|  |
| --- |
|  |
| Références croisées sur un ensemble de fichiers C++ |
| Présentation de la spécification et de la conception |
| **Binôme B3302** |
| Robin Gicquel  Arnaud Mery de Montigny |
| **Le 9 décembre 2011** |

1. **Spécification complète de l’application**

* La ligne de commande pour exécuter l’application doit être exactement de la forme d’une de ces quatre commandes :
* *TpReferencesCroisees.exe -e -k fichier\_mots\_cles [nomfichier]\**
* *TpReferencesCroisees.exe -e [nomfichier]\**
* *TpReferencesCroisees.exe -k fichier\_mots\_cles [nomfichier]\**
* *TpReferencesCroisees.exe [nomfichier]\**
* Lorsque l’on écrit des arguments syntaxiquement invalides ou mal positionnés, le programme envoie un message d’erreur spécifique et s’arrête.
* On considère que le premier numéro de ligne d'un fichier porte l'indice 1.
* Dans le cas où un même fichier est présent plusieurs fois dans les arguments, l'application ne réalise le traitement qu'une seule fois.
* Dans le cas où un identificateur est trouvé plusieurs fois sur une même ligne d’un même fichier, le numéro de cette ligne pour ce fichier apparait autant de fois que l’identificateur a été rencontré sur la ligne.

Ex (pour un identificateur présent deux fois sur la 3ème ligne) :

*cout file.cpp 3 3*

* Si le fichier fichier\_mots\_clés est vide, on considère qu’aucun mot-clé ne doit être pris en compte.
* Les commentaires seront toujours pris en compte selon la syntaxe du C++, c’est-à-dire les caractères situés après « // » au sein d’une même ligne ainsi que tous les caractères situés entre « /\* » et « \*/ ».
* On exclut d’analyser les chaînes de caractères ainsi que les caractères situés entre « ‘ » et « " » (simple et double guillemets).
* L’application considère que chaque ligne du fichier fichier\_mots\_cles ne doit comporter qu’un seul mot-clé. S’il y a présence des symboles suivants dans le fichier, le programme envoie un message d’erreur et s’arrête : *l’espace, la tabulation, la virgule, le point-virgule*. Si des mots sont séparés par un autre caractère, ils sont considérés comme un unique mot.
* Lorsque l’on écrit un fichier fichier\_mots\_clés qui n’existe pas ou qui ne peut pas permettre la lecture, le programme envoie un message d’erreur et s’arrête.
* Si on ne met aucun nom de fichier à analyser, le programme envoie un message d’erreur et s’arrête.
* Si un ou plusieurs noms de fichiers à analyser sont invalides, le programme envoie un message d’erreur et s’arrête sans donner de résultat sur les fichiers qui auraient pu être analysés.

1. **Test fonctionnels**

Pour certains tests, nous aurons besoin de fichiers fichier\_mots\_cles et de fichiers à analyser. Nous avons donc choisi plusieurs fichiers de test :

**keywords.txt:**

int

world

template

**keywords2.txt** est vide.

**keywords3.txt:**

int main();int entier;

**keywords4.txt :**

//commentaire

int a;

/\*voici

un autre commentaire\*/

char b = 'a';

string c = "Hello";

fileTest1.txt:

// affiche le message "Hello world"

int main() {

cout<<"Hello world"<<endl;

cout<<endl;

return 0;

}

**fileTest2.txt:**

int main();

**fileTest3.txt:**

int main();int entier;

**Plan de test :**

Test 1.1 : On positionne l’option « -e » à la fin de la commande

**Résultat attendu** : Affiche ce message : *Position de l'argument -e invalide*

Test 1.2 : On positionne l’option « -e » juste avant le premier fichier à analyser

**Résultat attendu** : Affiche ce message : *Position de l'argument -e invalide*

Test 1.3 : On positionne l’option « -e » juste après l’option « -k »

**Résultat attendu** : Affiche ce message : *FichierMotCles attendu après l'option –k*

Test 1.4 : On positionne l’option –k après fichier\_mot\_cles

**Résultat attendu** : Affiche ce message : *Position de l'argument -k invalide*

Test 1.5 : On exécute la commande sans mettre d’arguments

**Résultat attendu** : Affiche ce message : *Nombre d'arguments insuffisant*

Test 1.6 : On exécute la commande sans indiquer de fichier à analyser

**Résultat attendu** : Nombre d'arguments insuffisant

Test 1.7 : On exécute la commande en mettant une option inconnue

**Résultat attendu** : Affiche ce message : *Option inconnue*

Test 2 : On exécute la commande suivante pour vérifier que le fichier fileTest2.txt n’est traité qu’une seule fois :

*TpReferencesCroisees.exe fileTest2.txt fileTest2.txt*

**Résultat attendu** : Affiche ce message : *int fileTest2.txt 1*

Test 3 : On exécute la commande suivante :

*TpReferencesCroisees.exe fileTest3.txt*

**Résultat attendu** : Affiche ce message : *int fileTest3.txt 1 1*

Test 4 : On exécute la commande en indiquant comme fichier fichier\_mots\_cles un fichier vide

**Résultat attendu** : (rien ne sort de la sortie standard)

Test 5 : On exécute la commande en indiquant comme fichier à analyser un fichier comportant des « // », des « /\* » avec des « \*/ », des « ‘’ » et « ‘ »

**Résultat attendu** : (les caractères situés derrière « // » au sein d’une ligne et tous les caractères situés entre « /\* » et « \*/ » ne sont pas affichés comme indicateur. Il en va de même pour les chaines de caractères ainsi que les caractères situés entre « ‘ ».

Test 6 : On exécute la commande en indiquant comme fichier fichier\_mots\_cles un fichier comportant des tabulations, des espaces et des virgules

**Résultat attendu** : Affiche ce message : *Erreur de syntaxe dans le fichier de mots-clés*

Test 7 : On exécute la commande en indiquant un fichier fichier\_mots\_clés qui n’existe pas

**Résultat attendu** : Affiche ce message : *Erreur lors de l'ouverture du fichier de mots-clés*

Test 8 : On exécute la commande en indiquant plusieurs fichiers à analyser, mais avec un fichier invalide

**Résultat attendu** : Affiche ce message : *Erreur lors de l'ouverture d'un fichier source*

Test 9 : On exécute la commande : -e -k keywords.txt fileTest1.txt fileTest2.txt

**Résultat attendu** : Affiche ce message :

cout fileTest1.txt 3 4

endl fileTest1.txt 3 4

main fileTest1.txt 2 fileTest2.txt 1

return fileTest1.txt 5

Test 10 : On exécute la commande : -e fileTest1.txt fileTest2.txt

**Résultat attendu** : Affiche ce message :

cout fileTest1.txt 3 4

endl fileTest1.txt 3 4

main fileTest1.txt 2 fileTest2.txt 1

Test 11 : On exécute la commande : -k keywords.txt fileTest1.txt fileTest2.txt

**Résultat attendu** : Affiche ce message :

int fileTest1.txt 2 fileTest2.txt 1

Test 12 : On exécute la commande : fileTest1.txt fileTest2.txt

**Résultat attendu** : Affiche ce message :

int fileTest1.txt 2 fileTest2.txt 1

return fileTest1.txt 5

1. **Architecture générale de l’application**



Pour avoir une vue plus claire de l’architecture de notre programme, nous avons décidé de ne pas inclure les paramètres des méthodes.

1. **Algorithmes principaux**

Nous avons choisi de présenter ici l’algorithme permettant d’isoler et de sélectionner les mots à traiter dans un fichier source. Les autres opérations réalisées par le programme ne sont que des manipulations des structures de la STL et des tests de validité d’arguments.

On suppose que l'on dispose des fonctions suivantes :

RenvoyerErreur() : affiche un message d'erreur et termine le programme

LireCaractere() : renvoie un caractère du fichier et passe au suivant

ConsulterCaractere() : renvoie un caractère mais ne passe pas au suivant

Le rôle des autres fonctions utilisées peut être facilement déduit de leur nom.

On utilise la syntaxe du C/C++ pour les caractères, les négations, les affectations et les comparaisons.

Fonction LireFichier (nomFichier : chaîne)

Variables

caractère c

caractère c2

caractère carAttendu1

caractère carAttendu2

chaîne mot

entier numLigne

Début

si ( ! FichierEstLisible() ) alors

RenvoyerErreur()

fsi

numLigne = 1

carAttendu1 = -1

carAttendu2 = -1

mot = ""

tant que ( LectureNonTerminée() ) faire

c = LireCaractere()

si ( carAttendu == -1 ) alors

si ( c == '/' ) alors

c2 = ConsulterCaractere()

si ( c2 == '/' ) alors

carAttendu1 = '\n'

LireCaractere()

sinon si ( c2 == '\*' ) alors

carAttendu1 = '\*'

carAttendu2 = '/'

fsi

sinon si ( c == '"' ) alors

carAttendu1 = '"'

sinon si ( c == '\'' ) alors

carAttendu1 = '\''

sinon si ( ! EstCaractereSeparation( c ) ) alors

Concatener( mot, c )

sinon

si ( mot != "" ) alors

TraiterMot( mot, nomFichier, numLigne)

mot = ""

fsi

fsi

sinon

si ( c == carAttendu1 ) alors

si ( carAttendu2 == -1 ) alors

carAttendu1 = -1

sinon

si ( ConsulterCaractere() == carAttendu2 ) alors

carAttendu1 = -1

carAttendu2 = -1

si ( LireCaractere() == '\n' ) alors

numLigne = numLigne + 1

fsi

fsi

fsi

fsi

fsi

si ( ConsulterCaractere() == '\n' ) alors

numLigne = numLigne + 1

fsi

ftq

Fin Fonction

Fonction TraiterMot ( mot : chaîne, nomFichier : chaîne, numLigne : entier )

si ( ! EstNombreEntier( mot ) ) alors

si ( EstMotCle ( mot ) ) alors

si ( ! exclureMotsCles ) alors

AjouterReference ( mot, nomFichier, numLigne )

fsi

sinon

si ( exclureMotsCles ) alors

AjouterReference (mot, nomFichier, numLigne )

fsi

fsi

fsi

Fin Fonction

1. **Analyse critique**

Nous avons choisi de représenter la structure des références comme celle des fichiers par un arbre. La structure des lignes est un tableau

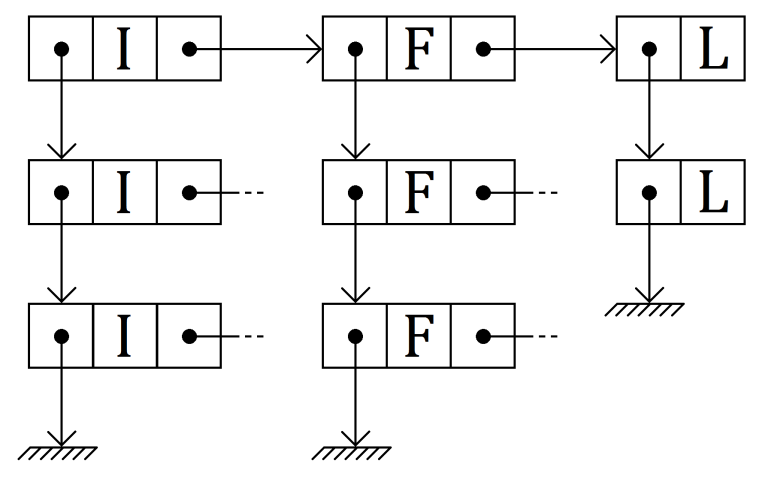


Figure 1: Double liste chainée

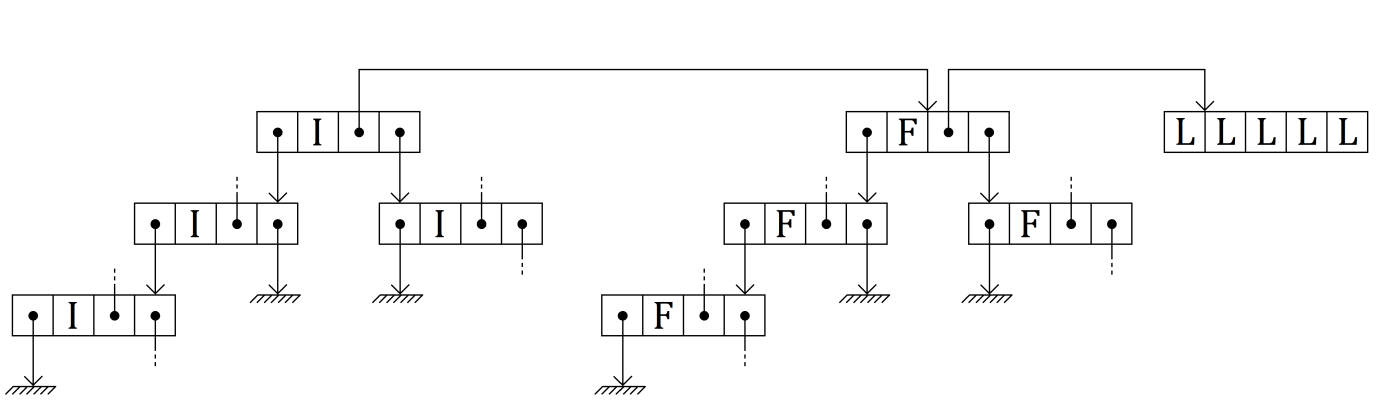


Figure 2:Double arbre

Comme on le voit grâce au schéma ci-dessus, une insertion d’un élément dans une liste chainée doit se faire en parcourant le début. Au pire la complexité est de O(n).

Si nous prenons par contre un arbre, l’insertion d’un élément se fera en suivant une seule branche, l‘insertion ne se fera donc jamais en parcourant tous les éléments. Au pire la complexité est de O(log(n)).

Le défaut de l’arbre est comme vous le voyiez est qu’il prend plus de place que la liste chainée.

L’arbre a un autre petit défaut. Il est un peu plus long à parcourir du début à la fin qu’une liste chainée. Mais comme on ne parcourt qu’une seule fois la structure pour pouvoir l’afficher dans la sortie standard et que l’on opère de nombreuses insertions, ce petit défaut est négligeable.

Le tableau utilisé pour les lignes est tout à fait justifiable puisque l’ajout des numéros de lignes se fait sans condition.

Par contre, il aurait été plus judicieux de mettre une structure de liste chainée pour les noms de fichiers. L’ajout d’un fichier se fera toujours en condition du dernier élément ajouté dans la structure pour une liste chainée. Si le nom du dernier élément est celui du nom qu’on veut insérer, on ne l’ajoute pas. Sinon on l’ajoute. On ne doit pas comparer avec les autres éléments car un fichier qui déjà traité ne pourra pas l’être une deuxième fois. Cela permet d’affirmer que les noms présents dans la liste chainée ne peut pas être le même que celui que l’on veut ajouter sauf si le nom du fichier vient juste d’être inséré. Ayant vu cette optimisation trop tard, nous n’avons pu modifier le code.