CS 3370 - Programing Assignment 3

February 17, 2026

A Assignment Overview

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.

Note: When you turn in this assignment, include all of the XML files in your onlinegdb.com link. This is so I can test everything.

If you do the bonus for file handling, you can just have all of the files load and process in a single run. If you do this, you need to print out the name of each file as it is getting processed.

B XML Format Specification

The XML tags are named the same as the attributes (ignoring case, of course). Employee tag groups contain attribute tags nested to one level only, like the following (indented to show the nesting, but they may appear free-form in the file):

```
<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.

C Required Employee Class Methods

Your Employee class must contain at least the following methods:

```
void display(std::ostream&) const;  // Write a readable Employee
 ↪  representation to a stream
void write(std::ostream&) const;    // Write a fixed-length
 ↪  (size) record to current file position
void store(std::iostream&) const;   // Overwrite (or append)
 ↪  record in (to) file (seek to id, if present, overwrite, if
 ↪  not, append)

Employee* read(std::istream&);          // Read record from
 ↪  current file position
Employee* retrieve(std::istream&,int); // Search file for record
 ↪  by id
Employee* fromXML(std::istream&);       // Read the XML record
 ↪  from a stream
```

Do not change any signatures or return types (unless for bonus points below). 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 (main should catch and print the error, then go on to the next XML file)

D XML Parser Hints/Reminders

Writing a string (and stream) based XML parser from scratch is complex. The following pseudocode will help you understand what you need to be parsing and validating, some of these points are reiterated from what was noted above.

Everything below is a suggestion, so if you find another preferred way that works, that's ok as long as the assignment's requirements are met.

1. Remember the following assumptions:
   (a) Do not assume a line of the file is one tag or one set of tags. XML does not use newlines (”) as a delimiter.
   (b) Tags are not case sensitive. Force case before checks. (Preserve case for the actual data, however.)
   (c) Preserve spaces for data between tags, otherwise ignore them. Data may be empty between tags. That's valid.
   (d) An employee may not have all of its tags (fields). Consider the missing ones to be empty fields.
   (e) Tags within the employee may be in any order.
2. You can use a Finite State Machine (FSM) to keep track of what you are processing. If you hit a character that is not expected for the correct next state of the FSM then you know it's invalid. The same is true if you hit EOF when the FSM is saying it's in the middle of a record or tag.
3. You may actually use nested FSMs - one for valid XML parsing, the other checking for valid employee fields. This is not the only way to do this however.
4. If you hit an error in the XML file, you can stop processing that file, just throw.

E Main Function Requirements

Your `main` function should perform the following steps using the file `employee.xml`:

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 (It's recommended, but not necessary to use `unique_ptr`s in the vector). Use vector::push_back to add items. (see https://en.cppreference.com/w/cpp/container/vector/push$_back$
3. Loop through your vector and print to `cout` the Employee data for each object (using the `display` member function).
4. Create a new file of fixed-length Employee records. 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` using the display member.
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 (just call display) to verify that the file now has the updated salary.

Make sure you don't leak any memory.

F Sample Output

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



Found:
id: 12345
name: Jack Dough
```

```
address: 24437 Princeton
city: Dearborn
state: Michigan
country: USA
phone: 303-427-0153
salary: 140000

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

G Error Handling Requirements

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.
Expected error outputs:

```
$ ./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
```

H Implementation Notes

H-1 XML Parsing

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. Use simple string operations for parsing.

H-2 Fixed-Length Records

To process fixed-length records in a file requires special processing. Use the C++ format library for this purpose. 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 read them back with `istream::read`). Below is a struct with the sizes required. You do not need to use the struct, but you can incorporate the size info into your string format calls.

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

H-3 File Operations

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.

H-4 Useful Functions

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
```

For case-insensitive string comparisons, use the non-standard C function `strcasecmp`, defined in `<cstring>` as a GNU/clang extension or force case, up to you.

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

I Assessment Rubric

As always: include a reflection on the assignment including a description of any concerns/difficulties as well as triumphs you had.

Competency	**Emerging**	**Proficient**	**Exemplary**
Memory Management	(no memory leaks); destructor does the right thing		Use `unique_ptr` in main to receive heap pointers from `fromXML` and `read`
File I/O		Use binary files properly; efficient use of file positioning functions; call clear when needed	
Clean Code	DRY code		Simplest possible logic; intelligent use of `std::string` and `format` functions
Defensive Programming	Exceptions thrown as requested	Use `reinterpret_cast` with `std::read/write`; don't hard-code special characters	Use assert in appropriate places

BONUS: C++17 Enhancements

Filesystem Library (5 points)

Modernize the file handling using the C++17 filesystem library:

```cpp
#include <filesystem>
namespace fs = std::filesystem;

// In main:
if (!fs::exists(argv[1])) {
    std::cerr << "Error: File " << argv[1] << " does not
    ↪ exist\n";
    return 1;
}

// Add a method to Employee class:
static void processDirectory(const fs::path& dir) {
    for (const auto& entry : fs::directory_iterator(dir)) {
        if (entry.path().extension() == ".xml") {
            // Process XML file
        }
    }
}
```

std::optional Return Types (5 points)

Replace pointer returns with `std::optional`:

```
#include <optional>

class Employee {
public:
    static std::optional<Employee> read(std::istream&);
    static std::optional<Employee> retrieve(std::istream&, int);
    static std::optional<Employee> fromXML(std::istream&);
};

// Usage:
if (auto emp = Employee::retrieve(file, 12345); emp.has_value())
 ↪  {
    emp->display(std::cout);
}
```

string_view for Parsing (5 points)

Use `std::string_view` to avoid unnecessary string copies:

```cpp
#include <string_view>

bool isValidTag(std::string_view tag) {
    static constexpr std::array validTags = {
        "name"sv, "id"sv, "address"sv, "city"sv,
        "state"sv, "country"sv, "phone"sv, "salary"sv
    };
    return std::find(validTags.begin(), validTags.end(), tag) !=
     ↪  validTags.end();
}
```