SBU CSE 390/590 - Special Topics in Computer Science, Spring 2023:
"C++ Programming for Real Life Challenges"

Demo Project to assist with Assignment 4 - Requirements and Guidelines

This is the requirements document for the demo project that comes with a solution, aimed for assisting in your 4th assignment.

It is totally independent from the previous assignments.

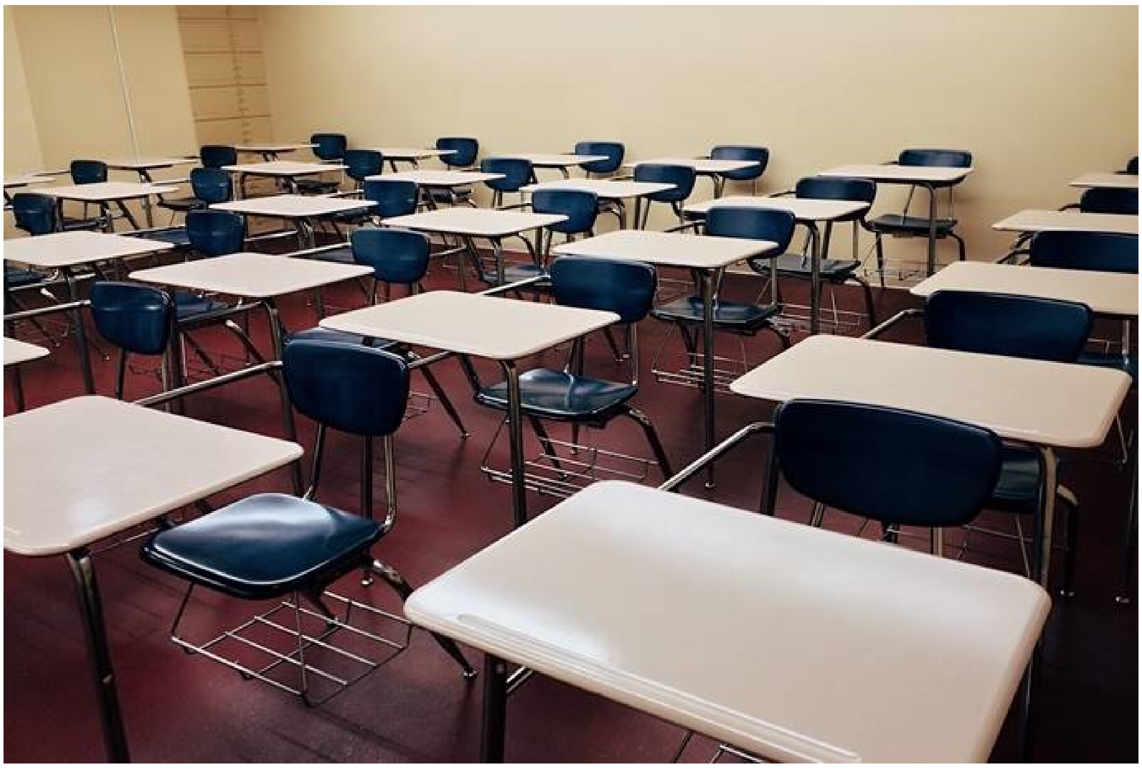
NOTE: this is NOT your 4th assignment! It's a requirements document for a demo project that comes with a solution.

Exam Hall - Requirements and Guidelines

NOTE: it is your responsibility to make your submission of assignment 4 work correctly. The code that would be presented for solving this demo project is provided as is without any guarantee that it is bug-free or even work. In case of any bugs or insufficiencies it is your responsibility to make sure that such bugs or insufficiencies would not appear in the solution of your exercise.

In this exercise you would be required to manage an "ExamHall" class as a container for "Examinees", based on the following requirements:

- An "Examinee" is an unknown type (modeled as a Template parameter)
- The "ExamHall" has dimensions X * Y, allowing sitting of X * Y examinees
- The API described below is strict



<u>ExamHall's API</u>

All classes and types would be inside the namespace: *exams*
Entire implementation shall be inside *ExamHall.h* (i.e., no *.cpp* as this is a template)

Special Global Types:

X, Y - *constructed explicitly by **int*** and has a ***casting to int***

Examinee template parameter for ExamHall:
ExamHall would be a template class, with a template parameter: ***typename Examinee***

type:
```
template<typename Examinee>
using Grouping = std::unordered_map<string, std::function<string(const Examinee&)>>;
```
above type defines a groping map, with:

- name of the grouping, std::string, as the *key*
- a grouping function that gets a const Examinee& and returns its group name as a string, for this grouping function, as the *value*

Groupings can be according to: course name (there might be different exams in the same class) or any other attribute that an examinee is able to provide.

ExamHall's Constructor:
```
ExamHall(X x, Y y, Grouping<Examinee> groupingFunctions) noexcept;
```
Usage Example:
```
ExamHall<int> myHall{X{5}, Y{12}, {}};
```
Above creates a *hall* with *examinees* of type *int* and *no grouping functions*.

ExamHall should be either *copyable* or *moveable*, or *both*.
ExamHall should be either *copy-assignable* or *move-assignable*, or *both*.

sitting an examinee:
```
void sit(X x, Y y, Examinee e) noexcept(false);
```
the method may throw BadPositionException.

un-sitting an examinee:
```
Examinee unsit(X x, Y y) noexcept(false);
```
the method may throw BadPositionException.

moving an examinee from one location to another:
```
void move(X from_x, Y from_y, X to_x, Y to_y) noexcept(false);
```
the method may throw BadPositionException.

iterators begin and end:
The hall would only have a const version of begin and end iterators for iterating over all examinees. There is no defined order. Iteration shall not create a copy of the examinees but rather run on the original.

getExamineesViewByGroup:
The hall would allow a “view” of examinees belonging to a certain group, by specific grouping criteria:

- `getExamineesViewByGroup(const string& groupingName, const string& groupName) const;`

functions would not throw an exception, but may return an empty view.

- Above function would return something of your choice which has iterators *begin* and *end* to allow traversal on the view.
- The view would never be a copy of the Examinees. If the user calls one of these functions and holds the result, then sits, un-sits or moves examinees, then runs on the view - the run on the view would be on the new data. On the other hand, the view **doesn’t have to support** traversing on the view, stopping, then sitting, un-sitting or moving an examinee, then continuing the traversal - such operation is not defined. In other words sit/un-sit/move operations may invalidate a view.
- After a full cycle over the view you cannot traverse over it again, but you can retrieve the same view again by calling *getExamineesViewByGroup* again.
- The order for running on the view is not important
- **iterator* provided by the view would be:
  std::pair<tuple {X, Y}, const Examinee&>

BadPositionException
has the following ctor: `BadPositionException(X x, Y y, string msg);`

- the message is yours, no specific requirements, but try to make it informative

---

Usage Example

```
#include "ExamHall.h"

using namespace exams;
using std::string;

int main() {
   // create simple ExamHall with no groupings
   ExamHall<std::string> examHall{ X{4}, Y{8}, {} };
   // sit a student
   examHall.sit(X{2}, Y{6}, "Danit");

   // create grouping pairs
   Grouping<string> groupingFunctions = {
      { "first_letter",
         [](const string& s){ return string(1, s[0]); }
      },
      { "first_letter_toupper",
         [](const string& s){ return string(1, char(std::toupper(s[0]))); }
      }
   };

   // create ExamHall 2
   ExamHall<std::string> examHall2{ X{5}, Y{12}, groupingFunctions };
```

```
    // sit examinees
    examHall2.sit(X{0}, Y{0}, "John");
    examHall2.sit(X{1}, Y{1}, "deer");
    examHall2.sit(X{1}, Y{2}, "Dove");

    auto view_d = examHall2.getExamineesViewByGroup("first_letter", "d");
    auto view_Dd = examHall2.getExamineesViewByGroup("first_letter_toupper", "D");

    examHall2.sit(X{2}, Y{2}, "david");

    // loop on all examinees - John, deer, Dove, david - in some undefined order
    for(const auto& examinee : examHall2) {
       std::cout << examinee << ' ';
    }
    std::cout << std::endl;

    // loop on view_d:  {X{1}, Y{1}, deer},
    //               {X{2}, Y{2}, david}
    // - in some undefined order
    for(const auto& examinee_data : view_d) {
       const std::tuple<X, Y>& pos = examinee_data.first;
       std::cout << "X{" << std::get<0>(pos) << "}, "
                 << "Y{" << std::get<1>(pos) << "}, "
                 << examinee_data.second << std::endl;
    }
    std::cout << std::endl;

    // loop on view_Dd: {X{1}, Y{1}, deer},
    //               {X{2}, Y{2}, david}
    //               {X{1}, Y{2}, Dove}
    // - in some undefined order
    for(const auto& examinee_data : view_Dd) {
       const std::tuple<X, Y>& pos = examinee_data.first;
       std::cout << "X{" << std::get<0>(pos) << "}, "
                 << "Y{" << std::get<1>(pos) << "}, "
                 << examinee_data.second << std::endl;
    }
    std::cout << std::endl;

} // end of main
```

Good Luck!