```
/*************************************************************************
Collection  -  description
-------------------
début                : 2015
copyright            : (C) 2015 par mfallouh mvirsolvy
*************************************************************************/

//---------- Réalisation de la classe <Collection> (fichier Collection.cpp) --

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

//-------------------------------------------------------- Include système
#include <fstream>
#include <iostream>
#include <list>
using namespace std;
//------------------------------------------------------ Include personnel
#include "Collection.h"
//------------------------------------------------------------------ Constantes
extern const char SEP_REQ, SEP, SEP_DATE_DEBUT, SEP_DATE_FIN, SEP_HEURE, SEP_PT,
SEP_INT, SEP_PVIRG; // les séparateurs
extern const string EXCLUSIE[];
extern const int NB_EXTENSIONS;
const unsigned int NOMBRETOP = 10;       //nombre de cibles à afficher dans le top
des plus consultées
//------------------------------------------------------ Variables de classe

//----------------------------------------------------------- Types privés


//------------------------------------------------------------------ PUBLIC
//----------------------------------------------------------- Fonctions amies

//-------------------------------------------------------- Méthodes publiques

void Collection::Top10(const bool e, const int t) const
// Algorithme : on remplit progressivement une liste avec les éléments les plus
consultés
// On vérifie pour chaque élément si il est déjà contenu dans cette liste et si le
nombre de hits dépasse le max actuel
// Pour gérer les collections de moins de 10 Cible, on vérifie si on a vidé la
collection
// Pour gérer les égalités, on mémorise la valeur de hits minimale du top
//
{
       typedef pair<string, int> elementTop;    //pour les cibles du top : nom +
nombre de hits
       list<elementTop> leTop;           //liste du top10
       bool fini = false;  //booleen pour vérifier les égalités, passe à true une
fois qu'on est sur qu'il n'y a plus d'égalité ou que la map est vide
       int max = 0;  //maximum temporaire à chaque itération
       string cibleMax;     //cible associée au maximum temporaire
       int min;      //minimum de hits actuel du top
       int cpt;      //stockage temporaire des comptes, pour éviter de refaire les
opérations plusieurs fois
       bool dejaDansTop;    //booleen permettant de savoir si la cible est deja
dans le top
```

```
	if (e) //option e spécifiée
	{
		string extensionFic;//extension du fichier à vérifier
		size_t debut;
		while (!fini)
		{
			max = 0;
			cibleMax = "";
			for (auto const &it1 : pages)	//parcours du dictionnaire
de Cible
			{
				/*vérification du type de fichier*/
				debut = it1.first.rfind(SEP_PT)+1;
				extensionFic = it1.first.substr(debut,
distance(it1.first.begin(), it1.first.end()) - debut);
				if (find(EXCLUSIE, EXCLUSIE + NB_EXTENSIONS,
extensionFic) == EXCLUSIE + NB_EXTENSIONS)		//extension n'est pas dans
la liste des extensions à exclure
				{
					cpt = it1.second.Compte("GET", e, t);
					if (cpt > max)	//nombre de hits supérieur
au max courant
					{
						dejaDansTop = false;
						for (auto const &it2 : leTop)
	//vérification que la Cible ne soit pas déjà dans le top
						{
							if (it2.first == it1.first)
	//deja dans le top
							{
								dejaDansTop = true;
							}
						}
						if (!dejaDansTop)	//pas deja dans le
top
						{
							max = cpt;		//MAJ des max
et cibleMax
							cibleMax = it1.first;
						}
					}
				}
			}	//on a trouvé le max dans le dictionnaire privé du top

			if (max == 0) //dico vide
			{
				fini = true;
			}
			else if (leTop.size() < NOMBRETOP)	//il reste de la
place, on insère et on continue
			{
				elementTop aInserer = { cibleMax, max };
				leTop.push_back(aInserer);
				min = max;	//on met à jour le minimum du top
			}
			else	//il n'y a plus de place
			{
```

```
                        if (max == min)            //cas d'égalité, on insère
quand même, et on continue
                        {
                                elementTop aInserer = { cibleMax, max };
                                leTop.push_back(aInserer);
                        }
                        else   //il n'y a pas égalité
                        {
                                fini = true;
                        }
                }
        }       //fin while(!fini)
    }
    else   //option e non-spécifiée
    {
        while (!fini)
        {
                max = 0;
                for (auto const &it1 : pages)     //parcours du dictionnaire
de Cible
                {
                        cpt = it1.second.Compte("GET", e, t);
                        if (cpt > max)
                        {
                                dejaDansTop = false;
                                for (auto const &it2 : leTop)     //vérification
que la Cible ne soit pas déjà dans le top
                                {
                                        if (it2.first == it1.first)
                                        {
                                                dejaDansTop = true;
                                        }
                                }
                                if (!dejaDansTop)
                                {
                                        max = cpt;
                                        cibleMax = it1.first;
                                }
                        }
                }       //on a trouvé le max dans le dictionnaire privé du top
                if (max == 0) //dico vide
                {
                        fini = true;
                }
                else if (leTop.size() < NOMBRETOP)       //il reste de la
place, on insère et on continue
                {
                        elementTop aInserer = { cibleMax, max };
                        leTop.push_back(aInserer);
                        min = max;    //on met à jour le minimum du top
                }
                else   //il n'y a plus de place
                {
                        if (max == min)            //cas d'égalité, on insère
quand même, et on continue
                        {
                                elementTop aInserer = { cibleMax, max };
                                leTop.push_back(aInserer);
```

```
                        }
                        else   //il n'y a pas égalité
                        {
                                fini = true;
                        }
                }
        }       //fin while(!fini)
    } // on a notre top bien formé
    for (auto const &iterTop : leTop) //affichage
    {
        cout << iterTop.first << " (" << iterTop.second << " hits)" << endl;
    }
}//----- Fin de Top10


/*//------------------------------------------------- Surcharge d'opérateurs
Collection & Collection::operator = ( const Collection & unCollection )
// Algorithme :
//
{
} //----- Fin de operator =

*/
//-------------------------------------------- Constructeurs - destructeur
/*Collection::Collection(const Collection & unCollection)
// Algorithme :
//
{
#ifdef MAP
    cout << "Appel au constructeur de copie de <Collection>" << endl;
#endif
} //----- Fin de Collection (constructeur de copie)
*/


Collection::Collection(const string & nomFichier)
// Algorithme : ouvre un flux de lecture de fichier.
// Tant que le fichier n'est pas terminé, traite les logs ligne par ligne.
// Identifie les cibles et les crée dans la collection
// Groupe les logs par cible et par requête
//
{
#ifdef MAP
    cout << "Appel au constructeur de <Collection>" << endl;
#endif
    ifstream fichier (nomFichier.c_str());
    if (fichier.good()) //fichier trouvé et non vide
    {
        string ligneLog;
        while (getline(fichier, ligneLog))             //tant que l'on a pas
fini de lire le fichier
        {
            /* extraction de la cible de la requête */
            if (ligneLog.find(SEP_REQ) == string::npos ||
ligneLog.find(SEP) == string::npos)
            {
                cerr << "Log invalide : " << ligneLog << endl;
                return;
```

```
                    }
                    size_t debut = ligneLog.find(SEP_REQ) +1;
                    debut = ligneLog.find(SEP, debut)+1;
                    size_t fin = ligneLog.find(SEP_REQ, debut);
                    string adrCible = ligneLog.substr(debut, fin - debut);
                    fin = ligneLog.find(SEP, debut);
                    adrCible = ligneLog.substr(debut, fin - debut);
                    if (adrCible.rfind(SEP_URL) != string::npos)    // l'URL
contient un '/'
                    {
                           adrCible = adrCible.substr(0,
adrCible.find(SEP_PVIRG)); //enlever tout ce qui est apres un ;
                           adrCible = adrCible.substr(0, adrCible.find(SEP_INT));
//enlever tout ce qui est après un ?
                    }

                    /* mise à jour du dictionnaire de Cible */
                    Cible cibleInser;
                    cibleInser.Ajouter(ligneLog);
                    pair<string, Cible> aInserer = { adrCible, cibleInser };
      // créér une paire pour l'insertion
                    pages.insert(aInserer);    //tentative d'insertion, ajoute la
requete si elle n'est pas presente
                    pages.find(adrCible)->second.Ajouter(ligneLog); //on ajoute le
nouveau log à la Cible
             }      //fin du fichier atteinte
      }
      else   //fichier non trouvé
      {
             cerr << "Fichier " << nomFichier << " non trouve !" << endl;
      }


} //----- Fin de Collection

Collection::~Collection()
// Algorithme :
//
{
#ifdef MAP
      cout << "Appel au destructeur de <Collection>" << endl;
#endif
} //----- Fin de ~Collection


  //------------------------------------------------------------------ PRIVE

  //------------------------------------------------------ Méthodes protégées

  //-------------------------------------------------------- Méthodes privées
```