Assignment 6: Move Over, Oracle!

This assignment was fully developed by Julie Zelenski, and adapted by Jerry to use the STL.

We've finally covered enough C++ for you to tackle more interesting and challenging problems with solutions that truly capitalize on the unique and powerful features of C++. Your next assignment will be to implement a simple "flat-file" database. A flat-file database is one in which each database record contains a set of fields and values, but, unlike relational databases, there is no facility for records to contain pointers to other records. This assignment will give you experience implementing and using C++ template functions and classes.

The assignment is presented as two tasks. The first task will be to write a `Record` class using the STL `vector` (or `set` or `list` or whatever you want, to be honest). Next you will move on to write the `Database` object that uses the `Record` as part of its implementation. Both of the these objects are templatized in primarily the same manner that the STL containers are.

Due Friday, December 8 at 11:59 p.m.

Task 1: The `Record` class

The `Record` is a class which is specific to databases. A database is a set of records. For example, for a database of people, each record will store the information for one person, and for a database of movies, each record will store the information about one movie. A traditional database record is somewhat like a C `struct`. If you were actually using a `struct`, it would need to be described and specified at "compile time" and all records would have to use the same rigid format.

The traditional definition of record is pretty limiting, so we're going to use something simpler but more flexible. Each record will be a collection of "attribute-value pairs." Each attribute-value pair has exactly two parts: the "attribute" gives the property which is being stored and the "value" stores the current value of that property. Here's what a record definition for Ming in a people database might look like:

```
{
   name = Ming Gu
   e-mail = ming.gu@cs.stanford.edu
   home page = http://www-cs-students.stanford.edu/~minggu
   occupation = CS193D TA
   favorite sport = cs193d@cs question answering
   office hours = Mon 6-8 pm
   office hours = Tues 6-8 pm
}
```

The nice thing about this representation is that it is quite easy to add information. We could add that Hiroshi hates anchovies on his pizza by adding the following pair, even if we didn't add such dislikes for any of the other TAs:

```
dislikes = anchovies on pizza
```

Records in the database can have different attributes depending on what needs to be stored on a per-record basis. The record does not constrain the pairs to be any particular order, and there can be multiple pairs for any given attribute. For example, there are two "office hours" pairs in Hiroshi's record and we could add many more of his dislikes with additional "dislikes" entries.

The interesting twist in the `Record` class concerns the types of the attributes and values. The attributes names are always constrained to be of type `string`. In the Hiroshi example above, the values are also of type `string`, but that may not be appropriate for all our record needs. For example, consider this record from a stock database where the records consists of stock prices on a particular day:

```
{
   Alcatel = 66+3/8
   Apple = 48
   Lilly Eli = 108+3/8
   Oracle = 77+7/16
}
```

In this record, the values are of `Fraction` type. This means the record must be written as a template class to accommodate different types for the values. Note that all pairs in one record must have values of the same type (and all `Record`s in a `Database` must be of the same type), but we will be able to create different databases to hold different kinds of information based on the type of the value in the pairs.

An attribute-value pair can be implemented as an STL `pair` with a `string` and an associated `Value`. The set of pairs within a record should be stored using a `vector` to make it easy to add and access them in a safe and efficient manner.

The `Record` class should be fairly straightforward to write, since much of hard work is in the `vector` already. In addition to the list of pairs, it also has a `selected` bit which can be turned on and off. This will be useful later when we build a database of record and have operations which selected and deselect various records.

Here is a sketch of the methods needed in your `Record` class:

- `Constructor, destructor`

The default (zero-arg) `Record` constructor should initialize a record to contain zero pairs and be unselected. This should be a lightweight operation. The destructor should deallocate any memory allocated on behalf of the record.

- ```
  bool isSelected() const;
  void setSelected(bool val);
  ```

These two methods set and retrieve the selected status of a record.

- `operator<<`

For streaming out the record. The format of a record begins with a opening brace on a line by itself, followed by the attribute-value pairs, one per line, in the form <attribute> = <value>. Each pair is indented a few spaces for visual clarity. The end of the record is marked by a closing brace on a line by itself. See the snippets in the handout and the sample data files for more examples. Being able to stream out the record requires that the value type have its own overloaded version of `operator<<`.

- `operator>>`

For reading record in from string. Expects to read record from stream in the same format written by the `operator<<` above. Since fussing with input seems to be giving some of you lots of trouble, the starting project contains a sketch of the code you can use to handle record reading. Being able to stream in the record requires that the value type have its own overloaded version of `operator>>`.

- ```
  bool matchesQuery(const string& attr, DBQueryOperator op,
                    const Value& want);
  ```

The most interesting method in the `Record` class concerns its ability to determine if a matches a particular query. A query consists of a attribute name, a value for that attribute, and a `DBQueryOperator` (one of `Equal`, `NotEqual`, `LessThan`, or `GreaterThan`). The record searches its pairs to find that attribute name and checks whether the associated value satisfies the query. For example, if the search was to find `"school" NotEqual "Stanford"` on a database of `strings`, the record would return `true` if it had a pair with a `school` attribute that didn't have value `Stanford`. A query of `"age" GreaterThan 15` on a database of integers would return `true` if the record had an `"age"` attribute with a value of `16` or more. One special feature is that an attribute of `"*"` matches any pair, so a search for `"*" Equal "California"` would match if any of the pairs had value `"California"`.

The values are assumed to implement reasonable semantics for the operators `==`, `!=`, `<`, and `>` so directly map the `DBQueryOperator` to the usual relational operators. You might notice there is no interface to directly set or retrieve the value for an attribute or manually add new pairs. Although a more complete implementation of the `Record` class

would likely include this functionality, it turns out you don't need it for our simple database. The only way records get values is by reading the data from a stream. So don't worry about making the most full-featured whizzy record, the above methods are pretty much all you need.

Task 2: The `Database` class

The `Database` class is represented as a unbounded collection of `Records`, some of which are selected and some are not. `Database` objects respond to a few messages which are concerned with manipulating the set of the records and the selection.

The collection of records should be managed using the ever-so-handy STL `vector` template. Like the `Record` class, the `Database` will also be a template class, parametrized by the `Value` type stored in the attribute-value pairs of the record.

- `Constructor, destructor`

The default (zero-arg) `Database` constructor initializes a database to contain zero records. This should be a lightweight operation. The destructor should clean up any memory allocated on behalf of the database.

- ```
  int numRecords() const;
  int numSelected() const;
  ```

Return the total number of records and the number of selected records. Both of these should run in constant time (iterating over all the records to count is too slow).

- `void write(ostream& out, DBScope scope) const;`

Write the entire contents of the database to the given `ostream`. The `DBScope enum` can be either `AllRecords` or `SelectedRecords` to indicate whether to write just the selected records or all records. The database output format is just writing out all the records one after another. See the `Record write` method for information about what the format of each record should look like.

- `void read(istream& in);`

This method deletes all records currently the database and reads a new set from the given `istream`. The database reads records in succession until the stream is exhausted. See the `Record` read method for information about what the format of each record looks like.
  ```

- `void deleteRecords(DBScope scope);`

Permanently removes records from the database. The `DBScope enum` can be either `AllRecords` or `SelectedRecords` to indicate whether to delete just the selected records or delete the entire contents of the database.

- ```
  void selectAll();
  void deselectAll();
  ```

These two methods change the selection to include all the records or none of the records.

- ```
  void select(DBSelectOperation selOp, const string& attr,
              DBQueryOperator op, const value& val);
  ```

A message to select some of the records in the database. The `select` message takes a attribute name, a value, and a `DBQueryOperator` and will search for those records that match the query. The work to determine if a record passes is handled in the `matchQuery` method of `Record`. The `DBSelectOperation enum` controls how records that match are treated. If the operation is `Add`, any record that matches the query is added to the current selection. If the operation is `Remove`, all records that are currently in the selection that match the query will be deselected. If the operation is `Refine`, only records in the selection that meet the new criteria will remain selected, the rest will be deselected.

Task 3: Tying it together

Once you have your `Database` and `Record` classes working, you're ready to try them out in the interactive database program. We have given you a little command-line interface that allows you enter database commands (`read`, `write`, `select`, etc.) and see their effects on the database.

The program will allow you to work with databases of type integer, `string`, or `Fraction`, and can easily be extended to other types as well. I think you will find this type of testbed much more helpful than a file of fixed C++ code, since this allows you to try any combination of operations on the database in order to fully exercise it.

The full text of the program code is attached at the end of this handout. Since it has some template functions and is a client of your template database, you may find it helpful as example code.

Miscellaneous notes

- All the design guidelines we gave you for the first five assignments still apply (compiles cleanly, is `const`-correct, properly deallocates memory, etc.).
- Getting the various template syntax right can be tough, especially given that even the g++ compiler has some quirks even when it comes to processing correct syntax. Keep your eye on your email, in case we can give you some

wisdom about known bugs in the compilers and advice for errors that we see students having trouble with.

Getting started

In our leland class directory `/usr/class/cs193d/assignments` you'll find the project directory `hw6` which contains the testing code we give you, skeleton files for the rest, and a `Makefile` to build the project. You also will need to bring over your `Fraction` class from `hw5`.

No matter where you do your development work, when done, be sure your project will compile and run on the leland workstations. All assignments will be electronically submitted there and that is where we will compile and test your project. I strongly suggest you do your development on Unix if at all possible, since the compilers are inconsistent when it comes to compiling templates, and you're best dealing with those quirks of gcc only and not a second one.