Assignment #3: C++ Classes

Due Date 1: Friday, March 1, 2024, 5:00 pm EST (30% marks)
Due Date 2: Friday, March 8, 2024, 5:00 pm EST (70% marks)

Learning objectives:

- Practice with the "spaceship" operator
- C++ classes, constructors, destructors, and operations
- Encapsulation through `class` keyword
- Iterator design pattern

- **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,
- 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.
- There will be a hand-marking component in this assignment. In addition to the usual style requirements, you will be required to implement a design pattern. Your solution must follow the necessary pattern form.
- You may `import`/`include` the following C++ libraries (and no others!) for the current assignment: `compare`, `utility`, `iostream`, `fstream`, `sstream`, and `string`. Marmoset will be setup to **reject** submissions that use C-style I/O or MM, 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

(20% of DD1; 20% of DD2) For this question: use C++20 imports. Compile the system headers with `g++20h` and compile your program with `g++20m`.

You have been given some starter code in `address.cc` and `address-impl.cc` that defines a `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, `a3q1.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 `suiteq1.txt` and zip the suite along with all needed `.in` and `.out` files into `a3q1a.zip`).

b) **Due on Due Date 2:** Implement this program in C++. Submit your `address.cc`, `address-impl.cc` and `a3q1.cc` files that make up your program in your zip file, `a3q1b.zip`.

Question 2

(40% of DD1; 40% of DD2) For this question: use `#includes`. Compile your program with `g++20i`.

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.h`, along with a sample executable. **You may not change the contents of `tierlist.h`, other than by adding your own private methods, variables, and comments, i.e., the interface must stay exactly the same.**

The test harness `a3q2.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 `suiteq2.txt` for this program and zip the suite and all needed `.in` and `.out` files into `a3q2a.zip`.

b) **Due on Due Date 2:** Implement this in C++ and place the files `Makefile`, `a3q2.cc`, `tierlist.h`, `tierlist.cc`, `list.h` and `list.cc` in the zip file, `a3q2b.zip`. The `Makefile` should create an executable named `a3q2` when the command `make` is given.

Question 3

(40% of DD1;40% of DD2) For this question: use `#includes`. Compile your program with `g++20i`.

This question simulates a simplified digital music player that has an underlying library of digital media (either songs or television shows) and a limited number of media play lists that can be set up from the library contents. In particular, we'll use the Iterator design pattern to traverse the library and the play list objects. (You have been provided with some starter code, a test harness, a sample input file, and a sample executable to help you. Make sure that you read through the test harness carefully to understand what it does.)

The `Song` and `TV` classes are subclasses of the base class `DigitalMedia`. They just contain data, with the appropriate accessors and no mutators. (One simplifying assumption that you may make is that the keys are alphanumeric sequences.) They implement the virtual method `print`, which is used to help the iterators output the object information. Use the provided `util.h` and `util.cc` to simplify your implementations of the associated input operators.

- The `TV` constructor sets the key, title, duration (in seconds), episode, season, and series information. See the provided starter file for the list of the error messages produced to stderr, and under what order and condition they are output.
- `operator<<` for `TV` outputs the object's information as `(key, "series" S<season>E<episode> "title", duration)` where each string that might have whitespace within it (i.e. title, series) is surrounded with double-quotation marks.
- `operator>>` for `TV` reads in information in the following format `key\ntitle\nduration\nseason\nepisode\nseries\n`. See the provided starter file for the list of the error messages produced to stderr, and under what order and condition they are printed.
- The `Song` constructor sets the key, title, duration (in seconds), artist, album, and genre information. See the provided starter file for the list of the error messages produced to stderr, and under what order and condition they are printed.
- `operator<<` for `Song` outputs the object's information as `(key, "title", "album", "artist", duration, "genre")` where each string that might have whitespace within it (i.e. title, album, artist, genre) is surrounded with double-quotation marks.
- `operator>>` for `Song` reads in information in the following format `key\ntitle\nduration\nartist\nalbum\ngenre\n`. See the provided starter file for the list of the error messages produced to stderr, and under what order and condition they are printed.

The `Library` holds the pointers to the objects of the `DigitalMedia` subclasses, `Song` and `TV`. When traversed, the items are output in lexicographical order by their key values i.e. standard `std::string` sort order. The keys must be unique.

- `Library::add` adds the specified object pointer to the library, indexed by its key. It will output to stderr the message `"key KKKK already exists in library"` if the key `KKKK` is a duplicate.
- `Library::find` returns the pointer to the object with the specified key, or `nullptr`.
- `Library::remove` removes an item by first searching for the item with the specified key value, and then removing (including deleting the item) it if it exists. If it doesn't exist, nothing happens and no error messages are produced.
- `operator<<` outputs the string `Library:\n`, followed by each item in the library in key order. Each item is preceded by a tab character, `'\t'`, and terminated by a newline.
- `operator>>` adds to the existing library by reading a sequence of digital media objects. Each set of input starts with either an `'s'` to denote a `Song`, or a `'t'` to denote a `TV` object. The information is then read in according to the specified input format for the `Song` or `TV` classes. (The simplest approach is to use the defined input operators to read them into temporary objects whose contents are over-written, and then use compiler-provided default copy operations.) If the specified type is neither an `'s'` nor a `'t'`, the error message `"invalid media type"` followed by the invalid type is output to stderr.

A `Playlist` holds a collection of `DigitalMedia` pointers to the subclasses. Items are kept in the order in which they are added to the play list. Play lists may share items. `Playlist::remove` removes an item by first searching for the item with the specified address, and then removing it if it exists. (It doesn't delete the item since it is not the owner.) If it doesn't exist, nothing happens and no error messages are produced. A play list knows the sum of the durations for each of its items, and how much time has elapsed when it is being "played". Playing the `Playlist` is simulated by iterating over it. When the `PlaylistIterator` is created, it starts at the first item in the list, and the amount of elapsed time is 0 seconds. When the iterator is moved via `operator++`, the duration of the current item is added to the elapsed time before the iterator is moved to the subsequent item. When the end of the sequence is reached, the iterator stops and the elapsed time equals the sum of all of the item durations. The amount of elapsed time can be reset through `Playlist::reset`, though a new iterator will have to be created to start at the beginning. Any change to the play list structure invalidates the iterator. (See the provided test harness for a description of the commands.)

The test harness is not intended to be particularly robust, so do not write test cases for it.

a) **Due on Due Date 1:** Design a test suite for this program. Submit a file called `a3q3a.zip` that contains the test suite you designed, called `suiteq3.txt`, and all of the `.in`, `.txt`, `.err`, `.ret` and `.out` files.

b) **Due on Due Date 2**: Write the program in C++. Save your solution files and `Makefile` in a ZIP file named `a3q3b.zip`. The `Makefile` should create an executable named `a3q3` when the command `make` is given.