CSE 241 Programming Assignment 5 (Double Assignment)

NOTE

The weight-point of this assignment is two times the weight-point of a regular programming assignment.

DUE

May 22th, 2025, 23:55

Description

- This is an individual assignment. Please do not collaborate
- If you think that this document does not clearly describes the assignment, ask questions before its too late.

This assignment is about using C++ STL, exception handling and creating Class templates.

- Your program reads two files:
  - `data.txt`
  - `commands.txt`
- According to content in `data.txt`, the program utilizes necessary STL classes and/or user-created classes for a catalog representation.
- Your program creates a log file(`output.txt`) for certain steps of operations performed on catalog.

`data.txt`

- This file holds information about a catalog. The first line of this file describes the format of the entries in the catalog. Other lines of the file are the entries of the catalog.

The format of an entry

The format of an entry is a single line ot text in the following form:

```
<field_name>:<field_type>:<is_array_or_not>|<field_name>:<field_type>:<is_array_or_not>|
... |<field_name>:<field_type>:<is_array_or_not>
```

Entry format descriptor line is a list of field format descriptors separated by the character `|`

A field format descriptor has the following components:

- field_name: A string which labels the field.
- field_type: A string which describes the data type of the field
  - There are 4 different types `string`, `integer`, `double` and `bool`.
- is_array_or_not: A string which can take two distinct values `single` or `multi`
  - `single`: the data stored in the field is a single value
  - `multi`: the data stored in the field is a list of values separated by the char `:`

Example 1

```
  title:string:single|author:string:multi|year:integer:single|tag:string:multi
  Hilbert Spaces With Applications|Lokenath Debnathl:Piotr Mikusinski|2005|Mathematics:Set Theory
  The Neolithic Revolution in the Near East Transforming the Human Landscape|Alan H. Simmons|2011|Social Science:Anthropology:Cultural:General:Archaeology
  Learning Flask Framework|Matt Copperwaite:Charles Leifer|2015|Computers:Programming:Languages:Python:Internet:Application Development:Web:Web Programming
  Graphics Gems V|Alan W. Paeth|1995|
```

In this example there are 4 entries. In each entry there are 4 fields named `title`, `author`, `year` and `tag`. `title` field is a string and there is only one value. `year` is an integer. `tag` is an array of strings etc...

Example 2

```
title:string:single|artist:string:multi|year:integer:single|genre:string:multi
Professor Satchafunkilus and the Musterion of Rock|Joe Satriani|2008|Guitar Virtuoso
Physical Graffiti|Led Zeppelin|1975|Rock
Witchdoctor's Son|Okay Temiz:Johnny Dyani|2017|Jazz:Fusion
Return Of The Mother Head's Family Reunion|Richie Kotzen|2007|Rock:Guitar Virtuoso
```

In this example there are 4 entries. In each entry there are 4 fields named `title`, `artist`, `year` and `genre`. `title` field is a string and there is only one value. `year` is an integer. `genre` is an array of strings etc...

Example 3

```
a:string:single|b:string:multi|c:integer:single|d:string:multi|e:double:multi
a1|b1_1:b1_2:b1_3:b1_4|1|d1_1:d1_2:d1_3|1.0
a2|b2_1:b2_2|2|d2_1:d2_2:|1.1:3.4:5.55:25.0
```

In this example there are 2 entries. In each entry there are 5 fields named `a`, `b`, `c`, `d` and `e`. `a` field is a string and there is only one value. `c` is an integer. `e` is an array of doubles etc...

`commands.txt`

This file includes several commands which work on the catalog information you read from `data.txt`. Each line is a command. The following should be recognized:

- There are two commands.

```
search <value> in <field_name>
sort <field_name>
```

`search` command

- Format:

```
search <value> in <field_name>
```

- Output:

This command returns a list of matched(partially or fully) entries (one entry in a line). Search should be limited to the field specified. Not found returns no line.

- Example:

```
search "Joe" in "artists"
```

This returns the following line:

```
Professor Satchafunkilus and the Musterion of Rock|Joe Satriani|2008|Guitar Virtuoso
```

`sort` command

- Format:

```
sort <field_name>
```

- Output:

This command returns a list of sorted entries (ascending order)

- Example:

```
sort "title"
```

This returns the following lines:

```
Physical Graffiti|Led Zeppelin|1975|Rock
Professor Satchafunkilus and the Musterion of Rock|Joe Satriani|2008|Guitar Virtuoso
Return Of The Mother Head's Family Reunion|Richie Kotzen|2007|Rock:Guitar Virtuoso
Witchdoctor's Son|Okay Temiz:Johnny Dyani|2017|Jazz:Fusion
```

`output.txt`

This file keeps the log of the operations. The following events should be recorded in the specified format:

- `catalog read`
- `output of commands`

`catalog read`

- First list the field names in a line
- At the end, state the number of unique entries.

```
title|artist|year|genre
4 unique entries
```

`output of commands`

- State the command.
- Append its output.

```
search "Joe" in "artist"
Professor Satchafunkilus and the Musterion of Rock|Joe Satriani|2008|Guitar Virtuoso
```

Exceptions

- Your program should catch certain exceptions and create log entries for them.
- You need to catch the following exceptions:

Missing field exception

- If any of the field in any entries is missing your program should omit that line and create an exception record in the log file.

```
Exception: missing field
```

Duplicate entry exception

- If the first field of any entries fully match, your program should create an exception for each repetition and log these instances in the log file.

```
Exception: duplicate entry
```

Wrong command exception

- If the command is not in the expected format(unrecognized field name, extra information etc. . .), log this instance as an exception.

```
Exception: command is wrong
```

A full example.

- Suppose we are given the following `data.txt` file and `commands.txt` file:
- `data.txt`

```
title:string:single|director:string:single|year:integer:single|genre:string:multi|artist:string:multi
The Lord of the Rings: The Fellowship of the Ring|Peter Jackson|2001|Adventure:Drama:Fantasy|Elijah Wood:Ian McKellen:Orlando Bloom
Twelve Monkeys|Terry Gilliam|1995|Mystery:Sci-Fi:Thriller|Bruce Willis:Madeleine Stowe:Brad Pitt
Twelve Monkeys|Terry Gilliam|1995|Mystery:Sci-Fi:Thriller|Bruce Willis:Madeleine Stowe:Brad Pitt
Perfume: The Story of a Murderer|Tom Tykwer|2006|Crime:Drama:Fantasy|Ben Whishaw:Dustin Hoffman:Alan Rickman
Twelve Monkeys|Terry Gilliam|1995|Mystery:Sci-Fi:Thriller|Bruce Willis:Madeleine Stowe:Brad Pitt
Cold Mountain|Anthony Minghella|2003|Adventure:Drama:History
```

- `commands.txt`

```
search "Monkeys" in "title"
search "Space" in "type"
sort "year"
```

- Following is the log file for this example:
- `output.txt`

```
title|director|year|genre|artist
Exception: duplicate entry
Twelve Monkeys|||Sci-Fi:Thriller|Bruce Willis:Madeleine Stowe:Brad Pitt
Exception: duplicate entry
Twelve Monkeys|||Sci-Fi:Thriller|Bruce Willis:Madeleine Stowe:Brad Pitt
Exception: missing field
Cold Mountain|Anthony Minghella|2003|Adventure:Drama:History
3 unique entries
search "Monkeys" in "title"
Twelve Monkeys|Terry Gilliam|1995|Mystery:Sci-Fi:Thriller|Bruce Willis:Madeleine Stowe:Brad Pitt
Exception: command is wrong
search "Space" in "type"
sort "year"
Twelve Monkeys|Terry Gilliam|1995|Mystery:Sci-Fi:Thriller|Bruce Willis:Madeleine Stowe:Brad Pitt
The Lord of the Rings: The Fellowship of the Ring|Peter Jackson|2001|Adventure:Drama:Fantasy|Elijah Wood:Ian McKellen:Orlando Bloom
Perfume: The Story of a Murderer|Tom Tykwer|2006|Crime:Drama:Fantasy|Ben Whishaw:Dustin Hoffman:Alan Rickman
```

Remarks

- You should store the data in an appropriate data type. For example, if the data type is stated as `integer`, you should not sore this value as `string`. You should create an object with an integer member data for the storage.
- Be careful with the order of exceptions. If an entry has a missing field and it has the same first field with another entry, you should throw missing field exception.
- Assume no other errors will be present in the files.
- Try to generalize your program. (you can use templates).
- Efficiency is important. (try to use the existing (STL) containers and their methods for sorting etc. . .)
- **You should generate a representation of an entry. Repeatedly parsing the same data is not allowed.**

Turn in:

- Source code of a complete C++ program and a suitable makefile. You should use c++11 standard. Your code will be tested in a linux-gcc environment.
- A script will be used in order to check the correctness of your results. So, be careful not to violate the expected output format.
- Provide comments unless you are not interested in partial credit. (If I cannot easily understand your design, you may loose points.)

- You cannot get full credit if your implementation contradicts with the statements in this document.

Sample Code

Here is a sample code you can use. You can also come up with a different design. You don't have to use this as the basis.

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

enum class fieldT{single, multi};
enum class dataT{S,I,D,B}; //string, int, double, bool


//fields in the entries share the same info.
// We don't want to repeatedly store the same data.
struct fieldformat
{
        string name;
        dataT type;
        fieldT sm;
};

//This class does not need to hold any data
class FieldBase
{
    public:
        virtual fieldT gettype() const = 0;
};


// this class will hold data of a field.
// You can store a reference to a particular fieldformat object
// or keep track of the indices of fields and fieldformat objects.
// Does not store format info. The format of a field is sored in a single fieldformat object.
template<class T>
class FieldSingle : public FieldBase
{
    public:

        FieldSingle(T data)
        {
            this->data = data;
        }
        T getdata() const {return this->data;}
        fieldT gettype() const {return  fieldT::single;}
```

```
    private:
        T data;
};

//this will hold multiple data
template<class T>
class FieldMulti : public FieldBase
{
    public:
    FieldMulti(vector<T> data)
    {
        this->data = data;
    }
    T getdata(int index) const {return this->data[index];}
    int getsize() const { return this->data.size();}
    fieldT gettype() const {return  fieldT::multi;}
    private:
        vector<T> data;
};


//You should also define a class about fields in an entry.
class entry
{
    //Normally, this should be private
    public:
        vector<FieldBase*> fields; //We don' store the objects here. Just pointers.
        //You cannot store objects here. Why?
};


class databaseorganizer
{
    public:
        void parsedata() { parseformat(); parseentries();}
    private:
        void parseformat(); // reads the first line
        void parseentries(); // reads entries
        vector<fieldformat> entry_format;
        vector<entry> entries;
};


//A comparator to compare two entries
struct mycomparator {
    mycomparator(fieldformat ff, int fieldindex)
    {
        this->ff = ff;
        this->fieldindex = fieldindex;
    }
    int fieldindex;
    fieldformat ff;
    bool operator() (entry e1, entry e2) {
        if(ff.sm==fieldT::single)
        {
            //compare depending on the data type
```

```
            if(ff.type==dataT::S)
            {
                FieldSingle<string>* fs1 = (FieldSingle<string>*) e1.fields[fieldindex];
                FieldSingle<string>* fs2 = (FieldSingle<string>*) e2.fields[fieldindex];
                return (fs1->getdata() < fs2->getdata());

            }
        }
        if(ff.sm==fieldT::multi)
        {
            if(ff.type==dataT::S)
            {
                FieldMulti<string>* fs1 = (FieldMulti<string>*) e1.fields[fieldindex];
                FieldMulti<string>* fs2 = (FieldMulti<string>*) e2.fields[fieldindex];
                return (fs1->getdata(0) < fs2->getdata(0));

            }
        }
    }
};


//The entry class is not used in the following example. But, you should use it.
int main()
{
    //normally, you shuld start with parsing fieldformats in a databaseorganizer object.
    //here we don't do that.
    //let's assume that we parsed the field formats. Then, we will get the following vector.
    vector<fieldformat> entry_format;
    //lets put some data in it.
    entry_format.push_back({"artist",dataT::S,fieldT::single});
    entry_format.push_back({"albums",dataT::S,fieldT::multi});
    //now we defined formats of two fields in a sequence.
    //we can parse many entries which share the same format.

    //Normally, you should parse them and dynamically create objects.
    //all of the entries will be stored in a vector of entries. (You can also choose it to be a
    ↪  set)
    vector<entry> entries;
    //lets manually define one entry
    entry e1;
    //lets populate the entry
    //we need fields. every entry will have two fields.
    //lets create the first field
    //The first field is single.
    if(entry_format[0].type==dataT::S) //Template speciization cannot be created dynamically.
    {

        FieldSingle<string>* e1f1 = new FieldSingle<string>(string("Tarkan"));
        e1.fields.push_back(e1f1);
    }

    //This is the second field. This one is multi.
    if(entry_format[1].type==dataT::S)//Template speciization cannot be created dynamically.
    {
        string raw_data = "first_album:second_album";
```

```
    //call your parser on raw_data and it produces the following vector:
    vector<string> albums;
    //I don't have a parser here. So, I manually add the elements.
    albums.push_back(string("first_album"));
    albums.push_back(string("second_album"));
    FieldMulti<string>* e1f2 = new FieldMulti<string>(albums);
    e1.fields.push_back(e1f2);
}

//Field object address are in the entry now.
//For another entry, you have to repeated the process.
//Lets store this entry in the vector of entries.
entries.push_back(e1);


//Now lets traverse the entries. (we have only one in this example.)
for(int i=0; i<entries.size(); i++)
{
    cout<<"entry"<<i<<endl;
    //lets print the data of every entry.
    //we need to learn the format of the particular field.
    //format vector and the fields in an entry are parallel.
    //we use the same index
    //Loop through fields of an entry
    for(int j=0; j<entry_format.size(); j++)
    {
        cout<<"  field:"<<entry_format[j].name<<endl;
        if(entry_format[j].sm == fieldT::single)
        {
            if(entry_format[j].type == dataT::S)
            {
                //this is a single string data
                //convert the pointer.
                FieldSingle<string>* fs = (FieldSingle<string>*) entries[i].fields[j];
                cout<<"     single data = "<<fs->getdata()<<endl;
            }

        }
        if(entry_format[j].sm == fieldT::multi)
        {
            if(entry_format[j].type == dataT::S)
            {

                //this is a multi string data
                //convert the pointer.
                FieldMulti<string>* fs = (FieldMulti<string>*) entries[i].fields[j];
                for(int k=0; k<fs->getsize(); k++)
                {
                    cout<<"     multi data"<<k<<" = "<<fs->getdata(i)<<endl;
                }

            }

        }

    }
```

```
    }

    //If we wan to sort entries based on a paricular field, we need to create a comparator
    ↪  object
    //lets say, we want to sort based on the second field.
    //Pass the field format and the index of it.
    mycomparator mc(entry_format[1], 1);
    std::sort(entries.begin(), entries.end(), mc);
    //since we have only one entry, you cannot see the effect of this sort.
    //popoulate the entries vector with more data to see them sorted.


}
```

Late Submission

- Not accepted.

Grading (Tentative)

- `Max Grade` : `100`.
- Multiple tests will be performed.

All of the followings are possible deductions from `Max Grade`.

- Do **NOT** use hard-coded values. If you use you will loose `10pts`.
- No submission: `-100`. (be consistent in doing this and your overall grade will converge to `N/A`) (To be specific: if you miss `3` assignments you'll get `N/A`)
- Compile errors: `-100`.
- Irrelevant code: `-100`.
- Major parts are missing: `-100`.
- Unnecessarily long code: `-30`.
- Inefficient implementation: `-20`.
- Using language elements and libraries which are not allowed: `-100`.
- Not caring about the structure and efficiency: `-30`. (avoid using hard-coded values, avoid hard-to-follow expressions, avoid code repetition, avoid unnecessary loops).
- Significant number of compiler warnings: `-10`.
- Not commented enough: `-10`. (Comments are in English. Turkish comments are not accepted).
- Source code encoding is not `UTF-8` and characters are not properly displayed: `-5`. (You can use 'Visual Studio Code', 'Sublime Text', 'Atom' etc... Check the character encoding of your text editor and set it to `UTF-8`).
- Missing or wrong output values: `Fails the test`.
- Output format is wrong: `-30`.
- Infinite loop: `Fails the test`.
- Segmentation fault: `Fails the test`.
- Fails 5 or more random tests: `-100`.
- Fails the test: `deduction up to 20`.
- Prints anything extra: `-30`.
- Unwanted chars and spaces in output: `-30`.
- Submission includes files other than the expected: `-10`.
- Submission does not follow the file naming convention: `-10`.
- Sharing or inheriting code: `-200`.
- Not storing data in an appropriate data dype variable: `-50`
- Sort does not work: `-50`
- Not using templates for the core part: `-200`
- No exception handling: `-100`
- Harcoded definitions about sizes of arrays and containers: `-100`
- Repeatedly parsing the same data: `-200`