Date : 18/12/2018 - Auteurs : PAFFI Lucca, WALLYN Valentin

TP4 C++

Table des matières :

I. Introduction

L'objectif de ce TP est d'appliquer l'utilisation de la Standard Library C++ (STL) et de manipuler des flux de données lors de la conception d'une application. Nous avons essayé de développer des solutions génériques et réutilisables, qui peuvent facilement être adaptées pour des nouveaux problèmes.

Plus spécifiquement, notre application doit lire un fichier journal qui contenait des logs Apache (des lignes décrivant des "parcours" entre des pages Web), les modéliser et les ranger dans une structure de données qui nous permettrait de les manipuler facilement. Étant donné que le fichier peut être très volumineux, le choix d'une structure de données cohérente devient crucial.

Ce projet a été fait en deux parties principales : la conception des classes et la réflexion aux structures de données qui seraient utilisées pour mieux répondre à la problématique d'une part et l'implémentation en langage C++ d'autre part.

II. Spécifications

Le programme fonctionne en ligne de commande. L'installation est faite en suivant les instructions dans le fichier README.md. Nous pouvons découper le fonctionnement général en trois parties :

Main :

Point d'entrée du programme : réalise la lecture du fichier et le traitement (filtrage ou non) des logs.

Cas normaux :

- On peut mettre les options dans n'importe quelle ordre (comme une commande linux classique). L'argument principal (fichier d'entrée) doit être le dernier argument **(Tests 6,7,8,9,10,11)**
- Options possibles :
  - -g nomFichier.dot : génère un fichier GraphViz **(Test 16)**
  - [default] Afficher les 10 documents les plus consultés (nom et nombre de hits associés) **(Test 13)**
  - -e : exclut documents de type image, CSS ou JS. **(Test 14)**
  - -t heure : filtre sur les heures [heure, heure+1[**(Test 15)**
  - -h : affiche le manuel d'utilisation du programme

Cas limites:

- Si nomFichier.dot est déjà existant : on demande confirmation à l'utilisateur **(Test 4)**
- Option inconnue : option ignorée **(Test 18)**
- Absence d'argument nom de fichier : afficher le manuel d'utilisation**(Test 21)**

Cas d'erreurs :

- Le paramètre t doit être un entier entre 0 et 23. **(Test 5, 12 et 17)**
- « Heure » et « nomFichier.dot » sont des arguments obligatoires : leur absence doit générer une erreur. **(Test 1 , 2)**

Parsing et traitement des données :

Cas normaux :

- Pour chaque ligne de log, on ajoute la requête à la structure de données. Si la cible est déjà connue, incrémenter son compteur de hits et ne pas l'ajouter une seconde fois. (de même pour le referer)
- On doit pouvoir exclure tous les documents images/css/javascript **(Test 14)**
  - Est définie comme « image », une requête contenant une des extensions suivantes : .jpg, .jpeg, .jfif, .jif, .png, .gif, .webp, .svg, .ico, .tiff, .tif, .jp2, .jpx, .j2k, .fpx, .pcd
  - Est définie comme « css » une requête contenant une des extensions suivantes : .css
  - Est définie comme « javascript » une requête contenant une des extensions suivantes : .js

Cas d'erreurs :

- Ligne de log mal formée : afficher une erreur de format du fichier **(Test 19)**

Génération du GraphViz :

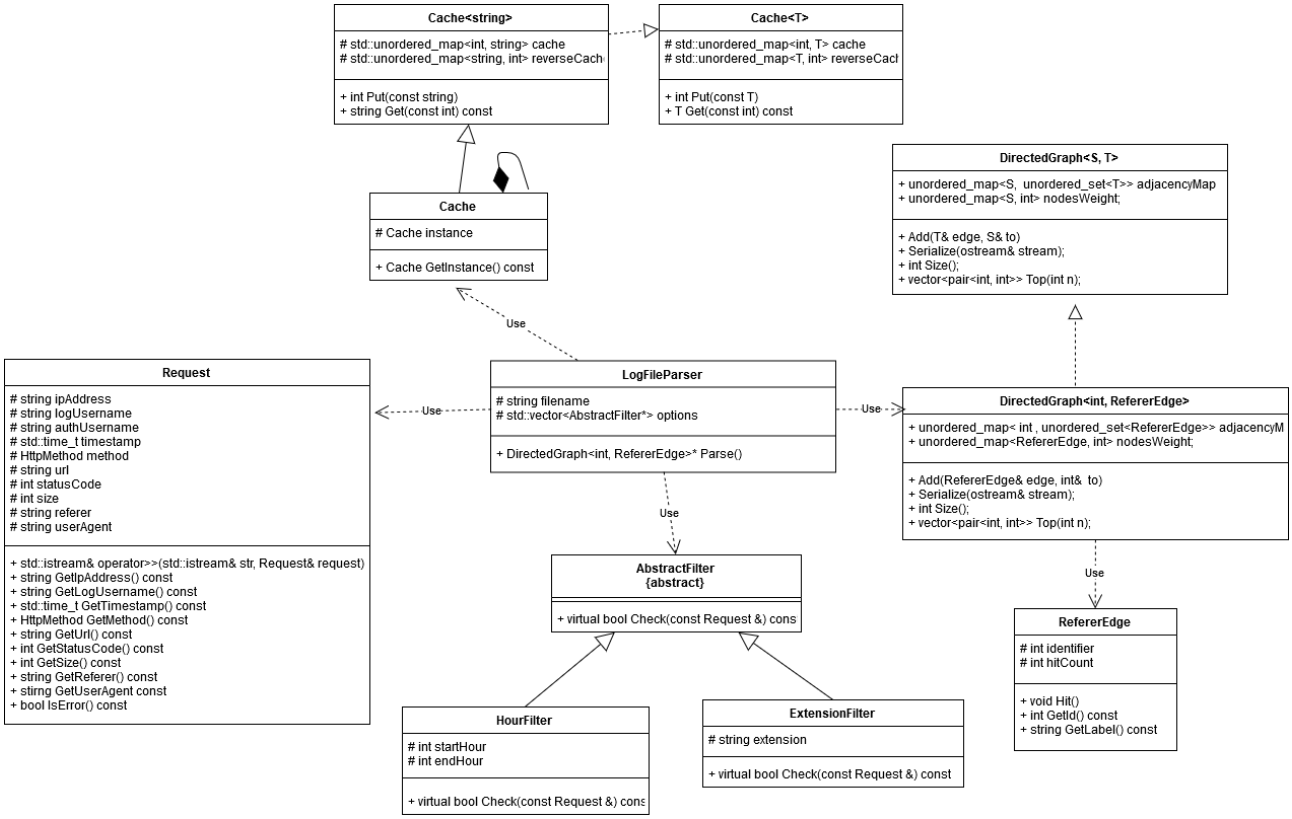
Cas normaux:

- Pouvoir passer de la structure de données des logs analysées à un graphe GraphViz bien formé **(Test 16)**

Cas limites :

- Pas de requêtes à exporter : générer un graphe vide **(Test 3)**

III. Architecture Globale de l'Application



- *Request*:

Chaque ligne de log est parsée dans un objet de type Request. Il contient tous les attributs correspondants ainsi que les opérateurs surchargés pour la lecture et écriture. Ceux-ci permettent notamment de parser une requête correctement formattée (format type Apache/Nginx) depuis n'importe quel flux d'entrée, et d'écrire dans ce même format sur n'importe quel flux de sortie.

- *LogFileParser*:

Cette classe fait la lecture du fichier en appliquant les options demandées et les filtres.

- *HourFilter* et *ExtensionFilter*:

Filtres implémentés à partir de la classe abstraite AbstractFilter. HourFilter peut être utilisé pour faire un filtre de [heure, heure+1[(défaut) ou [heureDebut, heureFin[. ExtensionFilter peut être utilisé pour filtrer n'importe quelle extension. Nous avons choisi de laisser les extensions en paramètre pour pouvoir réutiliser les classes.

- *Caches*:

Afin de gagner en espace mémoire, chaque string URI/referer est associée à un identifiant entier unique. Ainsi l'application manipule principalement des entiers qui sont moins coûteux en mémoire et ne récupère la valeur de la chaîne de caractères uniquement si nécessaires (affichage ou export).

- *RefererEdge*

Cette classe représente un arc du graphe (ie. les données associées au referer). Elle contient l'identifiant du referer et le nombre de hits associés.

- *DirectedGraph*:

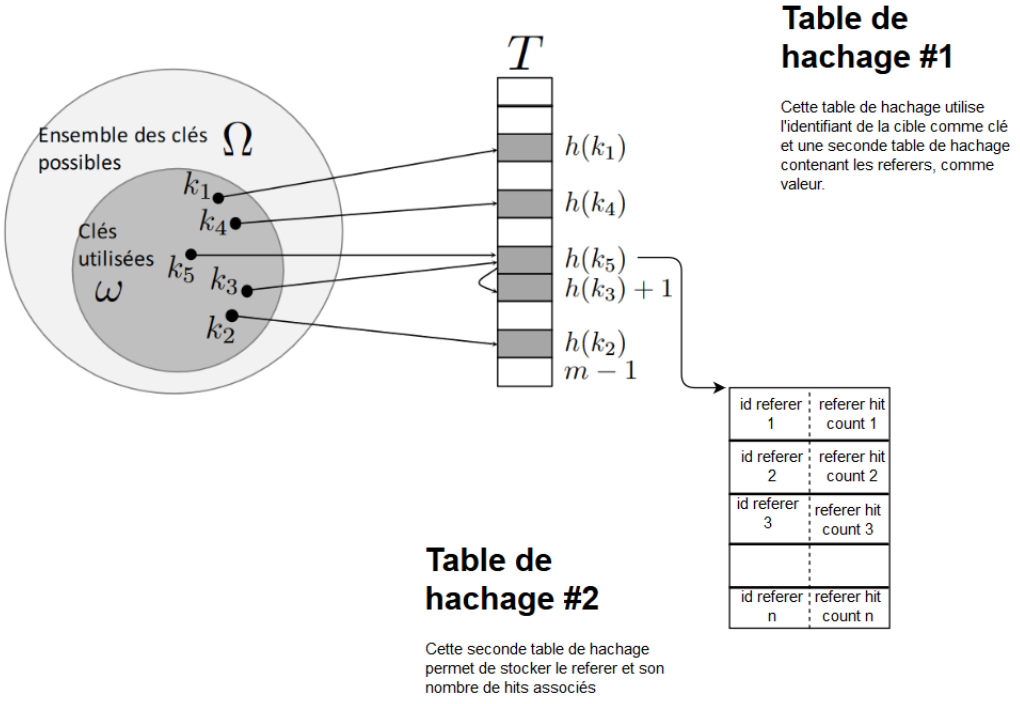
Il s'agit de l'abstraction d'un graphe orienté générique utilisable pour deux classes (S, T). S représente un nœud et T un arc. Chaque nœud est associé à une collection (set) d'arcs dont il est la destination.

IV. Structure de Données

Pour concevoir notre structure de données, nous avons auparavant réfléchi aux opérations les plus utilisées. Dans notre cas, nous avions un besoin intensif d'insertion (fichier potentiellement volumineux, une insertion par ligne différente) et d'un lookup rapide (vérifier si une URL est déjà présente pour tenir à jour sa popularité).

Pour le lookup, nous avons décidé de prendre un conteneur associatif en prenant la cible (ou dans notre cas, un identifiant entier unique de la cible) comme clé. Ceci nous élimine déjà quelques possibilités comme des listes, listes chaînées, piles et files etc. Notre choix s'est finalement porté sur une table de hachage, qui effectue les opérations mentionnées en temps constant (O(1)). Plus particulièrement, une table de hachage en adressage ouvert serait particulièrement adaptée : elle serait en effet plus rapide et moins coûteuse en mémoire que l'adressage fermée, et nous n'utiliserons pas la suppression donc nos performances seront stables.

Notre structure de données principale est ainsi composée de la manière suivante (tous les objets décomposés sont des entiers) :



En parallèle, une autre table de hachage a pour rôle de maintenir le compte du nombre de hits. La clé est encore une fois l'identifiant de la cible, et sa valeur le nombre de hits associés. Une fois le fichier complétement ingéré, nous pouvons ensuite copier le contenu de cette table dans un tableau afin d'effectuer un tri

pour établir le top 10. Cela nous permet à la fois de conserver la rapidité de la table de hachage pour l'insertion et le lookup, et de bénéficier des algorithmes de tri rapides dans les tableaux.

Enfin, chaque chaîne de caractères est insérée dans un cache composé de deux tables de hachages (identifiant->chaîne) et (chaîne->identifiant), le choix de la structure s'étant fait suivant le même raisonnement que pour la structure principale (insertion et lookup en temps constant).

VII. Conclusion

En conclusion, ce TP fût une première plongée très enrichissante dans la STL. Le sujet du TP a réussi à soulever de nombreuses problématiques de conception, et la recherche du meilleur compromis performance/mémoire/réutilisabilité fût à la fois difficile et très intéressant.

VIII. Annexes

ANALOG(1) Manuel de l'utilisateur Linux ANALOG (1)

NOM

analog - Analyser un fichier de log au format Apache.

SYNOPSIS

analog [-e] [-g outputFile.dot] [-h] [-t hour] inputFile.log

DESCRIPTION

Cette page de manuel documente la version GNU de analog.

analog affiche sur la sortie standard les 10 documents les plus consultés par les visiteurs du fichier de log d'entrée.

OPTIONS

-e

Exclure les requêtes correspondants à des fichiers JS, CSS ou images.

-h Afficher l'aide d'utilisation.

-t hour

Exclure les requêtes situées dans la plage horaire [hour, hour+1[.

-g outputFile.dot

Exporter le résultat de l'analyse (graphe de visite) au format GraphViz dans le fichier de sortie outputFile.dot

.