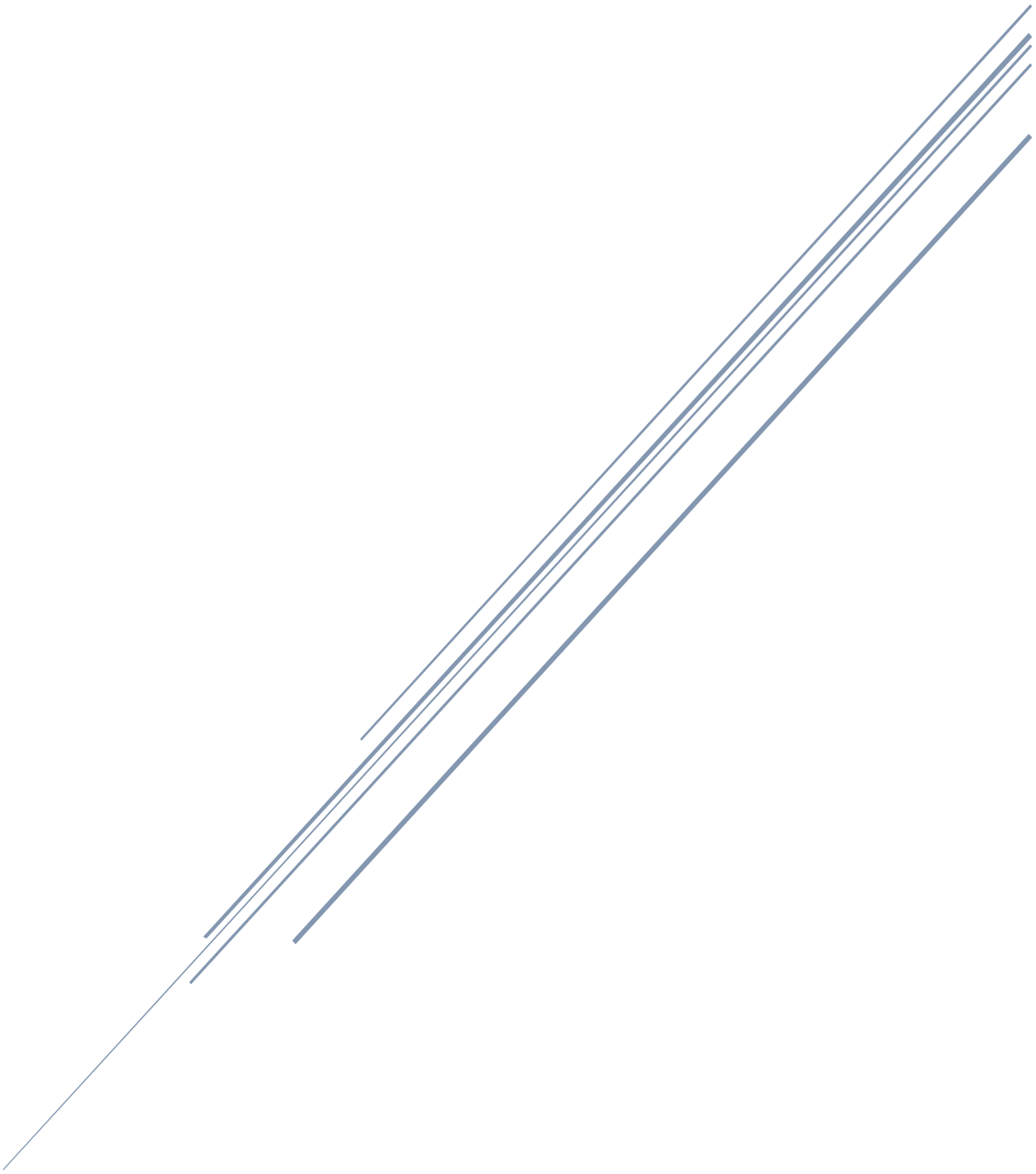
ERROR BY NIGHT

A school management system



The Error by Night Team

March 2021

Table of Contents

Introduction

This is our submission for the MusalaSoft School Project. Our program aims to model the relationships between students, teachers, teams and projects in a school. We have provided the option for multiple schools but each school is self-contained (i.e. one school's students cannot participate in another one's teams).

This document's goal is to provide you with sufficient knowledge to be able to piece together our implementation and avoid some pitfalls you may otherwise experience while operating or modifying the application.

Brief description

In its present state the program allows one to create and manage schools, students, projects and teams. We have also implemented our take on the example query from the task statement – about teachers who do not have any assigned teams for a particular project.

The project is far from complete, however. As it is now the user has to make sure they keep track of their data's logical consistency. This issue is discussed in greater detail in some of the following sections of this document. There is also room for improvement where our secondary features are concerned. Some possible future additions include more sophisticated querying options and enhancements for the user experience.

Diagram of the menus

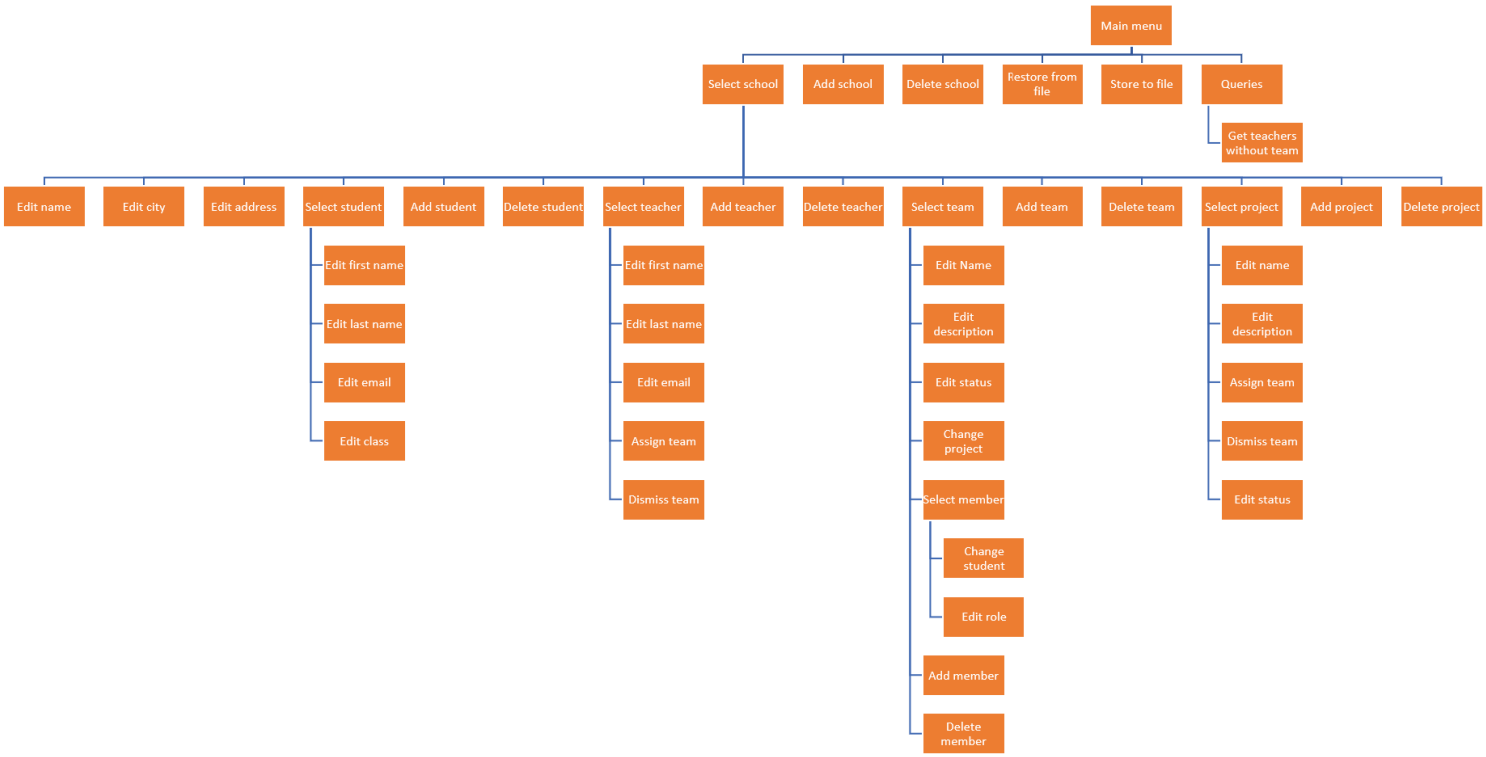


Figure 1 – Menu diagram

Implementation reference

Structures

PROJECT

```
struct PROJECT {
	std::string name;
	std::string description;
	std::vector<size_t> teams;
	STATUS status;

	bool restore(std::istream &file);
	void store(std::ostream &file) const;

};
```

Represents a project. Has a name, description and status. Stores the IDs of all teams assigned to this project.

Please note that the teams vector does not update automatically when the project is assigned to a team (see TEAM).

Member functions

```
bool restore(std::istream &file);
```

Tries to restore the project from the given stream. Returns true if the operation was successful, false otherwise.

```
void store(std::ostream &file) const;
```

Writes the project's information to the given stream.

STUDENT

```
struct STUDENT {
	std::string firstName;
	std::string lastName;
	std::string email;
	unsigned grade;
	char classLetter;

	bool restore(std::istream &file);
	void store(std::ostream &file) const;

};
```

Represents a student. Has a first and last name, an email and a class divided into a grade number and a letter.

Member functions

```
bool restore(std::istream &file);
```

Tries to restore the student from the given stream. Returns true if the operation was successful, false otherwise.

```
void store(std::ostream &file) const;
```

Writes the student's information to the given stream.

SCHOOL

```
struct SCHOOL {
	std::string name;
	std::string city;
	std::string address;
	std::unordered_map<std::string, STUDENT> students;
	std::unordered_map<std::string, TEACHER> teachers;
	std::unordered_map<size_t, TEAM> teams;
	std::unordered_map<std::string, PROJECT> projects;

	bool restore(std::istream &file);
	void store(std::ostream &file) const;

	std::vector<std::string> getTeachersWithoutTeam(const std::string &project) const;
};
```

Represents an entire school. Has a name, city and address. The students, teachers and projects are stored in unordered maps with strings as keys, while the teams have keys of size_t type.

Member functions

```
bool restore(std::istream &file);
```

Tries to restore the school from the given stream. Returns true if the operation was successful, false otherwise.

```
void store(std::ostream &file) const;
```

Writes the school's information to the given stream.

```
std::vector<std::string> getTeachersWithoutTeam(const std::string &project) const;
```

Returns a vector of strings corresponding to the keys of those teachers who do not have an assigned team[1] for the project pointed to by the key given as an argument.

[1] Note that this uses the team's *project* field, not the project's *teams* vector.

TEACHER

```
struct TEACHER {
	std::string firstName;
	std::string lastName;
	std::string email;
	std::vector<size_t> teams;

	bool restore(std::istream &file);
	void store(std::ostream &file) const;

};
```

Represents a teacher. Has a first and last name and an email. Stores the IDs of all teams which this teacher has been assigned to consult.

Member functions

```
bool restore(std::istream &file);
```

Tries to restore the project from the given stream. Returns true if the operation was successful, false otherwise.

```
void store(std::ostream &file) const;
```

Writes the project's information to the given stream.

TEAM

```
struct TEAM {
	std::string name;
	std::string description;
	std::string setupDate;
	std::vector<TEAM_MEMBER> members;
	STATUS status;
	std::string project;

	bool restore(std::istream &file);
	void store(std::ostream &file) const;

	std::vector<std::string> getMembers() const;
};
```

Represents a team. Has a name, description, an automatically assigned creation date and a status. Keeps track of its members and their roles.

Unfortunately, in this version of the program there does not exist a mechanism which prevents editing of the team's other fields when its *status* is `ARCHIVED`.

Member functions

```
bool restore(std::istream &file);
```

Tries to restore the team from the given stream. Returns true if the operation was successful, false otherwise.

```
void store(std::ostream &file) const;
```

Writes the team's information to the given stream.

TEAM_MEMBER

```
struct TEAM_MEMBER {
	enum ROLE {
		SCRUM_MASTER,
		QA_ENGINEER,
		DEV_BACKEND,
		DEV_FRONTEND
	};

	std::string username;
	ROLE role;

};
```

Represents a member of a team. Stores the unique key of the student and their role. This is only used in the TEAM structure.

Non-member functions

```
bool operator==(const TEAM_MEMBER &lhs, const TEAM_MEMBER &rhs);
```

Returns true if both the username and the role are equal.

Functions

addElement()

(1)
```
bool addElement(std::vector<SCHOOL> &v, const SCHOOL &school);
```
(2)
```
template <typename T>
bool addElement(std::vector<T> &v,
                const T &element)
```
(3)
```
template <typename KEY, typename T>
bool addElement(std::unordered_map<KEY, T> &m,
                const KEY &key,
                const T &element);
```

Adds an element to the given container. Returns true if the element is not present, otherwise false. For *(1)* two schools are considered different if their name is not the same. *(2)* uses `operator==` to decide whether *element* is already present, whereas *(3)* uses *key* to determine uniqueness.

clearConsole()

```
void clearConsole();
```

Clears the console window.

Currently only has an implementation on systems where the Windows API is present.

deleteElement()

(1)
```
template <typename T>
bool deleteElement(std::vector<T> &v,
                   size_t pos);
```
(2)
```
template <typename KEY, typename T>
bool deleteElement(std::unordered_map<KEY, T> &m,
                   const KEY &key);
```

Deletes an element from the given container.

(1) is applied to vectors. *pos* is the index at which the element to delete is. If *pos* is greater than the maximum possible position in the vector, the function returns false. Otherwise it deletes the element and returns true.

(2) is applied to unordered maps. If *key* exists in the map, the corresponding record is deleted with a return value of true. In all other cases the function returns false.

dereferenceElement()

(1)
```
template <typename T>
T &dereferenceElement(const std::vector<T> &v,
                      typename std::vector<T>::iterator it);
template <typename T>
const T &dereferenceElement(const std::vector<T> &v,
                            typename std::vector<T>::const_iterator it);
template <typename KEY, typename T>
T &dereferenceElement(const std::unordered_map<KEY, T> &m,
                      typename std::unordered_map<KEY, T>::iterator it);
template <typename KEY, typename T>
const T &dereferenceElement(const std::unordered_map<KEY, T> &m,
                            typename std::unordered_map<KEY, T>::const_iterator it);
```
(2)

```
template <typename KEY, typename T>
T &dereferenceElement(std::unordered_map<KEY, T> &m,
                      const KEY &key);
template <typename KEY, typename T>
const T &dereferenceElement(const std::unordered_map<KEY, T> &m,
                            const KEY &key);
```
(3)

Returns a reference to a particular element in a container.

(1) returns a reference to the element pointed to by *it*.

(2) returns a reference to the element pointed to by *it*. Since *it*'s value type is a `std::pair`, the reference returned is to the second (value) part.

(3) is the same as *(2)* but instead of an iterator it searches for a specific key.

Note that the second argument must be a valid one.

displayDetails()

```
void displayDetails(const STUDENT &student);
void displayDetails(const TEACHER &teacher);
void displayDetails(const PROJECT &project);
void displayDetails(const SCHOOL &school);
void displayDetails(const TEAM_MEMBER &member, const SCHOOL &parentSchool);
void displayDetails(const TEAM &team, const SCHOOL &parentSchool);
```
(1)

(2)

Displays details about the element passed as the first argument.

(2) also takes a reference to the school in which the element resides in order to be able to resolve the respective keys.

displayMenuOptions()

```
void displayMenuOptions(const std::vector<const char *> &options);
```

Takes a vector of C-strings and displays them as an ordered list starting from 1.

getAnsiEscape()

```
std::string getAnsiEscape(ANSI_ESCAPE colour);
```

Returns a string containing the ANSI escape code which corresponds to the value given to the function as its argument.

getKey()

```
bool getKey(size_t &key);
bool getKey(std::string &key);
```

Utility function used by menu functions when the user needs to enter a key[2] (see menuAdd(), (2))

getMenuChoice()

```
void getMenuChoice(size_t &choice, size_t maxValue, size_t minValue = 1);
```

Displays a prompt to enter a choice giving the user a range of acceptable values. This is repeated until a valid choice is entered. That value gets assigned to the variable given as its first argument.

[2] Refers to a key in a map.

getStudentClass()

```
bool getStudentClass(unsigned &grade, char &classLetter);
```

Accepts user input and tries to parse it as a student class.

Upon success the entered class gets assigned to the corresponding reference arguments and the function returns true.

If the input can't be parsed as a class, the function returns false. The references are left in the state they were in prior to the function being called.

getUnsignedNumber()

```
template <typename T>
void getUnsignedNumber(std::istream &stream,
                       T &x,
                       char delimiter = '\n',
                       unsigned long long maxValue = -1,
                       unsigned long long minValue = 0);
```

Extracts an unsigned integer from *stream*. *T* must be such a type so that an `unsigned long long` may be cast to it.

Extracts from the stream until *delimiter* is reached.

Throws an `std::runtime_error` if it encounters a non-digit character.

If the extracted number is not in the range [*minValue*; *maxValue*] an `std::out_of_range` gets thrown.

hasValidRecords()

```
    template <typename T>
(1) bool hasValidRecords(const std::unordered_map<std::string, T> &m);
    template <typename T>
(2) bool hasValidRecords(const std::unordered_map<size_t, T> &m);
```

Returns whether the unordered map has at least one valid element. *(1)* considers an element as valid if its key is not "INVALID" while for *(2)* the key must be different than 0.

isValidKey()

```
(1) bool isValidKey(const std::string &key);
(2) bool isValidKey(size_t key);
```

Returns whether the given key corresponds to a valid element. For `std::string` keys only "INVALID" is not a valid key and for `size_t` the only invalid value is 0.

listTable()

```
    void listTable(const std::vector<SCHOOL> &schools);
    void listTable(const std::unordered_map<std::string, STUDENT> &students,
                   const SCHOOL &parentSchool);
    void listTable(const std::unordered_map<std::string, TEACHER> &teachers,
                   const SCHOOL &parentSchool);
    void listTable(const std::unordered_map<size_t, TEAM> &teams,
                   const SCHOOL &parentSchool);
    void listTable(const std::unordered_map<std::string, PROJECT> &projects,
                   const SCHOOL &parentSchool);
    void listTable(const std::vector<TEAM_MEMBER> &members,
(1)                const SCHOOL &parentSchool);
```

```
void listTable(const std::vector<size_t> &keys,
               const std::unordered_map<size_t, TEAM> &teams,
               const SCHOOL &parentSchool);
void listTable(const std::vector<std::string> &keys,
               const std::unordered_map<std::string, TEACHER> &teachers,
               const SCHOOL &parentSchool);
```
(2)

Displays a formatted table of the given container's elements.

(1) directly prints a table of the elements.

(2) takes an extra argument – a vector of the keys whose corresponding elements to display. The second argument is mainly used for overload resolution, although the keys are matched to it, not to the parent school's respective container.

All keys in *keys* must be present in the map.

menu()

```
bool menu(std::vector<SCHOOL> &schools);
bool menu(SCHOOL &school);
bool menu(STUDENT &student, const SCHOOL &parentSchool);
bool menu(TEACHER &teacher, const SCHOOL &parentSchool);
bool menu(TEAM &team, const SCHOOL &parentSchool);
bool menu(PROJECT &project, const SCHOOL &parentSchool);
bool menu(TEAM_MEMBER &member, const SCHOOL &parentSchool);
```

Displays and handles all main menus. Returns true if the menu should be shown again, false otherwise.

Displays a list of applicable options and prompts for a choice. After a valid choice has been entered, the function clears the console before calling the appropriate handler for that specific case.

menuAdd()

(1)
```
void menuAdd(std::vector<SCHOOL> &schools);
```
(2)
```
template <typename KEY, typename T>
void menuAdd(std::unordered_map<KEY, T> &m, const SCHOOL &parentSchool);
```
(3)
```
template <typename T>
void menuAdd(std::vector<T> &v, const SCHOOL &parentSchool);
```

Adds a new element checking if it already. An error message is displayed if the operation fails, otherwise a menu gets automatically called to allow the user to edit the new element.

Can perform additional preparations before calling the next menu (see menuAddAdditionalPrep()).

(2) Prompts for a key prior to performing the other actions and checks if that key already exists.

menuAddAdditionalPrep()

(1)
```
void menuAddAdditionalPrep(SCHOOL &school);
```
(2)
```
void menuAddAdditionalPrep(TEAM &team);
```
(3)
```
void menuAddAdditionalPrep(TEAM_MEMBER &member);
```
(4)
```
template <typename T>
void menuAddAdditionalPrep(T &element);
```

Performs additional actions on the newly added element.

(1) adds a special “invalid” student, teacher, team and project to the new school. These elements are used as placeholders where such an element’s key is needed but not known (e.g. when a new team is created, its project is the Invalid project).

(2) sets the team’s project to be the Invalid project and sets the setup date. Please note that this function only compiles when Microsoft’s implementation of localtime_s is available.

(3) sets the team member’s student to be the Invalid student.

(4) is an empty function which only exists for overload resolution purposes when the function is called for a type which does not require additional preparations.

menuDriver()

```
    template <typename T>
(1) void menuDriver(T &element);
    template <typename T>
(2) void menuDriver(T &element, const SCHOOL &parentSchool);
```

Repeatedly calls the appropriate menu function until it returns false.

(1) is only used for the top-level menu and the school menu.

(2) gets called for all other menus. Some of them need information about a different branch of the school’s hierarchy so a const-reference to the school also gets passed down.

menuEditAddress()

```
void menuEditAddress(SCHOOL &school);
```

Displays a prompt showing the current address of the school and asking for a new one. Clears the console after the user has entered the school’s new address.

menuEditCity()

```
void menuEditCity(SCHOOL &school);
```

Displays a prompt showing the current city of the school and asking for a new one. Clears the console after the user has entered the school’s new city.

menuEditClass()

```
void menuEditClass(STUDENT &student);
```

Displays a prompt showing the current class of the student and asking for a new one. If the user enters invalid input, an error message is displayed and the values remain unchanged. Otherwise they are set and the console gets cleared.

menuEditDescription()

```
template <typename T>
void menuEditDescription(T &element);
```

Displays a prompt showing the current description and asking for a new one for all types which have a description. The description can consist of multiple lines so the user must terminate their input with the pipe character (|). The function also deals with the trailing newline left by this action. The console is cleared after this.

menuEditEmail()

```
template <typename T>
void menuEditEmail(T &element);
```

Displays a prompt showing the current email and asking for a new one for all types which have an email. The console is cleared after the user has entered what is asked of them.

menuEditFirstName()

```
template <typename T>
void menuEditFirstName(T &element);
```

Displays a prompt showing the current first name and asking for a new one for all types which have a first name. The console is cleared after the user has entered what is asked of them.

menuEditLastName()

```
template <typename T>
void menuEditLastName(T &element);
```

Displays a prompt showing the current last name and asking for a new one for all types which have a last name. The console is cleared after the user has entered what is asked of them.

menuEditName()

```
template <typename T>
void menuEditName(T &element);
```

Displays a prompt showing the current name and asking for a new one for all types which have a name. The console is cleared after the user has entered what is asked of them.

menuEditProject()

```
void menuEditProject(TEAM &team, const SCHOOL &parentSchool);
```

Displays a list of all projects and prompts the user to choose one. After a successful choice the chosen project gets assigned to the team[3] and the console is cleared.

menuEditRole()

```
void menuEditRole(TEAM_MEMBER &member);
```

Displays a legend for the roles along with the current one and prompts the user for a new one. After a successful choice the chosen role gets assigned to the project and the console is cleared. Otherwise an error message is displayed.

menuEditStatus()

```
template <typename T>
void menuEditStatus(T &element);
```

Displays a legend for the statuses along with the current one and prompts the user for a new one. This function is applicable to all types which have a status. After a successful choice the chosen status gets assigned and the console is cleared. Otherwise an error message is displayed.

menuEditUsername()

```
void menuEditUsername(TEAM_MEMBER &member, const SCHOOL &parentSchool);
```

Displays a table showing all students and prompts the user to select a new one. After a successful choice the new student gets assigned and the console is cleared.

---

[3] Note that currently we do not guarantee logical consistency between teams and projects. This means that it is possible for a project to be assigned to a team and for the team to not be assigned to that project at the same time.

menuLink()

```
template <typename KEY, typename T>
void menuLink(std::vector<KEY> &linkedElements,
              const std::unordered_map<KEY, T> &allElements,
              const SCHOOL &parentSchool);
```

Displays a table of all available elements (using the second argument), then prompts the user for a choice and tries to add that element's key to the *linkedElements* vector. The console is cleared and, if successful, the function exits, otherwise an error message is displayed.

menuQueries()

```
bool menuQueries(const SCHOOL &school);
```

Similar to menu(). Shows a list of available queries to choose from and prompts for a choice.

menuQueryTeachersWithoutTeam()

```
void menuQueryTeachersWithoutTeam(const SCHOOL &school);
```

Displays a list of projects and prompts the user to choose one. After that a table with all teachers who have no teams assigned to them for that particular project is displayed. Uses SCHOOL::getTeachersWithoutTeam() for the query.

menuRemove()

```
    template <typename KEY, typename T>
(1) void menuRemove(std::unordered_map<KEY, T> &m, SCHOOL &parentSchool);
    template <typename T>
(2) void menuRemove(std::vector<T> &v, const SCHOOL &parentSchool);
```

Displays a table of all elements and prompts the user to choose one. It can perform additional tasks (see menuRemoveAdditionalPrep()) before deleting that element.

menuRemoveAdditionalPrep()

```
    void menuRemoveAdditionalPrep(size_t key,
                                  const TEAM &team,
(1)                               SCHOOL &parentSchool);
    void menuRemoveAdditionalPrep(const std::string &key,
                                  const PROJECT &project,
(2)                               SCHOOL &parentSchool);
    void menuRemoveAdditionalPrep(const std::string &key,
                                  const STUDENT &student,
(3)                               SCHOOL &parentSchool);
    template <typename KEY, typename T>
    void menuRemoveAdditionalPrep(const KEY &key,
                                  const T &element,
(4)                               SCHOOL &parentSchool);
```

Performs additional actions before an element is deleted.

Note that for *(1)*, *(2)* and *(3)* the second argument is only used to differentiate between the overloads and can, in fact, be any element of that type (our implementation of menuRemove() calls it with the first element in the container).

(1) searches through all teachers' *teams* vector and removes any occurrences of *key* and does the same for all projects.

(2) searches through all teams and if any team's *project* field matches *key*, it is replaced with the Invalid project's key.

(3) searches through all teams' *members* vector and removes any member whose *username* matches *key*.

(4) is an empty function which only exists for overload resolution purposes when the function is called for a type which does not require additional preparations.

menuRestore()

```
void menuRestore(std::vector<SCHOOL> &schools);
```

If there are currently any schools, the function displays a prompt asking the user to acknowledge that their data will be entirely replaced. Currently we do not support partial and/or non-destructive importing of data. After that restoreSchools() is called.

menuSelect()

```
(1) void menuSelect(std::vector<SCHOOL> &schools);
    template <typename T>
    void menuSelect(std::vector<T> &v,
(2)                 const SCHOOL &parentSchool);
    template <typename KEY, typename T>
    void menuSelect(std::unordered_map<KEY, T> &m,
(3)                 const SCHOOL &parentSchool);
```

Displays a table of all available elements to be selected for editing/browsing and prompts the user to choose. After a successful choice the appropriate menuDriver() is called.

menuStore()

```
void menuStore(const std::vector<SCHOOL> &schools);
```

Displays a prompt asking the user to acknowledge that their backup file will be entirely rewritten. Currently we do not support partial and/or non-destructive exporting of data. After that storeSchools() is called.

menuUnlink()

```
template <typename KEY, typename T>
void menuUnlink(std::vector<KEY> &linkedElements,
                const std::unordered_map<KEY, T> &allElements,
                const SCHOOL &parentSchool);
```

printNewlines()

```
void printNewlines(size_t n);
```

Prints *n* empty lines on the console.

restoreSchools()

```
bool restoreSchools(std::vector<SCHOOL> &schools,
                    const std::string &filename);
```

Tries to open a file with name *filename* and import the information in it. Returns true on success and false otherwise.

storeSchools()

```
bool storeSchools(const std::vector<SCHOOL> &schools,
                  const std::string &filename);
```

Tries to open a file with name *filename* and export the current data in the system to it. If such a file does not exist, it is created. Returns true on success and false otherwise.