shapes easily without changing code
  - Shape * sp = new Circle();
  - Shape * sp2 = 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
Guidelines for using inheritance

● Create a base/super/parent class that specifies the behavior that will be implemented in subclasses
  ➤ Functions in base class should be virtual
    • Often pure virtual (= 0 syntax), interface only
  ➤ 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 with the MultiSet class

- **MultiSet** is a class that uses “add-to-front” to implement multisets
  - Has virtual functions for all set operations, but supplies implementations
  - Has state (doubly-linked list) that will be inherited by subclasses
- Derived classes can change implementation of some functions
  - Add to back, what functions change?
  - Move to front, what functions change?
  - Use a vector instead of linked list, what functions change?
- General guideline, base classes should be interfaces only, not implementation
  - Not followed in MultiSet, leads to less than ideal situation in MultiSetTable class since state inherited
Inheritance guidelines/examples

- **Virtual function binding is determined at run-time**
  - Non-virtual function binding (which one is called) determined at compile time
  - Can’t change which function called if compile-time determined
  - 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
See students.cpp, school.cpp

- Base class student doesn’t have all functions virtual
  - What happens if subclass uses new **name**() function?
    - **name**() bound at compile time, no change observed

- How do subclass objects call parent class code?
  - Use class::function syntax, must know name of parent class

- Why is data protected rather than private?
  - Must be accessed directly in subclasses, why?
  - Not ideal, try to avoid state in base/parent class, can lead to trouble – see MultiSetTable, for example
How to avoid making a copy

- Suppose you want to pass an object into a class, store the object in the private section, but avoid making a copy
  - Why is a copy normally made, what does private section look like, e.g., for a string object or a tvector or a MultiSet
  - To avoid copy, can use reference variable, bound at construction, or pointer, changeable, see code and examples in 12.1.3 – 12.1.6
- What about vector of pointers vs. vector of references?
What’s in a Makefile

- List of dependencies between .h and .cpp files
  ➤ Created by make depend
  ➤ Used to avoid unnecessary re-compilation/re-linking

- Instructions on how to create an executable from several source files
  ➤ Compile source, link source

- Instructions on where libraries and header files are located so that the compiler can find them, include them, link them

- Can’t download, must preserve tabs in many places