

Assignment 1: Textr

Ship Date: January 27, 2005, 11:59pm

The purpose of Assignment 1 is to get you acquainted with C++ or refresh your memory if it has been a while. This assignment can be accomplished using only the material covered in the first two lectures. However, if you are already familiar with material that hasn't been covered (such as the STL), feel free to use that knowledge in this assignment.

Textr: It's not a typo, it's trendy naming

In 2004, a web-based photo sharing service called Flickr (pronounced "flicker") was launched. It changed the web in two very important ways. First, it helped establish the notion of *tagging* as an important concept in the so-called "Web 2.0" movement. By tagging photos, Flickr users associate keywords with pictures, making it easier for other users to find pictures of interest. For example, if I upload a picture of my cat and tag it as *cat* and you do the same with a picture of your cat, somebody else can come to site and see a slideshow of all the cat pictures that have been uploaded.

The second way Flickr changed things was by ending the trend of naming things with *-ster* (Napster, Grokster, etc.) and ushering in the much more efficient *-r* naming convention (Flickr, Talkr, etc.)

Textr (pronounced "texter") is a C++ program that lets users input blurbs of text and categorize them through tagging. Users can also search by tag, and inform the system that two tags have the same meaning. Textr isn't quite as advanced as Flickr. Most notably, it's not a web service. It probably won't get bought out by Yahoo! But it *will* make use of the basic language features we've discussed in class.

A Sample Textr Session

Textr has a simple menu-based user interface, not unlike the interface used by the Employee example in Chapter 1. In fact, if you get stuck in this assignment, that would be an excellent example to consult.

The following transcript shows a user entering two blurbs into textr, assigning and removing tags, searching, and grouping tags together. When your program is complete, you should be able to mimic this session:

Textr

- 1) Add Blurb
- 2) Tag a Blurb
- 3) Remove Tag from a Blurb
- 4) Search by Tag
- 5) Consolidate Two Tags
- 0) Quit

---> 1

Blurb: **Now is the time for all good men to come to the aid of the party.**

Textr

[omitted for brevity]

---> 2

Select a blurb:

- 0) Now is the time for all good men to come to the aid of the party.

Blurb: 0

Tag to add: **funny**

Textr

[omitted for brevity]

---> 2

Select a blurb:

- 0) Now is the time for all good men to come to the aid of the party.

Blurb: 0

Tag to add: **historic**

Textr

[omitted for brevity]

---> 3

Select a blurb:

0) Now is the time for all good men to come to the aid of the party.

Blurb: **0**

Tag to remove: **historic**

Tag removed.

Textr

[omitted for brevity]

---> **1**

Blurb: CS193D is the best class ever.

Textr

[omitted for brevity]

---> **2**

Select a blurb:

0) Now is the time for all good men to come to the aid of the party.

1) CS193D is the best class ever.

Blurb: **1**

Tag to add: **ridiculous**

Textr

[omitted for brevity]

---> **5**

Tag 1: **funny**

Tag 2: **ridiculous**

funny and ridiculous have been consolidated.

Textr

[omitted for brevity]

---> **4**

Tag to search for: **funny**

Now is the time for all good men to come to the aid of the party.
CS193D is the best class ever.

```
Textr
-----
[omitted for brevity]

---> 0
```

The Blurb Class

The centerpiece of Textr is the Blurb class. A Blurb is created with a string and provides facilities to add a tag, remove a tag, get the blurb text, and inquire whether the Blurb has a particular tag. Tags are represented by their ID, which is an int assigned by the Database class (see below).

Below is the public interface for Blurb. Your Blurb class should conform to this public interface so that our automated tests can be run. However, you can add whatever additional public, protected, and private methods you need in addition.

```
class Blurb {
public:
    Blurb(const std::string& inText);
    ~Blurb();

    void addTagID(int inID);
    bool removeTag(int inID);
    bool hasTagID(int inID);

    const std::string& getText();

    // for testing only!
    int testing_getNumTags();
};
```

Behind the scenes, your Blurb class can do whatever you want subject to the following conditions:

Blurbs can have an unlimited number of tags assigned to them, subject to the limits of memory. This most likely means you'll want to use a dynamically allocated array that gets resized as necessary.

When destroyed, Blurbs must correctly clean up any memory that they allocate.

As you are developing your Blurb class, we suggest you test it in isolation by writing a throwaway `main()` that tests each public method in several different ways. For this assignment, your tests won't be graded, but future assignments will require unit tests as part of your solution. Be sure to remove or comment out this throwaway `main()` so that it doesn't conflict with your real `main()` later on.

The Database Class

The Database class is the central repository for Blurbs and tags. Here's how tags work in Texttr: when a user tags a blurb, the program asks the Database for the ID of the tag by calling `getOrCreateTagIDByName()`. If the Database has already seen the tag (it already has an ID), the existing ID is returned. If not, the tag is stored in the database and a new ID is assigned. If there's no room for the new tag, an exception is thrown.

The Database also stores the Blurbs that were created in Texttr and performs the logic of searching for Blurbs by their tag ID. When you add a Blurb to the Database, you give the Database a pointer to the heap-allocated Blurb (a `Blurb*`). The Database is then responsible for managing and eventually releasing the memory associated with the Blurb. Ideally, you would just pass copies or references around, but since we haven't covered copy constructors in class and C++ doesn't allow arrays of references, that would prove difficult. If the Blurb cannot be stored in the database (e.g. there is no room), an exception is thrown.

You will also notice that the Database returns lists of Blurbs as an array of Blurb pointers (`Blurb**`) when users call `getAllBlurbs()` or `getBlurbsByTagID()`. A reference parameter is used to return the size of the array to the caller. If this seems lame, it should. Later on, when we cover STL vectors, you'll learn about a much better way of doing things like this.

Finally, keep in mind that the database we're talking about is an in-memory database. You don't need to worry about writing data to disk. When the program exits, all the tags and blurbs are lost – that's fine!

Consolidation

The Database also supports consolidation of tags. Since users can enter whatever tags they want, there is no way to guarantee that everybody uses the same tag to describe the same thing. For example, suppose I enter a Shakespeare sonnet and tag it as *poem*. Then you add one and tag it as *poetry*. We've failed to inform the system that these two blurbs are related. To solve this problem, a user can consolidate the tags, indicating that they are synonyms.

Suppose we decide to consolidate *poem* and *poetry*. After we do that, searching by *poetry* and searching by *poem* should yield the same results. This applies to both existing blurbs and new blurbs.

There are many ways to implement the synonym functionality of consolidation. You could apply both tags behind the scenes or you could maintain some sort of lookup table. My suggestion is to keep it simple. Simply map the two tags to the same tag ID and everything should work fine. In other words, looking up the ID for the tag *poem* might return the same ID you get when you look up *poetry*.

The Database API

Below is the public interface for Database. As you can see, the class contains public constants for the maximum number of Blurbs and the maximum number of tags. You can use these constants to create fixed-size arrays in the Database class if you wish. The arrays don't need to be dynamically sized like the tag list in the Blurb class did. There's also a testing method to get the number of blurbs in the database. We'll use this when testing your implementation.

```
class Database {
public:
    Database();
    ~Database();

    // blurb operations

    // Once added, the database owns the memory for a blurb. Throws
    // an exception if the Blurb can't be added
    void addBlurb(Blurb* inBlurb);
    Blurb** getAllBlurbs(int& outNumBlurbs);
    const Blurb** getBlurbsByTagID(int inTagID, int& outNumBlurbs);

    // tag operations
    // Throws an exception if we're out of room for tags
    int getOrCreateTagIDByName(const std::string& inTagName);
    void consolidateTags(int inTag1, int inTag2);

    static const int kMaxTags = 100;
    static const int kMaxBlurbs = 100;
```

```
// testing
int testing_getNumBlurbs();
};
```

Again, the implementation behind the scenes is up to you.

The User Interface

The Textr user interface is a simple menu-driven system, as shown above. This can be implemented as a simple C-style collection of functions with a main, or you can build a class around it.

Please try to adhere as much as possible to the syntax of the user interface shown above – this will make it easier for us to grade. Of course, you can add extra features if you want, but please put them at the bottom of the menu so that the numbers don't change for the standard features.

Your user interface doesn't need to be particularly forgiving. We won't try entering letters instead of numbers, multi-word tag names, or long blurbs, for example. Most of our testing will be done on the classes behind the scenes. So don't worry about bad input. As described above, the Database class needs to throw exceptions in certain cases. You don't need to worry about catching these, or other, exceptions. It's fine if the program terminates.

When asking for a item in a menu from the user, you may find the following function helpful:

```
int simpleGetInteger()
{
    int selection;

    while (true) {
        cin >> selection;

        if (cin.good()) {
            string temp;
            getline(cin, temp);
            return selection;
        } else {
            cin.clear();
            string temp;
            getline(cin, temp);
        }
    }
}
```

```
        cout << "That was not an integer. Please try again: ";
    }
}
}
```

Getting Started

We'll post the header files shown above to `/usr/class/cs193d/assignments/` as a starting point. Feel free to modify them as needed, but be sure to keep the same public interface so that our automated tests can compile against your code.

We didn't include a Makefile with this assignment. You should compile your code manually, like this:

```
g++ *.cpp
```

Assuming your code all compiles and links, you can run the program by typing:

```
./a.out
```

Deliverables

Future assignments will require unit tests and detailed documentation. However, for this assignment, we simply need your code plus a simple README file containing your name, email address, and any comments about the assignment or your solution. When you're ready to turn in your assignment, type `"/usr/class/cs193d/bin/submit"` on the elaines and follow the directions.

Unlike other assignments, we won't factor documentation or testing into your grade, but the following things will be considered:

Correctness. We will run a suite of our own unit tests against the Database and Blurb classes.

Functionality. We will go through a set of prescribed steps through your user interface.

Scalability. We will inspect your code for memory leaks and will ensure that going over stated limits (such as the maximum number of tags) is detected and reported.

Readability. We will look over your code expecting to see appropriate variable and method names, good functional decomposition, and appropriate use of language features.