Generic Databases

COMP6771 14s2 Assignment 3 Specification

Version 1.0, Tue 16 Sep 2014 13:24:44 EST

Worth: 8 Marks **Due: 11:59pm Friday 10 October, 2014**

This deliverable was originally developed by Julie Zelenski of Stanford University, later adapted by Jerry Cain of Stanford University to use the STL and further modified by Jingling Xue.

1 Introduction

We've finally covered enough C++ for you to tackle more interesting and challenging problems with solutions that truly capitalise on the unique and powerful features of C++. In this deliverable you will implement a simple "flat-file" database. A flat-file database is one in which each 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 deliverable will give you experience implementing and using C++ function and class templates.

The deliverable is presented as two tasks. The first task will be to write a `Record` class using the STL `vector` (or another suitable STL container). Next you will move on to write the `Database` class that uses `Record` as part of its implementation. Both of these objects are templatised in primarily the same manner as the STL containers.

2 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 for 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 that is being stored and the *value* stores the current value of that property. Here's what a record definition for Mr. Burns in a people database might look like:

```
{
  name = Charles Montgomery Burns
  e-mail = burns@snpp.com
  home page = http://www.snpp.com
  occupation = director
  address = 1000 Mammon Street, Springfield Heights
  age = 126 years
}
```

The nice thing about this representation is that it is quite easy to add information. We could add some of Mr. Burns' favourite words by including the following pairs, even if we didn't include such entries for any other people:

```
  favourite word = velocitator
  favourite word = deceleratrix
```

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 in any particular order, and there can be multiple pairs for any given attribute. For example, there are two `favourite word` pairs in Mr. Burns' record and we could add many more of his favourite words with additional `favourite word` entries.

The interesting twist in the `Record` class concerns the types of the attributes and values. The attributes are always constrained to be of type `string`. In the 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 record 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 records 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 suitable container 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 the functionality is supplied by the container. In addition to the list of pairs, it also has a selected flag that can be turned on and off. This will be useful later when we build a database of records and have operations that select and deselect various records.

Here is an outline of the members needed in your `Record` class:

- *constructor, destructor*

  The default (no arg) `Record` constructor should initialise 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 *two* spaces for visual clarity. The end of the record is marked by a closing brace on a line by itself. You must match the output of the sample binary exactly. Being able to stream out the record requires that the value type has its own overloaded version of `operator<<`.

- `operator>>`

  For reading records in from a stream. Expects to read records from a stream in the same format written by the `operator<<` above, up to whitespace. Being able to stream in the record requires that the value type has its own overloaded version of `operator>>`.

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

  The most interesting operation in the `Record` class concerns its ability to determine if it matches a particular query. A query consists of an 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` "UNSW" on a database of `string`s, the record would return `true` if it had a pair with a "school" attribute that didn't have the value "UNSW". A query of "age" `GreaterThan` 15 on a database of `int`s 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` "Sydney" would match if any of the pairs had value "Sydney".

  The values are assumed to implement reasonable semantics for the operators `==`, `!=`, `<`, and `>` so that they 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 a fancy, full-featured record, the above methods are pretty much all you need.

3 Task 2: The `Database` class

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

The collection of records should be managed using a suitable STL container. Like the `Record` class, `Database` will also be a template class, parameterised by the `Value` type stored in the attribute-value pairs of its records.

Here is an outline of the members needed in your `Database` class:

- *constructor, destructor*

  The default (no arg) `Database` constructor initialises 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, respectively. Both of these should run in constant time (iterating over all the records and counting is too slow).

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

  Write the entire contents of the database to the given stream. The `DBScope enum` can be either `AllRecords` or `SelectedRecords` to indicate whether to write all records or just selected records. The database output format is just writing out all the records one after another. See the `Record` write operation for information about the format of individual records.

- `void read(istream& in);`

  This operation deletes all records currently the database and reads a new set from the given stream. The database reads records in succession until the stream is exhausted. See the `Record` read operation for information about the format of individual records.

- `void deleteRecords(DBScope scope);`

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

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

  These two methods change the selection to include all the records or no records, respectively.

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

  An operation to select some of the records in the database. The `select` operation takes an attribute name, a value, and a `DBQueryOperator` and will search for those records that match the query. The matching work should be handled by the `matchQuery` operation of `Record`. The `DBSelectOperation enum` controls how records that match are treated. If the operation is `Add`, all unselected records that match the query are added to the current selection. If the operation is `Remove`, all selected records that match the query are removed from the current selection. If the operation is `Refine`, only records in the selection that meet the new criteria will remain selected, the rest will be removed from the selection.

4 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 to 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 `int`, `string`, or `Fraction`, and can easily be extended to other types as well. You should find this type of testbed much more helpful than a file of fixed C++ code, since it allows you to try any combination of operations on the database in order to fully exercise it.

5 Getting Started

You are provided with a stub that contains all the files you will need to complete this deliverable, together with a `Makefile`. The stub is available on the subject account. Login to CSE and then type something like this:

```
% mkdir -p cs6771/db
% cp ~cs6771/soft/14s2/db.zip .
% unzip db.zip
% ls
% more README
% make
```

You will need to do some work before you have a code skeleton that compiles without errors or warnings. Spend some time familiarising yourself with the organisation of the program and the content of the files.

The full source of the interactive interface is provided. Since it has some template functions and is a client of your template database, you may find it helpful as example code.

A binary sample solution can be found at: `~cs6771/soft/14s2/db`.

6 Some Advice

Start! You'll soon learn that the amount of code you need to write is quite small.

You also need to heed the advice that you always ignore. Compile and test often. Don't try to write everything first and compile afterwards. Instead (and this applies to any developer, young or old) you should contrive lots of little milestones that sit along the path between what's given to you and the final product. Work toward that final product by slowly evolving your code into something incrementally closer to the place you want to be.

You never want to stray too far from a working system. The safest thing to do is to perturb a working system in the direction of your goal, but making sure the perturbation is small enough that it's easily reversed if things go wrong.

All the guidelines we gave you in class and for the previous deliverables still apply (compiles cleanly, is `const`-correct, properly deallocates memory, etc.).

Getting the template syntax right can be tough, especially given that even GCC has some quirks when it comes to processing correct syntax. Keep your eye on the forums, in case we can give you some wisdom about known bugs in the compiler and advice for errors that we see students having trouble with.

7 Testing

Your code will be compiled using the `Makefile` provided and with these compiler flags:

```
  g++ -std=c++11 -Wall -Werror -O2 ...
```

The testing code, `interactive.cpp`, will be used for marking this deliverable.

No matter where you do your development work, when done, be sure your deliverable works on the school machines. Try to do your development under `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.

Please ensure that your output, including the order of attributes in a record and the order of records in a database, exactly matches that of the sample binary. There is no logical constraint on the ordering, but unfortunately we have to be able to mark your submissions. Note that this pragmatic requirement is likely to limit your STL container choices.

8 Marking

This deliverable is worth **8%** of your final mark.

Your submission will be given a mark out of 100 with a 80/100 automarked component for output correctness and a 20/100 manually marked component for code style and quality.

As this is a third-year course we expect that your code will be well formatted, documented and structured. We also expect that you will use standard formatting and naming conventions. However, the style marks will be awarded for writing C++ code rather than C code.

9 Submission

Copy your code to your CSE account and make sure it compiles without any errors or warnings. Then run your test cases. If all is well then submit using the command (from within your `db` directory):

```
% give cs6771 db record.h record.tem database.h database.tem
```

Note that you are to submit specific files only, this means that these are the only files you should modify and also that you should ensure they work within the supplied infrastructure.

If you submit and later decide to submit an even better version, go ahead and submit a second (or third, or seventh) time; we'll only mark the last one. Be sure to give yourself more than enough time before the deadline to submit.

Late submissions will be penalised unless you have legitimate reasons to convince the LIC otherwise. Any submission after the due date will attract a reduction of 10% per day to the maximum mark. A day is defined as a 24-hour day and includes weekends and holidays. Precisely, a submission x hours after the due date is considered to be $\lceil x/24 \rceil$ days late. *No submissions will be accepted more than five days after the due date.*

Plagiarism constitutes serious academic misconduct and will not be tolerated. CSE implements its own plagiarism addendum to the UNSW plagiarism policy. You can find it here:
`http://www.cse.unsw.edu.au/~chak/plagiarism/plagiarism-guide.html`.

Further details about lateness and plagiarism can be found in the Course Outline.