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

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

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

//-------------------------------------------------------- Include système
#include <map>
#include <string>
#include <iostream>
//------------------------------------------------------ Include personnel
#include "Cible.h"
//------------------------------------------------------------------ Constantes
extern const int NB_HEURES;
extern const char SEP_REQ, SEP, SEP_DATE_DEBUT, SEP_DATE_FIN, SEP_HEURE;
extern const string EXCLUSIE[];
extern const int NB_EXTENSIONS;
//------------------------------------------------------ Variables de classe

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


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

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


int Cible::Ajouter(const string & log)
// Algorithme :
// Extrait le type de requête, ainsi que l'heure d'émission de la requête
// Après calcul de l'heure de Greenwich, insère le log à la bonne heure
{

	Log nouveauLog = Log(log);                      //     log à ajouter
	string requete;            // nom de la requête (GET, POST..)
	string date;         // informations de date
	int heureLocale;     //heure locale
	int decalage;        //decalage par rapport à Greenwich
	int heureGreenwich;  //heure de Greenwich
	size_t debut = log.find(SEP_REQ) +1;
	size_t fin = log.find(SEP, debut);
	if (debut != fin)
	{
		requete = log.substr(debut, fin - debut);
	}
	else
	{
		cerr << "erreur de parsing de l'adresse cible : requete" << endl;
	//ERREUR
		return 1;
	}
```

```
	/*recherche de la date dans le log*/
	debut = log.find(SEP_DATE_DEBUT) +1;
	fin = log.find(SEP_DATE_FIN);
	if (debut != fin)
	{
		date = log.substr(debut+1, fin - debut);
	}
	else
	{
		cerr << "erreur de parsing de l'adresse cible : date" << endl;
	//ERREUR
		return 1;
	}

	/*recherche de l'heure locale dans la date*/
	debut = date.find(SEP_HEURE) +1;
	fin = date.find(SEP_HEURE, debut);
	if (debut != fin)
	{
		heureLocale = stoi(date.substr(debut, fin - debut));
	}
	else
	{
		cerr << "erreur de parsing de l'adresse cible : heure locale" <<
endl;	//ERREUR
		return 1;
	}

	/*recherche du décalage horaire dans la date*/
	debut = date.find(SEP, fin) +1;
	string::iterator itFin = date.end();
	fin = distance(date.begin(), itFin);
	if (debut != fin)
	{
		decalage = stoi(date.substr(debut, fin - debut)) / 100;
		heureGreenwich = heureLocale - decalage;
		if (heureGreenwich > 23)	//gestion des dépassements faite à la main
car l'opérateur '%' tolère les valeurs négatives
		{
			heureGreenwich -= 24;
		}
		else if (heureGreenwich < 0)
		{
			heureGreenwich += 24;
		}
	}
	else
	{
		cerr << "erreur de parsing de l'adresse cible : decalage greenwich"
<< endl;	//ERREUR
		return 1;
	}


#ifdef MAP
	cout << "Insertion du log -- heure = " << heureGreenwich << endl;
#endif
```

```
	pair<map<string, list<Log>>::iterator, bool> insertion;
	list<Log> listeLogs;
	listeLogs.push_back(nouveauLog);
	pair<string, list<Log>> aInserer = { requete, listeLogs };   // créér une
paire pour l'insertion
	insertion = lesLogs[heureGreenwich].insert(aInserer); //tentative
d'insertion, on récupère le résultat
	if (!insertion.second)     //si la liste existait déjà dans la map
	{
		lesLogs[heureGreenwich].find(requete)->second.push_back(nouveauLog);
	//on ajoute le nouveau log à la liste
#ifdef MAP
		cout << "Requete " << requete << " deja presente pour la Cible, ajout
a la liste" << endl;
#endif
	}

#ifdef MAP
	cout << "Log correctement ajoute a la Cible" << endl;
#endif

	return 0;
}	//fin de Ajouter

int Cible::Compte(const string & requete, const bool e,  const int t) const
// Algorithme :
// Compte le nombre de hits en utilisant la taille de la liste de logs
// Tri par requete, filtrage de l'heure si spécifié en paramètre
{
	int compte = 0;	//variable de retour
	list<Log>::iterator itListe;	//itérateur pour le parcours des listes
	if (!e)	//pas d'option e
	{
		if (t == -1)  //pas d'option t
		{
			for (int heure = 0; heure < 24; heure++)
			{
				if (lesLogs[heure].find(requete) !=
lesLogs[heure].end())	//si la requete est présente dans le dictionnaire
				{
					compte += lesLogs[heure].find(requete)-
>second.size();
				}
			}
		}
		else   //option t spécifiée
		{
			if (lesLogs[t].find(requete) != lesLogs[t].end())     //si la
requete est présente dans le dictionnaire
			{
				compte += lesLogs[t].find(requete)->second.size();
			}
		}
	}	//fin pas d'option e
	else   //option e spécifiée
	{
		string extensionFic;
```

```
		size_t debut;
		if (t == -1)  //pas d'option t
		{
			for (int heure = 0; heure < 24; heure++) //parcours de toutes
les heures
			{
				if (lesLogs[heure].find(requete) !=
lesLogs[heure].end())	//si la requete est présente dans le dictionnaire
				{
					for (auto const &it :
lesLogs[heure].find(requete)->second)		//on parcourt la liste de Log
					{
						if ((debut = it.referer.rfind(SEP_PT)) !=
string::npos) //si on trouve un '.'
						{
							debut++;
							extensionFic =
it.referer.substr(debut, distance(it.referer.begin(), it.referer.end()) - debut);
	//on récupère la potentielle extension
							if (find(EXCLUSIE, EXCLUSIE +
NB_EXTENSIONS, extensionFic) == EXCLUSIE + NB_EXTENSIONS)		//si cette
extension n'est pas à retirer des résultats
							{
								compte++;
							}
						}	//fin de si on trouve un '.'
						else	//pas de '.' trouvé
						{
							compte++;
						}
					}	//fin parcours des logs
				}	//fin requête présente dans le dictionnaire
			}	//fin parcours de toutes les heures
		}	//fin pas d'option t
		else	//option t spécifiée
		{
			if (lesLogs[t].find(requete) != lesLogs[t].end())	//si la
requete est présente dans le dictionnaire
			{
				for (auto const &it : lesLogs[t].find(requete)->second)
		//on parcourt la liste de Log
				{
					if ((debut = it.referer.rfind(SEP_PT)) !=
string::npos) //si on trouve un '.'
					{
						debut++;
						extensionFic = it.referer.substr(debut,
distance(it.referer.begin(), it.referer.end()) - debut);	//on récupère la
potentielle extension
						if (find(EXCLUSIE, EXCLUSIE +
NB_EXTENSIONS, extensionFic) == EXCLUSIE + NB_EXTENSIONS)		//si cette
extension n'est pas à retirer des résultats
						{
							compte++;
						}
					}	//fin de si on trouve un '.'
					else	//pas de '.' trouvé
					{
```

```
                                        compte++;
                                    }
                                }      //fin parcours des logs
                            }      //fin requête présente dans le dictionnaire
                        }      //fin option t spécifiée
                }      //fin option e spécifiée
                return compte;
}       //fin de Compte

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

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


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


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


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

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

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