```
/*************************************************************************
                           DataManager  -  description
                             -------------------
    begin                : 23/11/2015
    copyright            : (C) 2015 by Edern Haumont & Nicolas Six
*************************************************************************/

//------- Realisation of the class DataManager (file DataManager) --------

//---------------------------------------------------------------- INCLUDE

//-------------------------------------------------------- System include
#include <algorithm>
//------------------------------------------------------ Personal include
#include "DataManager.h"
#include "config.h"

// don't use: "using namespace std;" to keep clear that we use std and
// not any other library and by the same way keeping ready to use an other
// library than the std.

//----------------------------------------------------------------- PUBLIC

//--------------------------------------------------------- Public methods

// Constructor
DataManager::DataManager() {
    std::ifstream extensionFile (EXTENSION_FILE);
    for(std::string extension ; std::getline(extensionFile,extension) ; )
    {
        excludedExtension.push_back(extension);
    }
    extensionFile.close();

    for (int i = 0; i < DATA_TAB_SIZE; ++i)
    {
        data[i] = nullptr;
    }
}

// Destructor
DataManager::~DataManager()
// Algorithm : Run through the graph and delete all dynamic elements.
{
    for (int c = 0; c < DATA_TAB_SIZE; c++)
    {
        if(data[c] != nullptr)
        {
            //iterate through the from node:
            for(dataFromLevel::iterator f=data[c]->begin() ; f!=data[c]->end() ; ++f)
            {
                //iterate through the referrer branches:
                for(dataDestinationLevel::iterator d=f->second->begin() ; d!=f->second->end() ; ++d)
                {
                    delete [] d->second;
                }
                delete f->second;
            }
            delete data[c];
        }
    }
}

int DataManager::LoadLogFile(const std::string &logFilePath)
// Algorithm :
// Open a log file. Reads line by line its content until end of file is reached.
// Each line is put in a string stream. Then it is parsed to obtain all its characteristics
{
    std::ifstream logFile(logFilePath, std::ios::in);  // on ouvre le fichier en lecture
    if(!logFile)
    {
        std::cerr << "erreur lors de l'ouverture du fichier de log: " << logFilePath << std::endl;
        return 1;
    }
    else
    {
        std::string logLine;

        std::string ip;
        tm time;
        unsigned int httpCode;
        std::string sizeTransfered;
        unsigned int sizeTransferedValue;
        std::string browser;
        std::string logname;
        std::string pseudo;
        std::string request;
        int GMT;
        std::string unusedBuffer;
        std::string dateBuffer, GMTBuffer;
        std::string protocolRequest;
        std::string URLRequest;
        std::string refferer;

        //loops until end of file or bad reading
        while(getline(logFile,logLine))
        {
            try
            {
                std::stringstream ss(logLine);
                ss >> ip >> logname >> pseudo >> dateBuffer >> GMTBuffer >> request;
                std::string bufferString;
                getline(ss, bufferString, '"');
                unsigned long lastSpace = bufferString.find_last_of(" ");
                URLRequest = bufferString.substr(1,lastSpace-1);
                protocolRequest = bufferString.substr(lastSpace+1, bufferString.length()-lastSpace-1);
                ss >> httpCode >> sizeTransfered >> refferer;
                if(sizeTransfered.compare("-") ==0)
                {
                    sizeTransferedValue = 0;
                }
                else
                {
                    sizeTransferedValue = (unsigned)atoi(sizeTransfered.c_str());
                }
                request.append(" ");
                request.append(URLRequest);
                request.append(" ");
                request.append(protocolRequest);

                //date extraction
                time.tm_mday = atoi(dateBuffer.substr(1,2).c_str());
                std::string Month [] = 
{"Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"};
                for(int i=0;i<12;i++)
                {
                    if (Month[i].compare(dateBuffer.substr(4, 3)) == 0) {
                        time.tm_mon = i;
                        break;
                    }
                }
                time.tm_year = atoi(dateBuffer.substr(8,4).c_str());
                time.tm_hour = atoi(dateBuffer.substr(13,2).c_str());
                time.tm_min = atoi(dateBuffer.substr(16,2).c_str());
                time.tm_sec = atoi(dateBuffer.substr(19,2).c_str());
                GMT = atoi(GMTBuffer.substr(1,4).c_str()); // /100 ? ( 0200 -> 2h)
                GMT *= (GMTBuffer.substr(0,1) == "-") ? -1 : 1;

                // referrer extraction and management
                if(refferer.length()>32 && refferer.substr(1,32).compare("http://intranet-if.insa-
lyon.fr/")==0)
                {
                    refferer = refferer.substr(32);
                }
                else
                {
                    refferer = refferer.substr(1);
                }
                refferer = refferer.substr(0,refferer.size()-1);

                getline(ss, unusedBuffer, '"');
                getline(ss, browser, '"');

                LogOtherInfos other
(ip,time,httpCode,sizeTransferedValue,browser,logname,pseudo,request,GMT);
                //add to structure
                add(refferer, URLRequest, (unsigned)time.tm_hour, httpCode, other);
            }
            catch (std::exception e)
            {
                std::cerr << e.what() << " when reading the log file" << std::endl;
            }
        }
    }



    return 0;
} // end of method

int DataManager::Request(const bool optionT, const int tHour, const bool optionE, const bool optionG,
const std::string &outputFile)
// Algorithm : depends on the options.
// Runs through the structure to find most popular URL.
// if optionG checked, associate referrer to destination URL in a .dot
{
    if(optionG)
    {
        graph = new GraphGenerator(outputFile);
    }


    int hourMin=0,hourMax=24;
    if(optionT)
    {
        hourMin = tHour;
        hourMax = tHour+1;
    }

    std::vector< pageAndHits > pageHit;

    for (int c = 0; c < 1; c++)
    {
        if(data[c] != nullptr)
        {
            //iterate through the from node:
            for(dataFromLevel::iterator f=data[c]->begin() ; f!=data[c]->end() ; ++f)
            {
                //option -e filter: if the option is activated then only select the specified extension
                if( !optionE || isNotExcludedDocument(f->first) )
                {
                    int numberOfHitsByPage=0;

                    //iterate through the referrer branches:
                    for(dataDestinationLevel::iterator d=f->second->begin() ; d!=f->second->end() ; +
+d)
                    {
                        int numberOfHitsByReferrer = 0;
                        for (int h=hourMin ; h<hourMax ; h++)
                        {
                            for (unsigned i = 0; i < d->second[h].size(); ++i)
                            {
                                if(d->second[h].at(i).getRequest().substr(1,3).compare("GET")==0)
                                {
                                    numberOfHitsByReferrer++;
                                }
                            }
                        }
                        if(optionG)
                        {
                            graph->addLinkToGraph(f->first,d->first,std::to_string
(numberOfHitsByReferrer));
                        }
                        numberOfHitsByPage += numberOfHitsByReferrer;
                    }
                    if(numberOfHitsByPage != 0)
                    {
                        pageAndHits tuple(f->first, numberOfHitsByPage);
                        pageHit.push_back(tuple);
                    }
                }
            }
        }
    }

    if(optionG)
    {
        delete graph;
    }

    std::sort(pageHit.begin(),pageHit.end(),&compareDateAndHits);

    for (unsigned i=0 ; i<10 && i<pageHit.size() ; i++)
    {
        std::cout << pageHit.at(i).first << " (" << pageHit.at(i).second << " hits)" << std::endl;
    }

    return 0;
}

int DataManager::add(const std::string &referrer, const std::string &destination, unsigned int hour, \
                     unsigned int httpCode, const LogOtherInfos &other)
// Algorithm : runs through the structure
{
    unsigned int indexHttpCode = transformToTabIndex(httpCode);

    dataDestinationLevel* referrerMap;

    if(data[indexHttpCode] == nullptr)
    {
        data[indexHttpCode] = new dataFromLevel();
    }

    //try to add the referrer level to the destination level (if he already exist does nothing)
    if(data[indexHttpCode]->find(destination) == data[indexHttpCode]->end())
    {
        referrerMap = new dataDestinationLevel();
        std::pair<std::string,dataDestinationLevel*> insertionPairDest(destination, referrerMap);
        data[indexHttpCode]->insert(insertionPairDest);
    }
    else
    {
        referrerMap = data[indexHttpCode]->at(destination);
    }

    //try to add the hour level to the referrer level (if he already exist does nothing)
    if(referrerMap->find(referrer) == referrerMap->end())
    {
        dataHourLevel * tempHourLevelVector = new dataHourLevel[24];
        for (int i = 0; i < 24; i++)
        {
            dataHourLevel temp;
            tempHourLevelVector[i] = temp;
        }
        std::pair<std::string,dataHourLevel*> insertionPairHour(referrer, tempHourLevelVector);
        referrerMap->insert(insertionPairHour);
    }
```

```cpp
    dataHourLevel * hourLevel = referrerMap->at(referrer);
    hourLevel[hour].push_back(other);

    return 0;
}




bool DataManager::compareDateAndHits(const pageAndHits &A, const pageAndHits &B)
// function made to order the values by number of hits and then by name of the page
{
    return (A.second > B.second) || ((A.second == B.second) && (A.first.compare(B.first)<0));
}

bool DataManager::isNotExcludedDocument(const std::string &pagePath) const
{
    if(pagePath.find('.') != std::string::npos)
    {
        std::string extension = pagePath.substr( pagePath.find_last_of('.'));

        for (unsigned i = 0; i < excludedExtension.size(); ++i) {
            if(extension.compare(excludedExtension.at(i))==0)
            {
                return false;
            }
        }
    }

    return true;
}

unsigned DataManager::transformToTabIndex(int httpCode) const {
    //equivalent to: (httpCode-100)/300 but handel error case:
    if(httpCode >= 100 && httpCode < 400)
    {
        return 0;
    }
    else // if on [400;600[ or if any error
    {
        return 1;
    }
}
```