2023

QT Project

PREMIER LEAGUE

NAYDENOVA NIKA 221-2

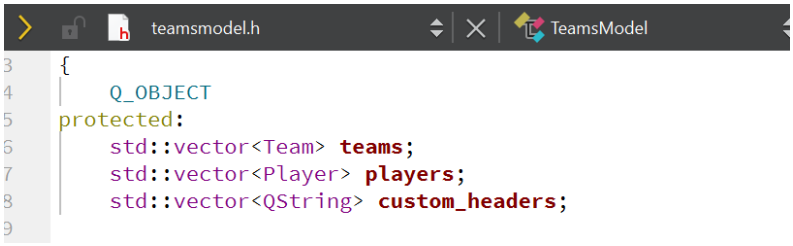
Problem statement

Making an application to easy analyze teams and football players in Premier League 2021-2022. Application allows users to add, remove, edit players and to make a prediction on the winning team.

Implementation details

First of all, I created two classes (Player and Teammodel) to simplify the process of dividing the data in my App. In teamsmodel.cpp I realized the function TeamsModel::parseLinesToTeams(QFile& file). It parses lines of the file (In case file can be open, otherwise there will appear MessageBox with “error”. I need only to read the file, here there are no changes -> I use QIODevice::ReadOnly.
`RowCount` and `columnCount` return the number of rows and columns in data, the way to finish their implementation is to access data from file. `Qt::DisplayRole` string, for display purposes.

In csv file, data in line is separated with ‘,’, so I use it as a delimetor. To keep data, I created vectors of classes.



By specification, I didn’t use all columns in dataset, so I skipped some ‘,’ while parsing. With functions in teamsmodel.h I described my model of table. Moreover, in mainwindow I had to impellent several addition functions to summarize the aggregate goals of each team, define a leader

```
    auto element = std::find(teams.begin(), teams.end(), team);

    if (element != teams.end()) {
        element->incrementCount();
        element->addGoals(currentGoals);
        if (element->getGoalLeader() < currentGoals) {
            element->setLeader(currentPlayerName);
        }
    }
    else {
        Team newTeam = Team();
        newTeam.setName(players.at(i).team);
        newTeam.incrementCount();
        newTeam.addGoals(currentGoals);
        newTeam.setGoalLeader(currentGoals);
        newTeam.setLeader(currentPlayerName);
        teams.push_back(newTeam);
    }
}
```

After reading data, adding headers and “playing” with styles of text in labels and adding push_buttoms as a sketch, I started implementing sorting. For that, in class Team I defined functions (getName() and etc.) to easily use them in teamsmodel. I performed sorting with comparators.

```
bool cmpByGoals(Team team1, Team team2)
{
    return team1.getGoals() >team2.getGoals();
}
bool cmpByLeader(Team team1, Team team2)
{
    return team1.getLeader()<team2.getLeader();
}
void TeamsModel::sortByName(){
    std::sort(teams.begin(), teams.end(), cmpByName);
}
void TeamsModel::sortByNumb(){
    std::sort(teams.begin(), teams.end(), cmpByNumbPlayers);
```

In QComboBox I appended all names of columns to then sort the whole table in App by resetting the model of table and then adding once again a sorted one. The kind of sort I defined by indexchanged function.

```
void MainWindow::on_comboBox_currentIndexChanged(int index)
{
    if (index==0)
    {
        ui->playerTableView->reset();
        teamsModel->sortByName();
        ui->playerTableView->setModel(teamsModel);

    }
    else if (index==1)
    {
        ui->playerTableView->reset();
        teamsModel->sortByNumb();
        ui->playerTableView->setModel(teamsModel);
    }
```

Then I added Dialog window as “help”. In this window, there is my logotype container which keeps custom widget draw. There, with instruments (pen, brush, setPen, drawEllipse and etc.) of QPainter I and id (as label).

```
draw::draw(QWidget* parent)
    : QWidget(parent)
{
}
void draw::paintEvent(QPaintEvent*)
{
    QPainter painter;
    painter.begin(this);
    QPen pen;
    pen.setColor(Qt::green);
    pen.setWidth(7);
    QBrush brush;
    brush.setColor(Qt::yellow);
    painter.setPen(pen);
    painter.setBrush(brush);
```

To have a better glance at each team, I created secondwindow (it is like second main window). There each team will be stored. I decided to do it this way: user clicks to any cell of a table with data in mainwindow -> then with function getTeamName it defines the name of team of chosen football player (on_playerTableView_doubleClicked contains a former function and opens secondWindow.

```
void MainWindow::on_playerTableView_doubleClicked(const QModelIndex &index)
{
    QString name = teamsModel->getTeamName(index);
    secondwindow* currTeam= new secondwindow(name);
    currTeam->show();

}
```

```
QString TeamsModel::getTeamName(const QModelIndex& index){
    int row = index.row();
    return teams.at(row).getName();
    }
```

The label in secondWindow is automatically changed by the name of current chosen team. To show model of a team in new window, I used quite similar approach as with mainwindow, but since that moment, I had been working with a different class Player. Similarly to teamSmodel, I implemented sorting (in a very same way as I have already described).

Here I added dialog windows to add new player and delete player(s). Adding is realized with EditLines in UI, after clicking the button “Add player”, the file with data ‘uploads’ (actually, regenerates with a new row (in which texts of EditLine are already added) and saves, team model updates).

```
void dialogAdd::on_buttonAdd_clicked()
{
    QString name = ui->editName->text();
    QString nation = ui->editNation->text();
    QString position = ui->editPosition->text();
    QString goals = ui->editGoals->text();
    int begin = 0;
    QValidator::State check = intvalid.validate(goals, begin);
    if (check == 0)
    {
        QMessageBox *mess = new QMessageBox();
        mess->setText("Invalid input");
        mess->show();
        return;
    }
    QString assists = ui->editAssists->text();
    QFile file("C:\\Users\\inaydenova\\Documents\\untitled5\\footba
    if (!file.open(QIODevice::Append))
    {
```

```
void secondwindow::on_button_add_clicked()
{
    dialogAdd *dialog = new dialogAdd(this->name, this); //this - parent of dialogWindow
    dialog->exec();
}
```

To delete, I added another dialog window. Once the button delete is clicked, I connect it with another function, which collects id and then goes to another function which works with our data file.

```
void secondwindow::on_button_delete_clicked()
{
    deleteDialog *dialogdel = new deleteDialog("player(s)");
    QObject::connect(dialogdel, SIGNAL(finished(int)), this, SLOT(dialog_delete_finished(int)));
    dialogdel->exec();
}
```

```
void TeamModel::deleteSelectedPlayers(std::vector<QString> names)
{
    QFile file(nameFile);
    if (file.open(QIODevice::ReadWrite))
    {
        QString str;
        QTextStream textStream(&file);
        while(!textStream.atEnd())
        {
            bool flagDelete = false;

            QString line = textStream.readLine();
            for (int i = 0; i<names.size(); ++i)
```

Similarly, I create new dialog window which is accessible via double click on any cell of team model. there are again Edit Lines which are, by default, filled with the values of the original line. Then I repeat the procedure on working with file (but the algorithm is a bit different). There is a checker which verify filled gaps (QIntValidator)

I didn't create a separate "Save" button because I save changed file simultaneously with functions of changes.

There is a button "Make prediction". This is a dialog window with two ComboBoxes. User select two teams and then there is a function which calculates the sums of goals os both teams and outputs the winner and % of chances.

Results

In Application, I realized storing, saving, sorting, ordering, editing, adding, specific calculations, creating a logotype, removal. I suppose, that sorting is not ideal, because csv file with data from Kaggle contains special symbols that I didn't manage to decode. I also included not precise checker to verify the type of data which user adds or edits (there is no verification of string cells). Moreover, I believe there is another, more efficient approach to read the data (in my project it is necessary to have the exact csv file and change the path to it, if user uses another PC). I tried to include feature that translates the interface if app into other languages, however I didn't manage to implement it.

Conclusion

From my point of view, my application is quite user-friendly and efficient. However, saving and showing new changed data could be improved (in my app, after editing or removing or adding, it requires to open second window one more time to see changes). It would also be better to serve more than one language.