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
class MultiSet
{
    public:
        MultiSet();                // construct a multiset

        // mutators
        void insert(const string & word);
        void clear();

        // accessor
        void apply(MSApplicant & app) const;
        int size() const;
        int count(const string & key) const;

    private:
};

● What’s missing? What is apply(..)
We’ll use a linked list

- New nodes added to back, but this will change in other versions: add to front, move to front, tree, ...
- 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?
- Notice return type of private `Find(...)` in multiset.cpp
What’s easy, hard, first?

● Searching the list is straightforward, visit all nodes
  ➤ We need to search for \texttt{insert} and for \texttt{count}, 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 \texttt{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
  ➤ \texttt{count} is simple, why?
  ➤ What is \texttt{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? What does `Current()` return? Internals?
  - 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 element in a `MultiSet`
    - An element is a (string, count) pair
  - Encapsulated in a class with other behavior/state

```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, ...

- C++ inheritance syntax is cumbersome, 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 `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 `=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 additional member 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 != 0)
    {
        app.apply(current->myKey, current->myCount);
        current = current->next;
    }
}
```

- How do we count # elements in a set? # distinct elements?
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 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: trouble
    - What if derived class doesn’t need data?
Consider a modification to MultiSet

- Instead of using prev and next to point to a linear arrangement, use them to divide the universe in half
  - Similar to binary search, everything less goes left, everything greater goes right
  - How do we search?
  - How do we insert?
  - How are lists and trees related?
How do we print all values in a tree?

- **When is root printed?**
  - After left subtree, before right subtree.

```cpp
void Visit(Node * t)
{
    if (t != 0)
    {
        Visit(t->prev);
        cout << t->info << endl;
        Visit(t->next);
    }
}
```

- **Inorder traversal**
Insertion and Find? Complexity?

- How do we search for a value in a tree, starting at root?
  - Can do this both iteratively and recursively, contrast to printing which is very difficult to do iteratively
  - How is insertion similar to search?

- What is complexity of print? Of insertion?
  - Is there a worst case for trees?
  - Do we use best case? Worst case? Average case?

- How do we define worst and average cases?
  - What about add-to-back MultiSet, add-to-front? Move-to-front?
Binary Trees

- Linked lists have efficient insertion and deletion, but inefficient search
  - arrays: search is efficient, insertion and deletion are not
- Binary trees are structures that can be used to yield efficient insertion/deletion and search
  - trees used in many contexts, not just for searching, e.g., expression trees
  - insertion is as efficient as binary search in array, insertion/deletion as efficient as linked list (once node found)
  - binary trees are inherently recursive, difficult to process trees non-recursively, but possible (recursion never required, but often makes coding/algorithms simpler)
Binary trees (continued)

- Binary tree is a structure:
  - empty
  - root node with left and right subtrees
  - terminology: parent, children, leaf node, internal node, depth, height, path
    - link from node N to M then N is parent of M
      - M is child of N
    - leaf node has no children
      - internal node has 1 or 2 children
    - path is sequence of nodes, N₁, N₂, ... Nₖ
      - Nᵢ is parent of Nᵢ₊₁
      - sometimes edge instead of node
    - depth (level) of node: length of root-to-node path
      - level of root is 1
    - height of node: length of longest node-to-leaf path
      - height of tree is height of root
Binary trees (continued)

- Trees can have many shapes: short/bushy, long/stringy
  ➤ if height is $h$, number of nodes is between $h$ and $2^h - 1$
  ➤ single node tree: height = 1, if height = 3

- C++ implementation, similar to doubly-linked list

```c++
struct Tree {
    string info;
    Tree * left;
    Tree * right;
};
```
Tree functions

- Compute height of a tree, what is complexity?

```c
int height(Tree * root)
{
    if (root == 0) return 0;
    else {
        return 1 + max(height(root->left),
                        height(root->right));
    }
}
```

- Modify function to compute number of nodes in a tree, does complexity change?

  ➤ What about computing number of leaf nodes?
Tree traversals

- Different traversals useful in different contexts
  - Inorder prints search tree in order
    - Visit left-subtree, process root, visit right-subtree
  - Preorder useful for reading/writing trees
    - Process root, visit left-subtree, visit right-subtree
  - Postorder useful for destroying trees
    - Visit left-subtree, visit right-subtree, process root
Balanced Trees and Complexity

- A tree is height-balanced if
  - Left and right subtrees are height-balanced
  - Left and right heights differ by at most one

```cpp
bool isBalanced(TreeNode *root) {
    if (root == NULL) return true;
    else {
        return isBalanced(root->left) &&
               isBalanced(root->right) &&
               abs(height(root->left) - height(root->right)) <= 1;
    }
}
```
What is complexity?

- Assume trees are “balanced” in analyzing complexity
  - Roughly half the nodes in each subtree
  - Leads to easier analysis

- How to develop recurrence relation?
  - What is $T(n)$?
  - What other work is done?

- How to solve recurrence relation
  - Plug, expand, plug, expand, find pattern
  - A real proof requires induction to verify that pattern is correct
sidebar: solving recurrence

\[ T(n) = 2T(n/2) + O(n) \]
\[ T(1) = 1 \]

what about 2n? 3n?

\[ T(n) = 2 \left[ 2T(n/4) + n/2 \right] + n \]
\[ = 4 T(n/4) + n + n \]
\[ = 4 \left[ 2T(n/8) + n/4 \right] + 2n \]
\[ = 8T(n/8) + 3n \]
\[ = \ldots \text{ eureka!} \]
\[ = 2^k T(n/2^k) + kn \]

let \( 2^k = n \)
\[ k = \log n, \text{ this yields } 2^{\log n} T(n/2^{\log n}) + n(\log n) \]
\[ n T(1) + n(\log n) \]
\[ O(n \log n) \]
Recognizing Recurrences

- Solve once, re-use in new contexts
  - T must be explicitly identified
  - n must be some measure of size of input/parameter
    - T(n) is the time for quicksort to run on an n-element vector

\[
\begin{align*}
T(n) &= T(n-1) + O(n) & \text{sequential search} & \mathcal{O}(n) \\
T(n) &= T(n/2) + O(1) & \text{binary search} & \mathcal{O}(1) \\
T(n) &= 2T(n/2) + O(1) & \text{tree traversal} & \mathcal{O}(n) \\
T(n) &= 2T(n/2) + O(n) & \text{quicksort} & \mathcal{O}(n) \\
\end{align*}
\]

- Remember the algorithm, re-derive complexity