Assignment #3

Due Date 1: Friday, October 25, 2024, 5:00 pm EDT (30% marks)
Due Date 2: Friday, November 1, 2024, 5:00 pm EDT (70% marks)

Learning objectives:

- C++ classes, constructors, destructors, and operators
- The spaceship operator
- Object-oriented programming, invariants, and encapsulation
- Iterators

- **Questions 1a, 2a, and 3a are due on Due Date 1; questions 1b, 2b, 3b, are due on Due Date 2.**
- See A1 for notes on: test suites, undefined behaviour, test partners, using C++-style I/O and memory management, hand-marking,
- For all questions, use C++20 `imports`; do *not* use `#include`. Compile the system headers with `g++20h` and compile your program files with `g++20`. Put the module interface into a file named something like `Balloon.cc` and put the module implementation in a file named `Balloon-impl.cc`.
- In all cases, your test suites should be testing the functionality that you are responsible for. **Do not submit test cases whose sole purpose is to verify the behaviour of the test harness.** There is no value in that, and there are no marks allocated for that.
- You may `import` the following C++ libraries (and no others!) for the current assignment: `iostream`, `fstream`, `sstream`, `string`, `utility`, `algorithm`, and `compare`. Marmoset will be setup to **reject** submissions that use C-style I/O or memory management, or libraries other than the ones specified above.
- We have provided some code and sample executables under the appropriate `a3` subdirectories. **These executables have been compiled in the CS student environment and will not run anywhere else.**
- **Before asking on Piazza, see if your question can be answered by the sample executables we provide. You are not permitted to ask any public questions on Piazza about what the programs that make up the assignment are supposed to do.** A major part of this assignment involves designing test cases, and questions that ask what the programs should do in one case or another will give away potential test cases to the rest of the class. Questions found in violation of this rule will be marked private or deleted; repeat offences could be subject to discipline.

Question 1

(45% of DD1; 45% of DD2)

A *bag* (also called a *multiset* or a *bunch*, depending on who taught you set theory and when) is a data structure that is similar to a mathematical set, except that a bag can contain multiple instances of the same value. Here, we are going to assume that the element type is a `std::string`, so our data structure will be called a `StringBag`. Here's are a few examples `StringBag`s using mathematical-like notation:

```
sb1 == { ("alpha", 1), ("beta", 3), ("gamma", 2), }
sb2 == { ("mink", 2), ("zebra", 1), ("beta", 2), }
sb3 == { ("beta", 4), ("mink", 1), ("horse", 1), }
sb4 == { (DVD, 1), (beta, 4), (vhs, 2), }
```

Note that the ordering of the pairs within the curly brackets is not significant. We're going to provide an overload of `operator<<` to let you print out `StringBag` values in the format shown above (i.e., the RHS of the equations above.)[1]

The *arity* of a value is the number of times is occurs in the bag. For `sb1` the arity of `"alpha"` is 1, `"gamma"` is 2, and `"omega"` is 0 since `"omega"` doesn't occur in the bag. Also, we distinguish between the *number of elements* in a `StringBag` (i.e., treating duplicates as individuals to be counted) and the *number of values* (i.e., ignoring duplicates). Our example `sb1` has 6 elements but only 3 values.

We will provide you with the declaration of the class `StringBag` as well as the helper struct `StringBag::Node`. We will also provide you with implementations of two print functions, so that your output will match ours (see information about `operation<<` and `StringBag::debugPrint` below.) Your job is to (1) provide a set of tests for this class (deadline #1) and (2) to provide implementations of all of the remaining methods, including constructors, destructors, and overloaded operators (deadline #2).

There are many ways we could implement this data structure, but we're going to use an unsorted linked list, where there is one `Node` for each value plus an `arity` field of type `size_t` that indicates the number of times that value occurs in the `StringBag`. Because the list is unsorted, you're going to find it useful to define a `find` function that takes a string `s` and returns a pointer to the `Node` for that string value, if it exists (and returns `nullptr` if it does not). Note that you may *not* use any of the container data structures from the C++ Standard Library like `map`, `vector`, `set`, `multiset`, or `unordered_multiset`. The whole point of this question is to give you some experience in implementing a library-like data structure yourself.

Here's a more detailed description of the methods and operators you'll have to implement:

- Default, copy, and move constructors, plus a destructor.

  ```
  StringBag::StringBag ();
  StringBag::StringBag (const StringBag& otherSB);
  StringBag::StringBag (StringBag && otherSB);
  StringBag::~StringBag ();
  ```

- Two methods to print `StringBag`s, both of which we will provide definitions for. One will be an overload of the global `operator<<(ostream & os)` — declared as a friend of `StringBag` so it can access the list of Nodes — and the other will be a method called `debugPrint`.

  ```
  void StringBag::debugPrint (ostream & os) const;
  friend ostream& operator<<(ostream &os, const StringBag & sb);
  ```

  While the output will be similar for both, there are a few small differences:

  - `operator<<` will print the bag in a compact format all on one line of output, and it will print only normal nodes (i.e., non-zombies).

[1] To make printing easier, we're going to put a comma after each pair, including the last one.

– `debugPrint` will first print the number of elements and the number of values in the `StringBag`; then, it will print the elements as pairs, one line at a time. Also, it will print both normal nodes and zombie nodes.

So the code below would result in output as described in the comments.

```
// Note that the ordering of the pairs within a StringBag is arbitrary
cerr << "sb4 == " << sb4 << endl;
// Output:  sb4 == { (DVD, 1), (beta, 4), (vhs, 2), }
sb4.removeAll("beta");
cerr << "sb4 == " << sb4 << endl;
// Output: sb4 == { (DVD, 1), (vhs, 2), }
sb4.debugPrint(cerr);
// Output: StringBag with 3 elements and 2 values:
// {
//     (DVD, 1),
//     (beta, 0),
//     (vhs, 2),
// }
```

Again, we're going to provide these for you, so just think about how you might use these functions to help with your debugging.

- Methods to return the arity of a string value, the number of elements, and the number of values of a `StringBag` instance.

```
size_t StringBag::arity(const string &s) const;
size_t StringBag::getNumElements() const;
size_t StringBag::getNumValues() const;
```

- Methods to add/remove individual string elements: `add`, `remove`, and `removeAll`.

```
void StringBag::add (const string & s);
void StringBag::remove (const string & s);
void StringBag::removeAll (const string & s);
```

The first time that you add a value to a `StringBag`, you'll have to create a `Node` for it and add it to the linked list; after that, you just need to find the `Node` and increment the `arity` by one.

`remove` deletes a single instance of a string value from the `StringBag`, while `removeAll` removes all of the instances of that value. If the element you're trying to remove doesn't actually occur in the `StringBag`, then the attempted removal is simply a no-op, rather than an error. If the string does occur in the bag (i.e., it has an arity of at least one), then all you need to do is decrease the arity. If the arity of a value reaches zero, we're just going to leave the `Node` in place as a kind of *zombie* placeholder. If that value is added back to the `StringBag`, all you have to do is find the `Node` and increment its `arity` back up to one.

`StringBag::removeAll` removes all of the instances of a string value. This means the following should be true:

```
sb1.remove("beta");
cout << sb1.artity("beta") << endl   // returns 2
sb1.removeAll("gamma);
cout << sb1.artity("gamma") << endl  // returns 0
```

- `StringBag` union and difference operators, which we're going to implement as overloads of `operator+` and `operator-` respectively.

```
StringBag StringBag::operator+(const StringBag& otherSB) const;
StringBag StringBag::operator-(const StringBag & otherSB) const;
```

The output from the following lines should be as in the comment that follows.

```
cout << sb2+sb3 << endl;  // { (horse, 1), (mink, 3), (zebra, 1), (beta, 6), }
cout << sb2-sb3 << endl;  // { (mink, 1), (zebra, 1), }
```

- Overloads of `operator+=` and `operator-=` to provide shortcut notation to take the union / difference with another `StringBag`.

```
StringBag& StringBag::operator+=(const StringBag& otherSB);
StringBag& StringBag::operator-=(const StringBag& otherSB);
```

- A definition of `operator==`, which returns true if and only if the two `StringBag`s have the same set of string values with the same `arity` for each string value. Note that we ignore zombie `Node`s here.

```
bool StringBag::operator==(const StringBag & otherSB) const;
```

- Copy and move assignment operators. To make life simpler, copy/move all of the `Node`s including the zombies.

```
StringBag& StringBag::operator=(const StringBag & otherSB);
StringBag& StringBag::operator=(StringBag && otherSB);
```

- A `StringBag::dezombify` method that deletes all of the zombie nodes in a `StringBag`. The idea is that you would run this yourself every so often, a bit like how garbage collection works in Java and C#.

```
void StringBag::dezombify ();
```

For example, let's consider `sb4` again, assuming that `sb4.removeAll("beta")` has been performed already:

```
sb4.debugPrint(cerr);
// Output: StringBag with 3 elements and 2 values:
// {
//     (DVD, 1),
//     (beta, 0),
//     (vhs, 2),
// }
sb4.dezombify();
sb4.debugPrint(cerr);
// Output:  StringBag with 3 elements and 2 values:
// {
//     (DVD, 1),
//     (vhs, 2),
// }
```

- A `private` function called `find` that returns a pointer to the `Node` corresponding to a string value, if there is one in the `StringBag`; otherwise, it returns `nullptr`.

```
Node* StringBag::find (const string & s) const;
```

Notes:

- A test harness is available in the file `a3q1.cc`, which you will find in your `a3/codeForStudents/q1` directory. Make sure you read and understand this test harness, as you will need to know what it does in order to structure your test suite. A sample test case that can be run using the test harness and the provided sample executable is also provided. Note that we may use a different test harness to evaluate the code you submit on Due Date 2 (if your functions work properly, it should not matter what test harness we use).
- We will provide files for the module interface (`StringBag.cc`) and implementation (`StringBag-impl.cc`) plus a main program `a3q1.cc`. **You should not change the module interface file.** This means you may not change the interface of the `StringBag` class or its helper `Node` struct or the overload of `operator<<` that we provided.
- You must create and submit a `Makefile` that correctly implements the build process for your program. Keep in mind that a module interface file must be compiled before any files that `import` it. The executable that gets built should be called `a3q1`.

a) **Due on Due Date 1:** Design a test suite for this program, using the `main` function provided in the test harness. Call your suite file `suite-sb.txt`. Zip your suite file, together with the associated `.in` and `.out` files, into the file `a3q1a.zip`. (This program does not accept command line arguments, so there will be no .args files.)

b) **Due on Due Date 2:** Submit the file `StringBag-impl.cc` containing the implementation of the remaining functions of the class `StringBag`, as well as the `Makefile` that builds your system.

Question 2

(20% of DD1; 20% of DD2)

You have been given some starter code in `address.cc` and `address-impl.cc` that defines an `Address` class:

```
class Address {
  public:
    enum class Direction { EAST, NORTH, SOUTH, WEST, NONE };

    Address(std::size_t streetNumber, const std::string &streetName, const std::string &city,
        const std::string &unit = "", Direction direction = Direction::NONE );

    std::strong_ordering operator<=>(const Address &other) const;

  private:
    std::size_t streetNumber;
    std::string unit, streetName, city;
    Direction direction;

    // Helper methods
    Direction convert(const std::string &direction) const;
    std::string convert(const Direction direction) const;

    // Friend declarations
    friend std::istream& operator>>(std::istream &in, Address &addr);
    friend std::ostream& operator<<(std::ostream &out, const Address &addr);
};

std::istream& operator>>(std::istream &in, Address &addr);
std::ostream& operator<<(std::ostream &out, const Address &addr);
```

You have also been provided with a simple test harness, `a3q2.cc`, and a sample input file, `sample.in`. Make sure that you read through the test harness carefully to understand what it does.

Implement the "spaceship" operation, `operator<=>`, for `Address`. Its behaviour should match that of the provided sample executable.

Implementation notes:

- No error-checking is required.
- All strings may technically be empty, though it's not very useful, since an empty string is lexicographically considered to be "less" than a non-empty string for comparison purposes.
- Since `std::size_t` is an unsigned integer, street numbers cannot be negative.
- The order of comparison, if previous components are the same, is: city, street name, street direction, street number, and unit number.
- The "unit" information and the street direction are *optional* and may not be provided for a particular address,
- The order that the directions are specified in `Address::Direction` specify their relative ordering i.e. `Address::Direction::EAST` < `Address::Direction::NORTH` < `Address::Direction::SOUTH` < `Address::Direction::WEST` < `Address::Direction::NONE`.

a) **Due on Due Date 1:** Design a test suite for this program (call the suite file `suiteq2.txt` and zip the suite along with all needed `.in` and `.out` files into `a3q2a.zip`).

b) **Due on Due Date 2:** Implement this program in C++. Submit your `address.cc`, `address-impl.cc` and `a3q2.cc` files that make up your program in your zip file, `a3q2b.zip`.

Question 3

(35% of DD1; 35% of DD2)

In this question, you are given an implementation of a `TierList` class: a ranked collection of *tiers*, each tier containing a set of elements ranked at that tier. Examples of tier lists can be found at https://tiermaker.com/.

Traditionally, the tiers in a tier list are ranked S to F, with S being the best tier and F being the worst tier. For simplicity, we will order our tiers by number, with 0 representing the best tier, 1 representing the second-best tier, 2 representing the third-best tier, etc.

Your task is to implement an *iterator* for the `TierList` class that will iterate through the tier list from the items in the best tier to the items in the worst tier.

In addition to the standard operations an iterator provides, you will also implement overloads of the `<<` and `>>` operators for the iterator. These will return a new iterator pointing to the start of the tier that is `n` tiers before/after (respectively) the tier of the current item.

Starter code and method signatures have been provided in `tierlist.cc` and `tierlist-impl.cc`, along with a sample executable. **You may not change the contents of `tierlist.cc`, other than by adding your own private methods, variables, and comments, i.e., the interface must stay exactly the same.**

The test harness `a3q3.cc` is provided, with which you may interact with your tier list for testing purposes. **The test harness is not robust and you are not to devise tests for it, just for the `TierList` class. Do not change this file.**

Implementation notes:

- It is possible that one or more tiers in the tier list could be empty. You should never end up iterating over an empty list if you've set up your iterator correctly.
  - `TierList::begin()` either sets the iterator to the first item of the *first non-empty tier* or sets the iterator to the `end()` iterator if there are no non-empty tiers.
  - Calling `TierList::Iterator::operator++` on an iterator pointing at the last item in a tier list must produce an iterator that compares equal to the tier list's `end()` iterator.
  - If `TierList::iterator::operator++` lands you on an empty tier, you need to return an iterator pointing to the first item in the *next non-empty tier* following that (or an `end()` iterator if there are no remaining items i.e. no non-empty tiers after this point).
  - If an operation `>> n`, where $n > 0$, lands you on an empty tier, you should return an iterator pointing to the first item in the next non-empty tier following that (or an `end()` iterator if there are no remaining items). Similarly, if an operation `<< n` lands you on an empty tier, return an iterator pointing to the first item in the last non-empty tier *preceding* where `<< n` (or an `end()` iterator if no such tier can be found).
  - It is undefined behaviour to use an iterator after a tier list has been modified.
- One of the main challenges in this problem will be figuring out what to store in your iterator class for the tier list. Remember that an iterator functions primarily as an indication of location, so think about what information you would need to store in order to unambiguously identify a position in the tier list (remember that the tier list is based off of the `List` class, which already provides an iterator abstraction into an individual list; you may be able to take advantage of this functionality within your own implementation).
- Another challenge you are likely to face is how to represent the "end" iterator for a tier list (or perhaps even a "begin" iterator, in the case that the tier list is empty). There are a number of ways you can do it, and we leave it up to your ingenuity. One approach is to have a notion of a "dummy" iterator that can be used as a sentinel. As a potential help, if you find yourself needing a list iterator, but have no list to produce an iterator from, you can always use an expression like `List{}.begin()` or `List{}.end()` to produce a "dummy" list iterator. There exist solutions to this problem that use this technique; there also exist solutions that do not. The design is up to you!
- A nested class can access private fields/methods of the wrapping class if either they are `static`, or it has a pointer/reference/object of the wrapping class through which it can access the field/method.

a) **Due on Due Date 1:** Design the test suite `suiteq3.txt` for this program and zip the suite and all needed `.in` and `.out` files into `a3q3a.zip`.

b) **Due on Due Date 2:** Implement this in C++ and place the files `Makefile`, `a3q3.cc`, `tierlist.cc`, `tierlist-impl.cc`, `list.cc` and `list-impl.cc` in the zip file, `a3q3b.zip`. The `Makefile` should create an executable named `a3q3` when the command `make` is given.