CS 3370

Program 3 – String and Stream Processing

Write a program that handles I/O of Employee objects containing the following data attributes:

name (string)
id (int)
address (string)
city (string)
state (string)
country (string)
phone (string)
salary (double)

You will read XML representations of Employee objects from a file, and then will create a file of fixed-length records of Employee data. Do not use an XML software library or regular expressions for this assignment; just use operations from the **std::string** class for parsing input (see the Notes section below). The input files have no XML header tag.

The XML tags are named the same as the attributes (ignoring case, of course). For those that know XML, don't worry about top-level, system XML tags; we just want sequences of same-level Employee tag groups with attributes tags nested to one level only, like the following (indented to show the nesting, but they may appear free-form in the file; don't assume that there will be any newlines or formatting):

```
<Employee>
	<id>12345</id>
	<name>John Doe</name>
	…
	<salary>40000</salary>
</Employee>
<Employee>
	…
</Employee>
```

A single XML text file may have multiple Employee records. Internal Employee field tags can appear in any order or not at all, except that **name** and **id** are required. When creating Employee objects, use 0.0 as a default for *salary* and empty strings for the other optional attributes.

I have provided several XML files to process. All but one of them have errors that you must catch. Throw exceptions of **runtime_error** (defined in <**stdexcept**>) for these cases. You will catch these exceptions in your **main** function.

Your Employee class must contain at least the following:

```
  void display(std::ostream&) const;  // Write a readable Employee representation to a stream
  void write(std::ostream&) const;    // Write a fixed-length record to current file position
  void store(std::iostream&) const;   // Overwrite (or append) record in (to) file
  void toXML(std::ostream&) const;    // Write XML record for Employee
  static Employee* read(std::istream&);         // Read record from current file position
  static Employee* retrieve(std::istream&,int); // Search file for record by id
  static Employee* fromXML(std::istream&);      // Read the XML record from a stream
```

Do not change any signatures or return types. Define any constructors you feel needful. You do not need a destructor, as Employee objects only contain string objects and numbers. **read**, **retrieve**, and **fromXML** return a **nullptr** if they read end of file. **retrieve** also returns a **nullptr** if the requested id is not found in the file. Throw exceptions for data errors.

The following is an overview of what your **main** function should do to test your functions using the file *employee.xml* (provided in this Zip file):

1) Obtain the name of an XML file to read from the command line (argv[1]). Print an error message and halt the program if there is no command-line argument provided, or if the file does not exist.
2) Read each XML record in the file by repeatedly calling Employee::fromXML, which creates an Employee object on-the-fly, and store it in a vector (I recommend using unique_ptrs in the vector).
3) Loop through your vector and print to **cout** the Employee data for each object (using the **display** member function).
4) The next step is to create a new file of fixed-length Employee records. This is explained below. Write the records for each employee to your new file (call it "employee.bin") in the order they appear in your vector. Open this file as an **fstream** object with both read and write capability, and in binary format.
5) Clear your vector in preparation for the next step.
6) Traverse the file by repeated calls to Employee::read, filling your newly emptied vector with Employee pointers for each record read.
7) Loop through your vector and print to **cout** an XML representation of each Employee using Employee::toXML.
8) Search the file for the Employee with id 12345 using Employee::retrieve.
9) Change the salary for the retrieved object to 150000.00
10) Write the modified Employee object back to file using Employee::store
11) Retrieve the object again by id (12345) and print its salary to verify that the file now has the updated salary.
12) Create a new Employee object of your own with a new, unique id, along with other information.
13) Store it in the file using Employee::store.
14) Retrieve the record with Employee::retrieve and display it to **cout**.

Make sure you don't leak any memory.

Here is sample output from employee.xml (except for steps 12–14):

```
$ ./a.out employee.xml
id: 1234
name: John Doe
address: 2230 W. Treeline Dr.
city: Tucson
state: Arizona
country: USA
phone: 520-742-2448
salary: 40000

id: 4321
name: Jane Doe
address:
city:
state:
country:
phone:
salary: 60000

id: 12345
name: Jack Dough
address: 24437 Princeton
city: Dearborn
state: Michigan
country: USA
phone: 303-427-0153
salary: 140000

<Employee>
       <Name>John Doe</Name>
       <ID>1234</ID>
       <Address>2230 W. Treeline Dr.</Address>
       <City>Tucson</City>
       <State>Arizona</State>
       <Country>USA</Country>
       <Phone>520-742-2448</Phone>
       <Salary>40000</Salary>
```

```
</Employee>

<Employee>
       <Name>Jane Doe</Name>
       <ID>4321</ID>
       <Salary>60000</Salary>
</Employee>
<Employee>
       <Name>Jack Dough</Name>
       <ID>12345</ID>
       <Address>24437 Princeton</Address>
       <City>Dearborn</City>
       <State>Michigan</State>
       <Country>USA</Country>
       <Phone>303-427-0153</Phone>
       <Salary>140000</Salary>
</Employee>

Found:
id: 12345
name: Jack Dough
address: 24437 Princeton
city: Dearborn
state: Michigan
country: USA
phone: 303-427-0153
salary: 140000
150000
```

Note that **store** and **retrieve** search the file for the correct record by looking at the ids in each record. **retrieve** fails if no record with the requested id is found; return a **nullptr** in that case. **store** overwrites the record if it exists already; otherwise it appends a new record to the file.

Throw **runtime_error** exceptions with a suitable message if any required XML tags are missing, or if any end tags for existing start tags are missing, or for any other abnormalities.

Notes

Your XML input function should not depend on the line orientation of the input stream, so don't read text a line at a time (i.e., don't use **getline( )** with the newline character as its delimiter [other delimiters are okay]—the input should be "free form", like source code is to a compiler). Do not use any third-party XML libraries. I want you to do your own basic, custom parsing by using simple string operations. An important part of this assignment is also the proper use of exceptions.

To process fixed-length records in a file requires special processing. Our Employee objects use **std**::**string** objects, which are allowed to have strings of any length, but we need to write fixed-length, byte-records to files using **ostream**::**write** (and we read them back into memory with **istream**::**read**). Some strings may therefore get truncated. Here is the record structure I used for transferring Employee data to and from files.

```
    struct EmployeeRec {
        int id;
        char name[31];
        char address[26];
        char city[21];
        char state[21];
        char country[21];
        char phone[21];
        double salary;
    };
```

Note that the strings here are C-style, zero-delimited strings. You need to copy data from and to an EmployeeRec when doing I/O with the binary file. It is the EmployeeRec object that is actually written/read. See employeedb.cpp. You may find some of the following functions useful for this assignment:

```
istream::gcount, istream::seekg, istream::tellg, istream::read, ostream::write,
istream::getline(istream&,string&, char), istream::unget, ios::clear, string::copy,
string::empty, string::stoi, string::stof, string::find_first_not_of, string::find,
string::substr, string::clear, string::c_str.
```

The goal here is to understand strings and streams better, as well as serializing simple object data. Along the way, I ended up creating a few handy XML-related functions for future use, which I named **getNextTag**, **getNextValue**, both of which take an input stream as a parameter. You might find such a practice useful. Remember to do a *case-insensitive* compare when looking for tag names. For case-insensitive string comparisons, it is handy to use the non-standard C function, **strcasecmp**, defined in <**cstring**> as a GNU/clang extension. (This function works like **std**::**strcmp** but ignores case.) In Microsoft Visual C++ the function is named **stricmp** (possibly with a leading underscore). *Note*: to extract a **char*** from a C++ string object to pass to **strcasecmp**/**stricmp**, you need to call **c_str( )** on the string objects:

```
strcasecmp(s1.c_str(), s2.c_str())      // Returns negative | 0 | positive for < | == | >
```

Sample input files are in this zip file. The files *employee2.xml* through *employee8.xml* have errors that you should catch through exceptions and print a meaningful message. Here's my output:

```
$ ./a.out employee2.xml
Missing <Name> tag
$ ./a.out employee3.xml
Missing </City> tag
$ ./a.out employee4.xml
Invalid tag: <Employee>
$ ./a.out employee5.xml
Missing <Employee> tag
$ ./a.out employee6.xml
Multiple <City> tags
$ ./a.out employee7.xml
Invalid tag: <village>
$ ./a.out employee8.xml
Missing <Employee> tag
```

You should get the same output.

Not counting the main driver, my code was 272 lines. FYI.

Assessment Rubric

Competency ↓	**Emerging →**	**Proficient →**	**Exemplary**
Memory Management	Every **new** has a **delete** (no memory leaks); destructor does the right thing.		(Use **unique_ptr** in **main** to receive heap pointers from **fromXML** and **read** so you don't have to explicitly **delete** pointers in the driver.)
Memory efficiency			No unnecessary temporary variables are used in **read** and **fromXML**; use a **std**::**bitset** to track which Employee fields have been encountered when reading an Employee's XML data
File I/O		Use binary files properly; efficient use of file positioning functions; call **clear** when needed	

Clean Code		No repeated code (refactor); No unnecessary code	Simplest possible logic to fulfill program requirements; intelligent use of **std**::**string** functions to parse data (do *not* use regular expressions)
Defensive Programming	Exceptions thrown as requested	Use **reinterpret_cast** with **std::read** and **std::write**; don't hard-code special characters like '<', '>', etc. (use constants).	Use **assert** in appropriate places
Other	Use range-based **for** when applicable	Use constructor initializer lists for non-built-in types of class members; use in-class initializers for non-static data members where applicable; proper use of command-line arguments as assigned	