Assignment #5

Due dates:

Tests (for Parts 1 and 2): Monday, 18 March 2024, 8:00 pm
Code (for Parts 1 and 2): Friday, 22 March 2024, 8:00 pm

- *For all programming questions below, write your solutions in the dialect of C++ used in class. For Part 1, you may `#include` the following libraries, and no others: `iostream`, `string`, `vector`. For Part 2, you may `#include` the following libraries, and no others: `iostream`, `string`, `map`, `cassert`, `cctype`.*
- *Store your solutions in the provided files named `a5p1.cc` and `a5p2.cc`. For part 2 you can (and may need to) modify `a5p2.h`. The `submit_project.sh` script has been updated to submit `a5p2.h` as well.*
- *For part 1, test coverage will be assessed by running your tests on the staff code as usual. The required coverage for `a5p1t` is 100%.*
- *For part 2, the provided public test cases already achieve 100% on our solution. There is still a `a5p2t` project on Marmoset which you can use to check the correctness of your tests, but it is not worth any marks. Your final grade for this assignment will be:*

$$(\texttt{a5p1c\_grade} \times 0.9 + \texttt{a5p1t\_grade} \times 0.1) \times 0.5 + \texttt{a5p2c\_grade} \times 0.5$$

- *You need to write your unit tests using GTest framework. Give your tests some meaningful names to understand where you should look in the case of test failures. You could use the Given-When-Then approach as a guide for unit test naming.*

Key ideas in this assignment

- Object-oriented programming
- Tree-like data structures
- `maps`
- Recursion

Part I: Inheritance - 50%

You are to create a simple program that manages flocks of sheep. You will need to define several C++ classes to do this: `Animal`, `Sheep`, `Dog`, and `Flock`. `Animal` is an abstract base class with two descendants: `Sheep` and `Dog`. The `Animal` class has the following variables and methods defined in `a5p1.h`:

```
class Animal {
    public:
        virtual ˜Animal ();
        virtual std::string speak () const = 0 ;
    protected:
        Animal (std::string name);
        std::string getName () const;
    private:
        std::string name;
};
```

Note that `Animal` has only one constructor, and it takes a string argument. Note also that this constructor is `protected` rather than `public`. Why? `Animal` is an *abstract base class*; no normal client will be creating instances of it, since it is abstract. The only way in which its constructor can be invoked is by a descendant class, which calls the `Animal` constructor to initialize the parts that are common to all `Animals`. This is an example of a fairly common *idiom* in object-oriented programming; there are many, many more as you'll discover in CS247.

`Sheep` and `Dog` should inherit from `Animal`; they should each define a single constructor of one string argument (as in `Animal`) and should each provide an appropriate implementation of `speak()` (see the example output at the end). `Sheep`, `Dog`, and `Animal` should each define a (likely trivial) virtual destructor.

A `Flock` consists of a single `Dog` and zero or more `Sheep`. The `Flock` class has the following variables and methods:

```
    class Flock {
        public:
            Flock (std::string dogName);
            virtual ~Flock ();
            void addSheep (std::string name);
            std::string soundOff () const ;
        private:
            Dog *dog;
            std::vector<Sheep*> sheepList;
    };
```

You create a `Flock` by passing in the `Dog`'s name at instantiation. You can then add sheep one at a time by passing in each sheep's name to `addSheep()`. Assume all names are distinct (*i.e.,* don't bother thinking about this). We'll discuss the variables `dog` and `sheepList` below. You will also need to define a destructor.

What does the `Dog` say?

The `speak` method of `Dog` returns a string that starts with 4 (leading) space characters, followed by the `Dog`'s message (don't add a `\n` newline character to the return value of `Dog::speak()`, that's a job for `Flock::soundOff()`). So for a Dog named Bolt, we get this (canine) return value (the symbol ␣ indicates a space character):

```
␣␣␣␣Dog␣Bolt␣says␣"woof".
```

When a `Sheep` named `Shaun` speaks, you get a similar (but ovine) value:

```
␣␣␣␣Sheep␣Shaun␣says␣"baaa".
```

When a `Flock` performs a `soundOff`, it returns a single `string` containing multiple embedded newlines; so, when you print such a `string`, it spans several lines of output. The first line indicates how many `Sheep` are there in the flock. Then the `Dog` speaks. Then the `Sheep` speak one by one in the order in which they were inserted. For example:

```
Flock* flock1 = new Flock("Gromit");
flock1->addSheep("Shaun");
flock1->addSheep("Sam");
cout << flock1->soundOff();

// Expected output below:
The flock of 2 sheep speaks!
    Dog Gromit says "woof".
    Sheep Shaun says "baaa".
    Sheep Sam says "baaa".
```

Your returned strings have to follow the exact syntax. Use `a5p1t` to make sure your tests are checking for the correct output.

Implementation notes

Within `Flock`, the `Dog` is represented via a `Dog*`, while the list of `Sheep` is represented using a `vector` of `Sheep*`. To initialize the `vector` of `Sheep` pointers (the `vector` itself is a "direct sub-object" of the `flock`, even tho it contains pointers to objects on the heap), you may call the `vector` default constructor in the constructor of `Flock`, or you can just rely on the fact that sub-objects are created automatically using their default constructors of no arguments if not otherwise initialized.

When you want to add a new `Sheep`, use a pointer and `new` to instantiate it, and add it to the end of the `vector` of `Sheep` pointers using the `push_back()` method of `vector`.

You'll have to think about how to design the destructor. Keep in mind that the `vector` is a direct subobject of your `Flock` which exists on the stack, whereas you have pointers to the `Dog` and `Sheep` which exist in the heap. Assume that each `Flock` "owns" its `Dog` and `Sheep` and is responsible for cleaning them up when the time comes.

A note on C++ strings

To write a string of multiple lines, we can do it in two ways. The first way is:

```
string multiple_lines = "Hello\n    Goodbye\n";
```

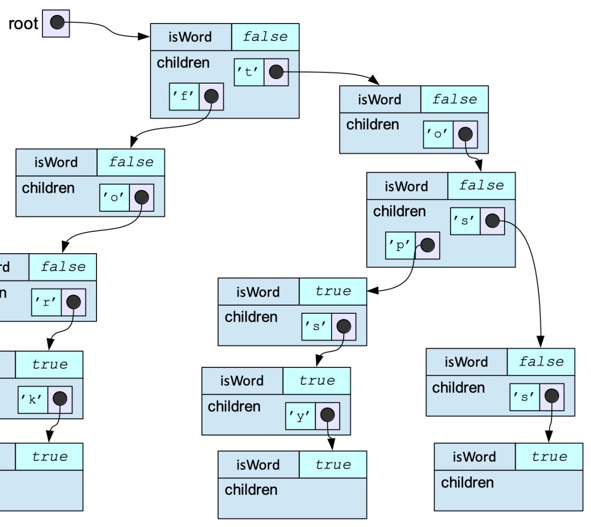
This is fine, but it looks a little ugly. A second way looks like this:

```
string multiple_lines = "Hello\n"
                        "    Goodbye\n";
// Note that there is no semicolon at the end of the first line
```

The second way is a (relatively) recent C++ feature: adjacent string literals are concatenated by the compiler! Notice how this makes it clearer where each line begins/ends, as well as exactly how many leading spaces there are in each line.

Part II: Lexicon Tree - 50%

We are going to finish implementing a class named `LexTree` which implements a *Lexicon* (a simple catalog of words) using a kind of N-ary tree. The diagram below shows an example of a `LexTree` into which (only) the following words have been added: `for fork top tops topsy toss`. Notice that although `"to"` is a perfectly valid English word, we have not added it to the `LexTree` yet, and so the boolean field `isWord` pointed to by the `map` element `'o'` is currently set to `false`. If you were now to add `"to"` to the `LexTree`, this field would be set to `true`, but no other changes to the `LexTree` would be necessary.



In the example above, we can see that each `LexTree` node has a boolean flag, `isWord`, and a set of children which we implement as a C++ `map`. Each `map` element represents the next letter in a word that has been entered into the `LexTree`. The boolean `isWord` flag of a given node indicates if the *unique* chain of letters corresponding to the keys that descend through the tree to the node in question forms a word that was entered into the `LexTree`. This means that for the root node of the `LexTree`, its `map` contains keys for all the *unique* first letters of the words which have been added into the `LexTree`, while its `isWord` field indicates if the the "word" `""` (i.e. the empty string) has been added to the `LexTree`.

To determine if a given `string`, call it `s`, is in our `LexTree`, we start at the root and see if the first `char` in `s` is one of the keys of `children` (the root node's `map`). If it is, then we follow the pointer associated with that key to the next `LexTree` node in the chain. We then check if the next character in `s` is a member of that `LexTree` node's children, and so on. We stop when either (a) we don't find a link to the next char of `s`, in which case we return `false`, or (b) we run out of characters in `s` to look for. In the latter case, we then look at the link to the `LexTree` node pointed to by the last map element and return the value of `isWord` we find there as the result of the search.

We've chosen to use a C++ `map` to implement the links between nodes, which you may recall is implemented using a kind of BST. Note that if we were to use a C++ `unordered_map` instead of a `map`, then both insertion and lookup (`addWord` and `hasWord`) would be linear in the number of characters in the key value; that is, they wouldn't depend on the number of elements in the tree! This is no worse than hashing since a hash function typically uses all of the characters of a string key to calculate the hash value. However, if we did that, then the Lexicon would not be sorted since `unordered_maps` are implemented as hash tables. In the end, we decided to use a `map` for two reasons: first, a Lexicon seems like it ought to be sorted, and second, the output from your program would be harder to test with Marmoset since the ordering of `toString()` would be non-deterministic.

We have also chosen to make our `LexTree` *case-insensitive* (the words "Apple", "apple" and "aPpLe" are all treated as the same) by first converting any given strings to lowercase before processing them. This means that the keys in the `LexTree` should all be lowercase characters. The C++ standard library contains a handy little function called `tolower` which will help us with this. It's in the `#include <cctype>` header which we've added to `a5p2.cc`.

The functions you are expected to implement are:

- `LexTree::LexTree()` — The default constructor for `LexTree`. It should set `isWord` to false and call the `std::map` default constructor for `children`.
- `LexTree::~LexTree()` — The destructor for `LexTree`. It should clean up all heap-allocated memory associated with the given `LexTree` instance.
- `void LexTree::addWord(const string& s)` — Adds the given word into the `LexTree`.
- `bool LexTree::hasWord(const string& s) const` — Checks if the given word is contained in the `LexTree`. Notice how this method is declared as `const`, and so it must take the "`const` pledge".
- `string LexTree::toString() const` — Returns a string containing all the words in the `LexTree` in lexicographical order (separated by newline characters). This method also takes the "`const` pledge".
- `bool LexTree::isValid() const` — Checks if the `LexTree` is valid. If it is valid returns `true`, if it isn't an assertion should be triggered. This function should never return `false`. More precisely, we expect your `isValid` to check the following two properties:
  1. Every key in the `LexTree` should be a lowercase letter (there's an `islower` function in the `<cctype>` header which you may find useful here).
  2. Unless the `LexTree` is empty, every leaf node should have `isWord == true` (food for thought: why is this true?).

  There will be secret tests for your `isValid` function on this assignment. The `LexTree` inputs they test will either be valid or will contain at least one of the above issues.

Notes:

- The strings passed to your `addWord` and `hasWord` functions by Marmoset will not contain any *non-alphabetical* characters. In other words, they will only contain the characters {`'A', 'B', ..., 'Z', 'a', 'b', ..., 'z'`}.

- The first thing you'll want to implement is the default constructor since without it none of your tests will work. Then, similar to Assignment 3, before implementing the other functions you should start by implementing `isValid`. That way you can use it as a precondition/postcondition in your other functions.

`a5p2.h` - What is going on in there?

For part II of this assignment you are allowed to modify `a5p2.h`. Here is what the contents of `a5p2.h` look like:

```cpp
// Don't change any of the includes
#include <string>
#include <map>
#include "gtest/gtest.h"

class LexTree {
    public:
        // Don't modify the public interface. If you change it
        // then your solution probably won't compile on Marmoset
        LexTree();
        virtual ~LexTree();
        void addWord (const std::string& s);
        bool hasWord (const std::string& s) const;
        std::string toString() const;

    private:
        // Fields: You may add new fields if you'd like, but don't
        // change isWord or children since then you'll break the
        // Marmoset whitebox tests
        bool isWord;
        std::map<char,LexTree*> children;

        // Private Methods: The is isValid method needs too stay
        // as-is (since Marmoset needs to be able to call it),
        // but feel free to add some helper functions here if you
        // want.
        bool isValid () const;
        // void yourFunction(std::string example) const;
        // ... etc ...

    // Friends
    FRIEND_TEST(defaultCtorPub, defaultCtorShouldMakeEmptyLex);
    FRIEND_TEST(isValidPub, emptyTreeShouldBeValid);
    FRIEND_TEST(isValidPub, checkAllLowerCase1);
    FRIEND_TEST(isValidPub, checkAllLowerCase2);
    FRIEND_TEST(isValidPub, leafNodeShouldBeWords);
    FRIEND_TEST(addWordPub, addAWordAndCheckNodes);
    FRIEND_TEST(hasWordPub, addAWordAndCheckIt);
    FRIEND_TEST(toStringPub, addOneWordAndPrint);
    FRIEND_TEST(toStringPub, addTwoWordsAndPrint);
    // FRIEND_TEST(yourWhiteboxTest, yourTestName);
    // ... etc ...

    // When you make new whitebox LexTree test cases you will need
    // to add them as friends here so that they can access LexTree's
    // private members.
    // On the other hand, if you make new blackbox test cases which
    // only use the public API then you should NOT make them friends
    // with the LexTree class.
};
```

The first thing that pops out here is all of those calls to `FRIEND_TEST(..., ...)` at the bottom. Those are there to give the `gtest` tests access to `LexTree`'s private fields and methods so that they can perform white box testing. When you make your own white box tests, you will have to add them as friends of the `LexTree` class so that they have access to its internal parts. More info on the `FRIEND_TEST` macro will be covered during the week 8 tutorial.

The tree nature of `LexTree` makes it work well with recursive solutions. Because of this, you may wish to create some recursive helper functions for yourself. These should be added as private methods alongside `isValid`.