TP3 C++

HAFEDH Jebalia

SOTIR Paul-Emmanuel Binôme B3311

# INTRODUCTION

Le but de ce TP est d’apprendre à utiliser au mieux le code existant, ici, la librairie standard, mais aussi de savoir créer du code qui soit réellement réutilisable dans d’autres projets. Ce TP portant sur l’analyse de logs Apache, c’est également l’occasion d’apprendre à « parser » un fichier et créer des graphes.

# CHOIX GENERAUX

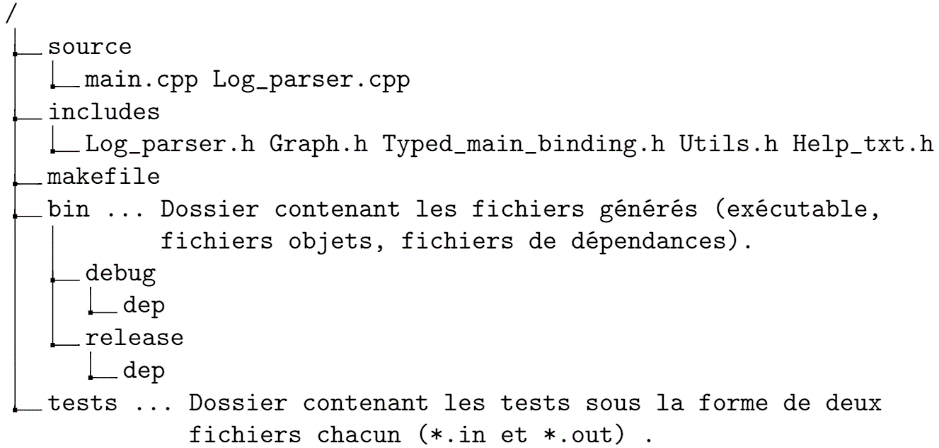
## Guide de style

Notre guide de style se base sur le guide de style officiel du C++ moderne, disponible sur GitHub (<https://github.com/isocpp/CppCoreGuidelines>), auquel nous avons ajouté certains choix comme par exemple les commentaires au format doxygen et le guide de style de l’INSA (parfois incompatibles, par exemple, les classes suivent la règle de 0 : pas de destructeurs).

Architecture du projet

Travaillant sur Visual Studio, nous avons créé un makefile générique gérant les dépendances automatiquement pour pouvoir facilement compiler le projet avec gcc. Le makefile crée des fichiers de dépendance créant à leur tour de nouvelles règles pour ce makefile.

Le projet est organisé de manière à isoler les headers, les fichiers sources et les fichiers générés :



## Options disponibles et tests

Comme définit dans le sujet du TP, le programme prend en paramètre les options ‘-e’, ‘-g’ et ‘-t’ mais certaines précision sont à apporter :

Par exemple les options on plusieurs tags valides (voir main.cpp ou manuel d’utilisation) (validé par le test 6 avec ‘--listCount’).

Si l’option ‘–g’ est utilisé seul le graphe est produit (la validité des graphes est vérifiée par les tests 1, 3, 4, 5).

L’option ‘-e’ exclue du résultat les URLs vers des fichiers dont l’extension est parmi les extensions listées dans la variable statique ‘EXCLUDED\_EXTENTIONS’ (dans le fichier ‘Log\_parser.cpp’) (vérifié par les tests 5 et 6).

De plus l’option ‘-t’ prend en compte les fuseaux horaires : on récupère le fuseau horaire dans le timestamp des lignes de log pour que le paramètre donné à cette option correspond à une heure GMT +0000. (ce comportement est vérifié dans le test 4)

Enfin, le programme accepte d’autre options comme ‘-l’ qui permet de spécifier la taille maximale de la liste des URLs les plus demandés (10 par defaut) (utilisée dans le test 6). On peut aussi spécifier l’option ‘-h’ pour afficher le manuel d’utilisation (retourné par une fonction constexpr dans le header ‘Help\_txt.h’)

## Classes principales

L’implémentation et la conception de ce TP s’est fait du plus réutilisable vers le plus spécifique :

Etant donné qu’il faut interpréter les arguments argc et argv du main pour pouvoir accepter des options, nous avons commencé par créer une classe template ‘Typed\_main\_binding<Args…>’ pouvant s’occuper de convertir automatiquement argc et arv vers les types pris en paramètres par un « main typé » donné.

Nous avons ensuite implémentés la classe template ‘Graph<T>’ stockant simplement des nœuds contenant une valeur de type T et des liens entre ces nœuds.

Enfin, nous avons créés une classe ‘Parse\_log’ créant, à partir du fichier log, un ‘Graph<URL\_t>’ si un graphe est demandé ou une ‘std::unordered\_multimap<URL\_t, unsigned int>’ si la liste des URLs associée à leur nombre d’occurrences a été demandée.

## Regex

Pour extraire des informations d’un fichier log, nous avons choisi d’utiliser les regex (std::regex). Les regex nous permettent non seulement d’extraire les informations qui nous intéressent d’une ligne de log, mais aussi de vérifier sa validité et donc pouvoir garantir que l’on a pas mal interprété une ligne et introduit de mauvaises données dans le résultat.

# SPECIFICATION DES CLASSES ET FONCTIONS PRINCIPALES

## Header Utils.h

Le fichier ‘Utils.h’ contient quelques fonctions basiques qui pourraient être communes à n’importe quels projets C++ (principalement utilisées dans Typed\_main\_binding.h ici).

Nous avons créés une métafonction récursive permettant d’obtenir l’indice d’un type T dans un variadic template ‘Args…’. La récursion se fait sur les types de Args et s’arrête si l’on trouve le type T. L’indice est alors disponible grâce au fait que la structure hérite de ‘std::integral\_constant’ :

// Récursion

template <typename T, typename Tail, typename... Args>

struct index\_of<T, Tail, Args...>

: std::integral\_constant<std::size\_t, 1 + index\_of<T, Args...>::value> { };

Un alias nous permet de faire le chemin inverse, c’est-à-dire, obtenir un type T à partir de son indice dans un variadic template :

template <size\_t N, typename... Args>

using get\_by\_Index\_t = typename std::tuple\_element<N, std::tuple<Args...>>::type;

Le header ‘Utils.h’ contient également la fonction template ‘TP3::parse<T,bool>’. C’est une fonction permettant d’obtenir un type T à partir d’une ‘std::string’. Une surcharge pour chaque types numériques a été faite en utilisant les fonctions ‘std::stoi’, ‘std::stol’, ‘std::stof’, … Le second paramètre template est un booléen indiquant si l’on doit vérifier que la chaine de caractères ne contient rien d’autre (par exemple si "58foo" est considéré comme un int valide). Enfin, si le type T n’est pas numérique, et que l’utilisateur n’a pas fourni de surcharges, un static\_cast<T> sera fait :

template<typename T, bool strong\_convertion = true>

inline T parse(const std::string& str)

{ return static\_cast<T>(str); }

## Classe Typed\_main\_binding<Args…>

La grosse majorité du travail a été pour la classe template ‘Typed\_main\_binding<Args…>’, ce qui est une bonne chose car c’est aussi la plus réutilisable.

La classe ‘Typed\_main\_binding<Args…>’ permet de simplifier grandement l'interprétation de argv et argc grâce aux variadic templates et au principe SFINAE (Substitution Failure Is Not An Error). A partir d'une fonction (que l’on appellera ici « main typé ») dont la signature est de la forme ‘void(std::experimental::optional<Args>...)' (avec les types Args... validant des concepts décrits plus loin), la classe template en déduit statiquement la forme des commandes acceptables et gère automatiquement toutes les vérifications et conversions nécessaires au runtime pour pouvoir appeler, si possible, le main typé spécifié.

Concrètement, il suffit de créer un type pour chaque options validant certains concepts et de créer un « main typé » prenant ces options en paramètres et la classe ‘Typed\_main\_binding<Args…>’ feras tout le reste :

//! Exemple de main typé appellé par un 'Typed\_main\_binding<hour\_option, help\_option>'

static void typed\_main(optional<hour\_option> hour\_opt, optional<help\_option> help) {

if (help)

std::cout << "Help message... “;

else if (hour\_opt)

auto hour = hour\_opt->value;

// ...

}

int main(int argc, char\* argv[]) {

auto main\_binding = TP3::make\_typed\_main\_binding(&typed\_main);

try { main\_binding.exec(argc, argv); }

catch (std::invalid\_argument e) { std::cout << "ERROR: " << e.what() << std::endl; }

return EXIT\_SUCCESS;

}

Puisque qu’il n’est pas obligatoire de spécifier toutes les options lors de l’utilisation du programme, le main typé prend ses paramètres en utilisant le type ‘std::experimental::optional<T>’ (C++ 17). Ce type représente une valeur de type T qui n’est pas forcément définie, tout comme un pointeur peut être à nullptr. L’un des avantages de ce type est que la valeur n’est pas allouée dynamiquement et qu’il peut être casté implicitement en booléen indiquant si la valeur est définie.

Les types des options sont simplement des classes ou structures devant :

Soit contenir une méthode statique permettant d’obtenir une liste de tags (option sans paramètres) :

//! La méthode peut retourner n’importe quelle classe itérable contenant des std::string : ici, tags\_t<3> est un std::array de taille 3)

struct exclusion\_option

{

static tags\_t<3> get\_option\_tags() noexcept { return { "-e", "-E" }; }

};

Soit avoir un attribut publique appelé ‘valeur’ (option obligatoire sans tags) :

//! Il faut que le type de ‘value’ dispose d’une overload de TP3::Parse<T>(std::string)

struct input\_log\_file\_option

{ std::string value; };

Soit les deux :

//! Valide le concept de 'option\_with\_tags' et 'option\_with\_value'

struct hour\_option

{

unsigned short value;

static tags\_t<3> get\_option\_tags() noexcept { return{ "-t", "-T", "--hour" }; }

};

La forme de la commande est déduite de l’ordre et du type des options prises en paramètres par le main typé : par exemple, une options sans tags (comme l’option prenant le fichier log) est obligatoire et impose aux options précédentes dans la signature du main, d’être spécifiées avant dans la commande entrée par l’utilisateur (de même pour les options après l’option sans tags).

Pour accomplir cette classe, nous avons tout d’abords créés des métafonctions (exploitants le principe SFINAE) permettant de vérifier si un type T donné contient un attribut ‘value’ ou une méthode ‘get\_option\_tags()’. Ces fonctions sont publiques car l’utilisateur de la classe pourrait vouloir vérifier si un type T valide les concepts de 'option\_with\_tags' ou 'option\_with\_value'.

Ensuite dans la fonction ‘Typed\_main\_binding<Args...>::exec(int argc, char\* argv[])’ on vérifie d’abord la validité de la commande et on cherche l’emplacement des différentes options (on peur itérer sur les types Args... en construisant des tableau contenant les informations nécessaires sur ces type : tags et type d’option). Une fois l’emplacement des options trouvées on peut convertir les chaines de caractère vers les paramètres et appeler le main typé :

m\_typed\_main\_func((get\_opt<Args>(argc, argv))...);

Avec ‘get\_opt<T>’ une fonction privée ayant une surcharge pour chacun des trois types d’options (sans tags, avec tags et avec tags et paramètre). Cette fonction parse l’éventuel paramètre et retourne ‘std::experimental::nullopt’ si l’option n’est pas spécifiée ou la valeur de l’option si elle est utilisée dans la commande.

## Classe Graph<T>

La classe Graph<T> permet de stocker des nœuds nécessairement liés par des liens orientés. Les nœuds contiennent une valeur de type T et les liens ont un entier indiquant leur nombre d’occurrences.

Nous avons choisi de contenir les nœuds dans une ‘std::unordered\_map<T,node\_id\_t>’ associant un identifiant à la valeur d’un nœud. Ce conteneur permet des recherches et insertions en temps constant grâce à une table de hachages, cependant le type T doit être par conséquent pourvut d’une surcharge de ‘std::hash<T>’ (les fonctions de hachage, d’égalité et d’allocation peuvent être également spécifiées à la construction d’un Graph<T>).

Les liens sont stockés dans une ‘std::unordered\_multimap<node\_id\_t, Link\_to>’ qui permet la recherche de liens à partir de sa source en temps constant (le type ‘Link\_to’ est simple une structure contenant l’identifiant du nœud de destination et le nombre d’occurrences du lien).

Ces choix concernant le stockage des données dans le graphe permettent à la fois une implémentation des méthodes nécessaires simple, et de bonnes performances en temps et mémoire. Par exemple, la fonction globale permettant de sérialiser un graphe se résout à deux boucles d’une ligne :

template<typename T>

void serialize\_graph(const std::string& output\_filename, std::unique\_ptr<Graph<T>> graph);

La classe Graph<T> contient une méthode ‘void add\_link(T&& sour, T&& dest)’ ajoutant un lien (et ses nœuds si nécessaire) au graphe. Ses paramètres sont des « universal references », c’est-à-dire des r-value references templates qui permettent de minimiser le nombre de copies des paramètres en une seule implémentation (au lieu de faire une surcharge dédiée aux r-values et une autre aux l-values).

Une autre méthode ‘void clear()’ vide le graphe. Et deux autres méthodes permettent d’accéder directement au conteneur des nœuds ou des liens (ne vas pas à l’encontre de l’encapsulation car Graph<T> est un type simple qui pourrait être considéré comme l’addition de deux conteneurs).

## Classe Log\_parser

La classe Log\_parser construit soit un ‘Graph<URL\_t>’ si un graphe est demandé ou une ‘std::unordered\_multimap<URL\_t, unsigned int>’ si la liste des URLs associée à leur nombre d’occurrences est demandée :

std::unique\_ptr<Graph<URL\_t>> parse\_graph(const std::string& log\_file\_name) const;

std::unique\_ptr<urls\_scores\_t> parse\_urllist(const std::string& log\_file\_name) const;

La classe contient également les quatre méthodes suivantes pour spécifier s’il faut exclure les images ou filtrer les résultats pour une heure donnée :

void enable\_hour\_filter(hour\_t hour);

void disable\_hour\_filter();

void enable\_exclusion() noexcept;

void disable\_exclusion() noexcept;

La lecture du fichier log apache se fait principalement grâce à des regex. Nous avons construit un regex pour chaque partie constituant une ligne de log apache. Nous nous sommes inspirés du regex de stephenhay (<https://mathiasbynens.be/demo/url-regex>) pour matcher les URLs avec fiabilité et performance. Ces regex forment ensuite un seul regex qui match toute une ligne de log correspondant à une requête GET ayant un statut à 200. Les sous-matchs de ce regex correspondent aux informations que l’on veut extraire : l’heure, le fuseau horaire, l’URL du document demandé, l’URL source et la présence de ‘<http://intranet-if.insa-lyon.fr>’ (ou https://... ou ftp://...). Si l’option ‘–e’ a été spécifiée (appel à la méthode enable\_exclusion()) on utilise un regex légèrement différent qui exclue les fichiers image, les fichiers css ou javascript.

# CONCLUSION ET AMELIORATIONS POSSIBLES

Nous avons appris à créer du code réutilisable en évitant de tomber dans des pièges tels que le fait de se dire « si j’utilise des templates, alors mon code seras réutilisable » ou « si je modélise mon problème avec beaucoup de classes différentes, mon code seras plus réutilisable ».

Nous avons également appris à créer des regex et exploiter toutes les possibilités des templates pour faire des classes dont l’utilisation en est grandement améliorée sans gros coût au runtime (Typed\_main\_binding).

Certaines améliorations, comme aller plus loin dans l’optimisation des regex sans perdre de leur qualité ou traiter le fichier log en multithread, pourraient améliorer les performances.

Nous pourrions également compléter la classe ‘Typed\_main\_binding ‘ pour, par exemple, autoriser les options prenant un tableau de paramètres (simplement une surcharge de get\_opt<T> à ajouter) ou permettre au main typé de prendre les paramètres dont le type est une option obligatoire sans optional<Param\_t> :

// input\_log is a mandatory tagless parameter

void typed\_main( input\_log\_file\_option input\_log,

optional<output\_graph\_file\_option> output\_graph,

optional<hour\_option> hour\_opt /\*, ...\*/) { /\*...\*/ }

Enfin, on pourrait même ajouter à cette classe la génération au compile time de la description de la commande sur le même modèle que les documentations de la commande ‘man’ (les descriptions des options seraient, comme pour les tags, disponibles dans leurs types respectifs).