Thibaut Patel 3IF3 - 2010/2011
Stanislas Signoud **Binôme B3310**

Document de Réalisation

TP-AC – Références croisées

Table des matières

Document de Réalisation....1
Choix généraux de réalisation....2
Utilisation de la STL....2
Classe Index : Parcours de la structure de données....2
Listing des sources....3
LeMain.ccp....3
Main.h....3
Main.cpp....5
Analyseur.h....9
Analyseur.cpp....11
Parseur.h....13
Parseur.cpp....15
Index.h....17
Index.cpp....21
ListeMotCles.h....24
ListeMotCles.cpp....26

Choix généraux de réalisation

Utilisation de la STL

Pour implémenter les structures de données définies dans le document de spécification et de conception, nous avons choisi d'utiliser la librairie standard de templates (STL) qui propose nombre de conteneurs adaptés à nos besoins.

Ainsi, les listes chainées sont implementées à l'aide de std::list, les paires par std::pair, et les arbres binaires équilibrés par std::map.

Classe Index : Parcours de la structure de données

Pour parcourir la structure de données complexe des références croisées de façon simple pour l'utilisateur de la classe Index, des méthodes `HasNext`*`Xxx`* et `Next`*`Xxx`* ont été créées dans Index.

Ces méthodes utilisent des itérateurs, `it`*`Xxx`* et `cur`*`Xxx`*, qui respectivement représente la position suivante et courante de l'itérateur, pour permettre de faire un test rapide pour les méthodes `HasNext`*`Xxx`*.

Lorsque l'arbre est modifié, les itérateurs ont un risque de ne plus être valable : un booléen areItValid est alors mis à `false`. Les méthodes `HasNext` vérifie avant l'utilisation des itérateurs que ce booléen soit bien à `true`, et si ce n'est pas le cas, les itérateurs sont réinitialisés à l'aide de la méthode protégée `resetIterators`.

Listing des sources

LeMain.ccp

```
1.  // Point d'entrée minimal du programme.
2.
3.  #include "Main.h"
4.
5.  int main( int argc, const char* argv[] )
6.  {
7.      return Main::Run(argc, argv);
8.  }
```

Main.h

```
1.  /*************************************************************************
2.   Main - Gère les arguments et les éléments du programme
3.   -------------------
4.   début                : 19 nov. 2010
5.   copyright            : (C) 2010 par ssignoud et patel
6.   *************************************************************************/
7.
8.  // Comme autorisé (après demande) par les professeurs en séance de TP,
9.  // ces fichiers sources utilisent la norme d'écriture Doxygen/Javadoc en
10. // lieu et place du « Guide de Style INSA » original.
11. // Ses conseils et ses principes de formatages (retours à la ligne et
12. // informations algorithmiques complémentaires) sont cependant conservés.
13.
14. // ==========[ Interface de la classe <Main> (fichier Main.h) ]============
15.
16. #if ! defined ( MAIN_H_ )
17. #define MAIN_H_
18.
19. //------------------------------------------------- Interfaces utilisées
20. #include "Analyseur.h"
21.
22. /**
23.  * Gère les arguments et les éléments du programme.
24.  *
25.  * La méthode principale de cette classe (Run) analyse les arguments et
26.  * formate la sortie. Ce formatage n'est pas délégué à une autre classe,
27.  * car il est trop spécifique à l'utilitaire pour être réutilisé, tout
28.  * en prenant très peu de lignes de code grâce au report de la complexité
29.  * du parcours de l'index dans la classe Index.
30.  */
31. class Main
32. {
33.     //================================================================ PUBLIC
34.     public:
35.         //------------------------------------------------ Méthodes publiques
36.         /**
37.          * Démarre le programme, analyse les arguments passés par argc et argv,
38.          * effectue la recherche et affiche les résultats, ou les erreurs
39.          * le cas échéant.
40.          * @param argc Le nombre d'arguments de la ligne de commande.
41.          * @param argv Les arguments de la ligne de commande.
42.          * @return Le statut de l'application ; 0 si tout s'est bien passé
```

```
43.           * ou 1 si une erreur fatale s'est produite.
44.           */
45.          static int Run(int argc, const char* argv[]);
46.
47.      //============================================================== PRIVÉ
48.      protected:
49.          //------------------------------------------------ Méthodes protégées
50.          /**
51.           * Affiche sur la sortie d'erreur standard une chaîne d'explication
52.           * de l'usage du programme, à-la-UNIX.
53.           *
54.           * @param nomprogramme Le nom du programme (typiquement argv[0]).
55.           */
56.          static void printUsage(const char* nomprogramme);
57.
58.
59.
60.  };
61.
62.  #endif // MAIN_H_
```

Main.cpp

```
1.  /*************************************************************************
2.   Main  -  Gère les arguments et les éléments du programme
3.   -------------------
4.   début                : 19 nov. 2010
5.   copyright            : (C) 2010 par ssignoud et tpatel
6.   *************************************************************************/
7.
8.  // Comme autorisé (après demande) par les professeurs en séance de TP,
9.  // ces fichiers sources utilisent la norme d'écriture Doxygen/Javadoc en
10. // lieu et place du « Guide de Style INSA » original.
11. // Ses conseils et ses principes de formatages (retours à la ligne et
12. // informations algorithmiques complémentaires) sont cependant conservés.
13.
14. // Vous recherchez le point d'entrée du programme ?
15. // Il a été séparé de ce fichier et se trouve dans LeMain.cpp, pour plus
16. // de clareté, bien que son contenu n'aie pas grand intérêt.
17.
18. // ========[ Réalisation de la classe <Main> (fichier Main.cpp) ]========
19.
20. //================================================================ INCLUDE
21.
22. //--------------------------------------------------- Include système
23. using namespace std;
24. #include <string>
25. #include <set>
26. #include <fstream>
27. #include <iostream>
28. #include <vector>
29.
30. //------------------------------------------------------ Include personnel
31.
32. #include "Main.h"
33. #include "Parseur.h"
34.
35. //================================================================ PUBLIC
36.
37. /**
38.  * @algorithm La gestion des arguments et des erreurs se fait au fur et à mesure de
39.  * l'analyse de la ligne de commande, qui se fait séquentiellement.
40.  * L'affiche ne pose pas de problème en soit, en utilisant les méthodes pratiques
41.  * proposées par la classe Index.
42.  */
43. int Main::Run(int argc, const char* argv[])
44. {
45.     // Contient le paramètre actuellement analysé
46.     int paramCounter = 1;
47.
48.     // Doit-on exclure ou non les mots clés indiqués ?
49.     bool exclusion = true;
50.
51.     if(argc - 1 == 0)
52.     {
53.         // Aucune action demandée ? Rappelons le mode d'emploi...
54.         printUsage(argv[0]);
55.         return 1;
56.     }
57.
58.     //Contient l'argument courant
59.     string tmpArg = argv[paramCounter];
60.
```

```
61.      if(tmpArg      == "-e")
62.      {
63.          exclusion = false;
64.          paramCounter++;
65.      }
66.
67.      if(argc-paramCounter == 0)
68.      {
69.          cerr << "E: Aucun fichier à analyser." << endl;
70.          printUsage(argv[0]);
71.          return 2;
72.      }
73.
74.      //Contient la liste des mots clés
75.      ListeMotCles listeMotCles;
76.
77.      //Contient l'argument courant
78.      tmpArg = argv[paramCounter];
79.
80.      if(tmpArg == "-k")
81.      {
82.          paramCounter++;
83.          if(argc-paramCounter == 0)
84.          {
85.              cerr << "E: Aucun fichier de mot clé à analyser." << endl;
86.              printUsage(argv[0]);
87.              return 3;
88.          }
89.
90.          // Contient l'argument courant
91.          tmpArg = argv[paramCounter];
92.
93.          // On teste si le fichier existe...
94.          ifstream* fMotCles = new ifstream(tmpArg.c_str());
95.          if(!fMotCles)
96.          {
97.              cerr << "E: Le fichier de mots clés n'est pas accessible ou n'existe pas." << endl;
98.              return 4;
99.          }
100.         delete fMotCles; // On le supprime : il n'a été créé que tester son accessiblité
101.
102.         // Le fichier avec le nom tmpArg existe, donc on peut créer un parseur pour ce
    fichier...
103.
104.         Parseur parseur(tmpArg);
105.
106.         // ...et l'utiliser
107.         string motcle = parseur.NextIdent()->first;
108.         while(motcle != "")
109.         {
110.             listeMotCles.AddMotCle(motcle);
111.             motcle = parseur.NextIdent()->first;
112.         }
113.
114.         if(listeMotCles.IsVide())
115.         {
116.             cerr << "E: Le fichier de mots clés n'est pas valide." << endl;
117.             return 5;
118.         }
119.
120.         paramCounter++;
121.
122.     }
123.     else
```

```
124.      {
125.          // Création de la liste des mot clés du C++ par défaut
126.          // Cette liste est incorporée pour augmenter la portabilité de l'application
127.          // (pas besoin de fichiers annexes).
128.          const int NB_MOTCLES = 63;
129.          string mots[] = {"asm", "auto", "bool", "break", "case", "catch", "char",
130.                           "class", "const", "const_cast", "continue", "default",
131.                           "delete", "do", "double", "dynamic_cast", "else", "enum",
132.                           "explicit", "export", "extern", "false", "float", "for",
133.                           "friend", "goto", "if", "inline", "int", "long", "mutable",
134.                           "namespace", "new", "operator", "private", "protected",
135.                           "public", "register", "reinterpret_cast", "return",
136.                           "short", "signed", "sizeof", "static", "static_cast",
137.                           "struct", "switch", "template", "this", "throw", "true",
138.                           "try", "typedef", "typeid", "typename", "union", "unsigned",
139.                           "using", "virtual", "void", "volatile", "wchar_t", "while"};
140.          for(int i=0; i<NB_MOTCLES; i++)
141.          {
142.              listeMotCles.AddMotCle(mots[i]);
143.          }
144.      }
145.
146.      if(argc - paramCounter == 0)
147.      {
148.          cerr << "E: La liste des fichiers à analyser est vide.";
149.          return 10;
150.      }
151.
152.      vector<string> listFichiers(argc - paramCounter);
153.
154.      for(int i = paramCounter; i < (int) argc; i++)
155.      {
156.          // On vérifie si les fichiers existent bien, dès maintenant
157.          ifstream unFichier;
158.          unFichier.open(argv[i]);
159.          if(!unFichier) {
160.              cerr << "E: Le fichier " << argv[i]
161.                   << " est inconnu ou inaccessible en lecture." << endl;
162.              return 11;
163.          }
164.          listFichiers[i - paramCounter] = argv[i];
165.      }
166.
167.
168.      Analyseur unAnalyseur(listeMotCles, listFichiers, exclusion);
169.
170.      //Lancement de l'analyse des fichiers
171.      Index * lindex = unAnalyseur.Run();
172.
173.      // Si la recherche n'a retourné absolument aucun résultat...
174.      if(!lindex->HasNextIdent()){
175.          cerr << "W: La recherche des mots clés n'a donné aucun résultat." << endl;
176.          return 0; // Ce n'est pas une erreur : code d'erreur nul
177.      }
178.
179.      // Affichage du résultat de la recherche, en moins de quinze lignes ;
180.      // toute la complexité du parcours de l'arbre/la map est située du
181.      // côté de l'Index et est masquée à l'utilisateur de l'objet
182.      while(lindex->HasNextIdent())
183.      {
184.          cout << lindex->NextIdent();
185.
186.          while(lindex->HasNextFile())
187.          {
```

```
188.             cout << "\t" << lindex->NextFile();
189.
190.             while(lindex->HasNextLine())
191.             {
192.                 cout << " " << lindex->NextLine();
193.             }
194.         }
195.         cout << endl;
196.     }
197.
198.     return 0;
199. }
200.
201. //======================================================================== PRIVE
202.
203. //------------------------------------------------------ Méthodes protégées
204.
205. void Main::printUsage(const char * nomprogramme)
206. {
207.     cerr << "Usage: " << nomprogramme
208.          << " [-e] [-k <keyword file>] file1 [file2 ...]" << endl;
209. }
```

Analyseur.h

```
1.  /*************************************************************************
2.   Analyseur - Recherche les références contenues dans les fichiers.
3.   -------------------
4.   début                : 19 nov. 2010
5.   copyright            : (C) 2010 par ssignoud et tpatel
6.   *************************************************************************/
7.
8.  // Comme autorisé (après demande) par les professeurs en séance de TP,
9.  // ces fichiers sources utilisent la norme d'écriture Doxygen/Javadoc en
10. // lieu et place du « Guide de Style INSA » original.
11. // Ses conseils et ses principes de formatages (retours à la ligne et
12. // informations algorithmiques complémentaires) sont cependant conservés.
13.
14. // ======[ Interface de la classe <Analyseur> (fichier Analyseur.h) ]======
15.
16. #if ! defined ( ANALYSEUR_H_ )
17. #define ANALYSEUR_H_
18.
19. //---------------------------------------------------- Interfaces utilisées
20. #include <set>
21. #include <string>
22. #include <vector>
23.
24. #include "ListeMotCles.h"
25. #include "Index.h"
26.
27. /**
28.  * Recherche les références contenues dans les fichiers.
29.  *
30.  * Cette classe effectue le travail demandé par Main, avec les fichiers
31.  * dont les noms sont donnés lors de sa construction. Il effectue ce
32.  * travail en faisant appel à la classe Parseur, et ajoute les occurences
33.  * trouvées à un Index.
34.  */
35. class Analyseur
36. {
37.
38.     public :
39.
40.         //------------------------------------------------- Méthodes publiques
41.
42.         /**
43.          * Lance l'analyseur.
44.          *
45.          * @return Un index prêt à être utilisé.
46.          */
47.         Index * Run();
48.
49.         //------------------------------------------------ Surcharge d'opérateurs
50.
51.         /**
52.          * Affecte les données d'unAnalyseur à l'analyseur.
53.          * @param unAnalyseur Analyseur d'où proviendront les données affectées
54.          * à l'analyseur.
55.          * @return L'analyseur, avec les données d'unAnalyseur.
56.          */
57.         Analyseur & operator =(const Analyseur & unAnalyseur);
58.
59.         //-------------------------------------------- Constructeurs - destructeur
60.
```

```
61.         /**
62.          * Construit un Analyseur.
63.          *
64.          * @param listeMotCles Liste de mots clés à rechercher (ou à ne pas rechercher)
65.          * dans les fichiers.
66.          * @param fichiers Vecteurs de noms de fichiers dans lesquels la recherche
67.          * doit se faire.
68.          * @param exclusion Booléen indiquant si les mots indiqués sont exclus de la
69.          * recherche (true) ou au contraire sont les seuls à être recherchés (false).
70.          */
71.         Analyseur(ListeMotCles listeMotCles, std::vector<std::string> fichiers,
72.                   bool exclusion);
73.
74.         /**
75.          * Construit un Analyseur à partir des données d'un autre (constructeur
76.          * par copie).
77.          * @param unAnalyseur Analyseur à partir duquel les données vont être copiées.
78.          */
79.         Analyseur(const Analyseur & unAnalyseur);
80.
81.         /**
82.          * Détruit l'analyseur.
83.          */
84.         virtual ~Analyseur();
85.
86.     protected:
87.         /**
88.          * Contient les mots clés à rechercher, renseignés lors de la construction
89.          * de l'objet.
90.          */
91.         ListeMotCles motscles;
92.
93.         /**
94.          * Noms des fichiers qui doivent être analysés.
95.          */
96.         std::vector<std::string> nomsFichiers;
97.
98.         /**
99.          * Indique si les mots clés inclus dans motscles sont à exclure, ou au
100.         * contraire s'ils sont les seuls à être retournés.
101.         */
102.        bool motsExclus;
103.
104. };
105.
106. #endif // ANALYSEUR_H_
```

Analyseur.cpp

```cpp
/*************************************************************************
 Analyseur - Recherche les références contenues dans les fichiers.
 -------------------
 début                : 19 nov. 2010
 copyright            : (C) 2010 par ssignoud et tpatel
 *************************************************************************/

// Comme autorisé (après demande) par les professeurs en séance de TP,
// ces fichiers sources utilisent la norme d'écriture Doxygen/Javadoc en
// lieu et place du « Guide de Style INSA » original.
// Ses conseils et ses principes de formatages (retours à la ligne et
// informations algorithmiques complémentaires) sont cependant conservés.

// ====[ Réalisation de la classe <Analyseur> (fichier Analyseur.cpp) ]===

//================================================================ INCLUDE

//---------------------------------------------------- Include système
using namespace std;
#include <iostream>
#include <fstream>

//------------------------------------------------------ Include personnel
#include "Analyseur.h"
#include "ListeMotCles.h"
#include "Index.h"
#include "Parseur.h"

//================================================================ PUBLIC

/**
 * @algorithm Pour tous les fichiers, un Parseur est créé. On boucle
 * tant que ce parseur renvoie un identificateur, et on teste si cet
 * identificateur est un identifiant (ou mot clé). Si c'est le cas
 * ou si ça n'est pas le cas (selon le booléen motsExclus), on l'ajoute
 * à un Index créé en début de méthode. Une fois ces boucles terminées,
 * on renvoie l'index fraîchement rempli.
 */
Index * Analyseur::Run() {
    Index * index = new Index();
    // Pour tous les fichiers...
    for(vector<string>::iterator currFichier = nomsFichiers.begin() ;
          currFichier != nomsFichiers.end(); ++currFichier)
    {
        Parseur fichpars(*currFichier);
        std::pair<std::string, int> * ident = fichpars.NextIdent();
        while(ident->first != ""){
          bool isMotCle = motscles.IsMotCle(ident->first);
          if((!motsExclus && isMotCle) || (motsExclus && !isMotCle)) {
            index->AddLine(ident->first, ident->second, *currFichier);
          }
          ident = fichpars.NextIdent();
        }
    }
    return index;
}

//----------------------------------------------- Surcharge d'opérateurs

Analyseur & Analyseur::operator =(const Analyseur & unAnalyseur)
```

```
61.  {
62.      motscles = unAnalyseur.motscles;
63.      nomsFichiers = unAnalyseur.nomsFichiers;
64.      motsExclus = unAnalyseur.motsExclus;
65.        return *this;
66.  } //----- Fin de operator =
67.
68.
69.  //-------------------------------------------- Constructeurs - destructeur
70.
71.  Analyseur::Analyseur(ListeMotCles listeMotCles, vector<string> fichiers, bool exclusion)
72.      : motscles(listeMotCles), nomsFichiers(fichiers), motsExclus(exclusion)
73.  {
74.  } //---- Fin de Analyseur (constructeur)
75.
76.  Analyseur::Analyseur(const Analyseur & unAnalyseur)
77.  {
78.  #ifdef MAP
79.      cout << "Appel au constructeur de copie de <Analyseur>" << endl;
80.  #endif
81.      motscles = unAnalyseur.motscles;
82.      nomsFichiers = unAnalyseur.nomsFichiers;
83.      motsExclus = unAnalyseur.motsExclus;
84.  } //----- Fin de Analyseur (constructeur de copie)
85.
86.  Analyseur::~Analyseur()
87.  {
88.  #ifdef MAP
89.      cout << "Appel au destructeur de <Analyseur>" << endl;
90.  #endif
91.  } //----- Fin de ~Analyseur
92.
93.
94.  //================================================================== PRIVE
95.
96.  //------------------------------------------------------ Méthodes protégées
```

Parseur.h

```
1.  /*************************************************************************
2.   Parseur - Analyse un fichier pour y trouver des identificateurs C++
3.   -------------------
4.   début                : 19 nov. 2010
5.   copyright            : (C) 2010 par ssignoud et tpatel
6.   *************************************************************************/
7.
8.  // Comme autorisé (après demande) par les professeurs en séance de TP,
9.  // ces fichiers sources utilisent la norme d'écriture Doxygen/Javadoc en
10. // lieu et place du « Guide de Style INSA » original.
11. // Ses conseils et ses principes de formatages (retours à la ligne et
12. // informations algorithmiques complémentaires) sont cependant conservés.
13.
14. //========[ Interface de la classe <Parseur> (fichier Parseur.h) ]========
15.
16. #if ! defined ( PARSEUR_H_ )
17. #define PARSEUR_H_
18.
19. //------------------------------------------------- Interfaces utilisées
20. #include <fstream>
21. #include <string>
22.
23. /**
24.  * Analyse un fichier pour y trouver des identificateurs C++.
25.  *
26.  * Les identificateurs C++ sont des chaînes de type [A-Za-z_][A-Za-z0-9_]*
27.  * qui sont contenues dans des fichiers. Cette classe ne s'occupe pas de
28.  * savoir si ces identificateurs correspondent à des identifiants (ou mots
29.  * clés) ; voir la classe Analyseur.
30.  */
31. class Parseur
32. {
33.         //================================================================== PUBLIC
34.
35.     public:
36.         //------------------------------------------------------ Méthodes publiques
37.
38.         /**
39.          * Cherche le prochain identifiant présent dans le fichier et le retourne
40.          * ainsi que sa position.
41.          *
42.          * @return Une paire composée de l'identifiant (chaîne) et d'un numéro
43.          * de ligne (positif non nul). Si aucun autre identifiant ne peut être
44.          * trouvé dans le fichier, retourne une paire contenant une chaîne vide
45.          * comme premier élément et -1 comme second élément.
46.          */
47.         std::pair<std::string, int>* NextIdent();
48.
49.         //-------------------------------------------------- Surcharge d'opérateurs
50.
51.         // Aucune surcharge d'opérateur d'affectation. L'objet contient
52.         // un ifstream, qui ne gère pas la copie ; de plus, l'affectation
53.         // d'un parseur à un autre n'a pas de sens dans l'application.
54.
55.         //-------------------------------------------- Constructeurs - destructeur
56.
57.         // Aucun constructeur de copie. L'objet contient un ifstream, qui ne
58.         // gère pas la copie ; de plus, la construction d'un parseur à partir
59.         // d'un autre n'a pas de sens dans l'application.
60.
```

```
61.         /**
62.          * Construit un parseur chargeant le fichier au nom fileName.
63.          *
64.          * @contract Le fichier fileName doit correspondre à un fichier existant
65.          * et valide.
66.          */
67.         Parseur(std::string fileName);
68.
69.         /**
70.          * Détruit le parseur.
71.          */
72.         virtual ~Parseur();
73.
74.         //================================================================== PRIVÉ
75.
76.     protected:
77.         //------------------------------------------------------ Méthodes protégées
78.
79.         //-------------------------------------------------------- Attributs protégés
80.
81.         /**
82.          * Numéro de la ligne actuellement analysée.
83.          */
84.         int currentLine;
85.
86.         /**
87.          * Flux de lecture positionné sur le fichier.
88.          */
89.         std::ifstream* fichier;
90.
91.     private:
92.         // Méthodes désactivées (objets non copiables, voir plus haut)
93.         Parseur& operator=(Parseur const &);
94.         Parseur(const Parseur & unParseur);
95.
96. };
97.
98. //--------------------------- Autres définitions dépendantes de <Parseur>
99.
100. #endif // PARSEUR_H_
```

Parseur.cpp

```
1.  /*************************************************************************
2.   Parseur - Analyse un fichier pour y trouver des identificateurs C++
3.   -------------------
4.   début                : 19 nov. 2010
5.   copyright            : (C) 2010 par ssignoud et tpatel
6.   *************************************************************************/
7.
8.  // Comme autorisé (après demande) par les professeurs en séance de TP,
9.  // ces fichiers sources utilisent la norme d'écriture Doxygen/Javadoc en
10. // lieu et place du « Guide de Style INSA » original.
11. // Ses conseils et ses principes de formatages (retours à la ligne et
12. // informations algorithmiques complémentaires) sont cependant conservés.
13.
14. //===[ Réalisation de la classe <Parseur> (fichier Parseur.cpp) ]===
15.
16. //================================================================ INCLUDE
17.
18. //--------------------------------------------------- Include système
19. using namespace std;
20. #include <iostream>
21. #include <string>
22. #include <cctype>
23.
24. //------------------------------------------------------ Include personnel
25. #include "Parseur.h"
26.
27. //================================================================ PUBLIC
28.
29. /**
30.  * @algorithm Cet algorithme est disponible dans le dossier de
31.  * Spécification.
32.  */
33. pair<string, int>* Parseur::NextIdent()
34. {
35.     char c;
36.     bool estDansIdentif = false;
37.     string buffer = "";
38.     int nbreCar = 0;
39.
40.     // Tant qu'il reste quelque chose à lire dans le fichier
41.     while(*fichier) {
42.         fichier->get(c);
43.         if(c == '\n') {
44.             // Mise à jour du numéro de ligne à chaque retour chariot
45.             currentLine++;
46.         }
47.         if((isalnum(c) || c == '_') && estDansIdentif) {
48.             nbreCar++;
49.             if(nbreCar > 256) {
50.                 // Cas où l'identifiant est trop grand : il est ignoré
51.                 buffer = "";
52.                 estDansIdentif = false;
53.                 nbreCar = 0;
54.             } else {
55.                 // Tout va bien : on ajoute le caractère au buffer
56.                 buffer += c;
57.             }
58.         } else if ( (!isalnum(c) && c != '_') && estDansIdentif) {
59.             // On vient de terminer un identifiant : on le retourne
60.             estDansIdentif = false;
```

```
61.             return new pair<string, int>(buffer, currentLine);
62.         } else if((isalpha(c) || c == '_') && !estDansIdentif) {
63.             // On découvre un nouveau caractère de début : on démarre
64.             // l'enregistrement de le buffer
65.             buffer = c;
66.             estDansIdentif = true;
67.             nbreCar = 1;
68.         }
69.     }
70.     // Plus rien à lire : le fichier est terminé. On retourne une
71.     // chaîne vide pour indiquer cet état de fait
72.     return new pair<string, int>("", -1);
73. } //----- Fin de NextIdent
74.
75. //------------------------------------------------- Surcharge d'opérateurs
76.
77. // Aucune : ifstream ne supporte pas la copie.
78.
79. //-------------------------------------------- Constructeurs - destructeur
80.
81. Parseur::Parseur(string fileName) :
82.     currentLine(1)
83. {
84. #ifdef MAP
85.     cout << "Appel au constructeur de <Parseur>" << endl;
86. #endif
87.     fichier = new ifstream(fileName.c_str());
88. } //----- Fin de Parseur
89.
90. Parseur::~Parseur()
91. {
92. #ifdef MAP
93.     cout << "Appel au destructeur de <Parseur>" << endl;
94. #endif
95.     fichier->close();
96.     delete fichier;
97. } //----- Fin de ~Parseur
98.
99.
100. //============================================================================ PRIVE
101.
102. //------------------------------------------------------ Méthodes protégées
```

Index.h

```
1.  /*************************************************************************
2.   Index - Stocke des références vers des identifiants, fichiers et lignes
3.   -------------------
4.   début                : 19 nov. 2010
5.   copyright            : (C) 2010 par ssignoud et tpatel
6.   *************************************************************************/
7.
8.  // Comme autorisé (après demande) par les professeurs en séance de TP,
9.  // ces fichiers sources utilisent la norme d'écriture Doxygen/Javadoc en
10. // lieu et place du « Guide de Style INSA » original.
11. // Ses conseils et ses principes de formatages (retours à la ligne et
12. // informations algorithmiques complémentaires) sont cependant conservés.
13.
14. // ==========[ Interface de la classe <Index> (fichier Index.h) ]=========
15.
16. #if ! defined ( INDEX_H_ )
17. #define INDEX_H_
18.
19. //------------------------------------------------ Interfaces utilisées
20. #include <string>
21. #include <map>
22. #include <list>
23.
24. /**
25.  * Stocke des références vers des identifiants, contenus dans des fichiers
26.  * et des lignes.
27.  *
28.  * Pour utiliser efficacement cette classe, il est recommandé d'utiliser
29.  * une boucle testant si un nouvel identifiant, fichier ou ligne existe.
30.  * Les méthodes HasNextIdent et NextIdent vous permettront, par exemple,
31.  * de parcourir rapidement et clairement la liste des identifiants contenu
32.  * dans l'index.
33.  * Pour redémarrer le parcours du contenu de l'index, on utilisera Reset.
34.  */
35. class Index
36. {
37.         //================================================================ PUBLIC
38.
39.     public:
40.
41.         typedef std::pair< std::string, std::list<int>* > fichierLignes;
42.         typedef std::map< std::string, std::list<fichierLignes>* > identFichiers;
43.
44.         //------------------------------------------------- Méthodes publiques
45.
46.         /**
47.          * Ajoute une référence à identifiant, placé à la ligne lineNum dans
48.          * le fichier nomFichier.
49.          *
50.          * Cette opération remet à zéro les méthodes de parcours de l'index.
51.          *
52.          * Contrat : nomFichier doit être le même fichier que la référence qui
53.          * vient d'être ajoutée, ou un nom de fichier qui n'a pas déjà été référencé
54.          * dans l'index pour l'identificateur indiqué.
55.          *
56.          * @param identifiant L'identifiant à référencer (chaîne non vide).
57.          * @param lineNum La ligne concernée (entier naturel non nul).
58.          * @param nomFichier Le fichier concerné (chaîne non vide).
59.          * @return Si l'opération se déroule sans encombres, true ; sinon, false.
60.          */
```

```
61.          bool AddLine(std::string identifiant, int lineNum, std::string nomFichier);
62.
63.          /**
64.           * Indique si un autre identifiant existe dans l'index.
65.           *
66.           * @return S'il existe un autre identifiant, retourne true ; sinon, false.
67.           */
68.          bool HasNextIdent();
69.
70.          /**
71.           * Indique si un autre fichier associé à l'identifiant courant existe dans l'index.
72.           *
73.           * @return S'il existe un autre fichier, retourne true ; sinon, false.
74.           */
75.          bool HasNextFile();
76.
77.          /**
78.           * Indique s'il existe une autre ligne associée à l'identifiant courant dans
79.           * l'index.
80.           *
81.           * @return S'il existe une autre ligne, retourne true ; sinon, false.
82.           */
83.          bool HasNextLine();
84.
85.          /**
86.           * Récupère la prochaine ligne associée à l'identifiant en cours.
87.           *
88.           * @return La prochaine ligne, ou -1 si plus aucun autre ligne ne contient
89.           * l'identifiant.
90.           */
91.          int NextLine();
92.
93.          /**
94.           * Récupère le prochain fichier où se situe l'identifiant en cours.
95.           *
96.           * @return Le prochain fichier, ou une chaîne vide si plus aucun autre fichier
97.           * ne contient
98.           * l'identifiant.
99.           */
100.         std::string NextFile();
101.
102.         /**
103.          * Récupère le prochain identifiant contenu dans l'index des références.
104.          *
105.          * @return Le prochain identifiant, ou une chaîne vide si la fin de l'index est
    atteinte.
106.          */
107.         std::string NextIdent();
108.
109.         //-------------------------------------------------- Surcharge d'opérateurs
110.
111.         /**
112.          * Assigne le contenu d'unIndex dans l'index.
113.          *
114.          * @param unIndex L'index dont le contenu est copié.
115.          */
116.         Index & operator =(const Index & unIndex);
117.
118.         //-------------------------------------------- Constructeurs - destructeur
119.
120.
121.         /**
122.          * Construit un nouvel Index à partir des données d'unIndex (constructeur
123.          * de copie).
```

```
124.          *
125.          * @param unIndex L'index dont le contenu est copié.
126.          */
127.         Index(const Index & unIndex);
128.
129.         /**
130.          * Construit un nouvel Index vide.
131.          */
132.         Index();
133.
134.         /**
135.          * Dértuit l'index.
136.          */
137.         virtual ~Index();
138.
139.         //========================================================================= PRIVÉ
140.
141.     protected:
142.         //------------------------------------------------------ Méthodes protégées
143.
144.         void resetIterators();
145.
146.         //-------------------------------------------------------- Attributs protégés
147.         /**
148.          * Contient les données de l'index.
149.          *
150.          * Représentation rapide (syntaxe YAML) :
151.          *   world:
152.          *     - (fichier1.cpp, [1, 2, 8, 12])
153.          *     - (fichier2.cpp, [3, 4, 5, 12])
154.          *     - (fichier3.cpp, [15, 16, 23, 42])
155.          *   test:
156.          *     - (fichier1.cpp, [4, 8])
157.          *     - (fichier5.cpp, [2])
158.          */
159.         identFichiers refs;
160.
161.         /**
162.          * Indique si les itérateurs internes sont valides ou non.
163.          *
164.          * Typiquement, dès qu'une insertion est effectuée, les itérateurs
165.          * ne sont plus utilisables. AddLine mets alors areItValid à false
166.          * et les méthodes utilisant les itérateurs testent si areItValid
167.          * est vrai ; sinon, la méthode resetIterators est appelée.
168.          */
169.         bool areItValid;
170.
171.         /**
172.          * Contient l'itérateur utilisé pour parcourir les identifiants.
173.          */
174.         identFichiers::iterator itIdent;
175.
176.         /**
177.          * Itérateur pointant sur l'identifiant courant.
178.          */
179.         identFichiers::iterator curIdent;
180.
181.         /**
182.          * Contient l'itérateur actuel sur les fichiers.
183.          */
184.         std::list<fichierLignes>::iterator itFichier;
185.
186.         /**
187.          * Itérateur pointant sur le fichier courant.
```

```
188.          */
189.         std::list<fichierLignes>::iterator curFichier;
190.
191.         /**
192.          * Contient l'itérateur actuel sur les lignes.
193.          */
194.         std::list<int>::iterator itLigne;
195.
196.         /**
197.          * Itérateur pointant sur la ligne courante.
198.          */
199.         std::list<int>::iterator curLigne;
200. };
201.
202. //---------------------------- Autres définitions dépendantes de <Index>
203.
204. #endif // INDEX_H_
```

Index.cpp

```cpp
/*************************************************************************
 Index - Stocke des références vers des identifiants, fichiers et lignes
 -------------------
 début                : 19 nov. 2010
 copyright            : (C) 2010 par ssignoud et patel
 *************************************************************************/

// Comme autorisé (après demande) par les professeurs en séance de TP,
// ces fichiers sources utilisent la norme d'écriture Doxygen/Javadoc en
// lieu et place du « Guide de Style INSA » original.
// Ses conseils et ses principes de formatages (retours à la ligne et
// informations algorithmiques complémentaires) sont cependant conservés.

// =======[ Réalisation de la classe <Index> (fichier Index.cpp) ]========

// ================================================================ INCLUDE

// ---------------------------------------------------- Include système

using namespace std;
#include <iostream>

//------------------------------------------------------ Include personnel
#include "Index.h"

// ================================================================ PUBLIC

/**
 * @algorithm L'algorithme a été décrit dans le dossier de Spécifiations.
 */
bool Index::AddLine(std::string identifiant, int lineNum, std::string nomFichier)
{
    // Les structures internes sont modifiées. On invalide les itérateurs
    // de lecture pour éviter des erreurs légitimes et forcer leur
    // réinitialisation.
    areItValid = false;

    identFichiers::iterator it;
    it = refs.find(identifiant);
    if (it == refs.end())
    {
        list<int>* lignes = new list<int>();
        lignes->push_back(lineNum);

        list<fichierLignes>* liste = new list<fichierLignes>();
        liste->push_back(make_pair(nomFichier, lignes));
        return refs.insert(make_pair(identifiant, liste)).second;
    }
    else if ( (it->second->back().first) != nomFichier)
    {
        list<int>* lignes = new list<int>();
        lignes->push_back(lineNum);

        it->second->push_back(make_pair(nomFichier, lignes));
        return true;
    }
    // On parse linéairement les fichiers, il suffit donc de vérifier que la dernière
    // ligne mémorisée (pour cet identifiant dans ce fichier) n'est pas celle que
    // l'on veut ajouter
    else if( (it->second->back().second->back()) != lineNum)
```

```
61.     {
62.         it->second->back().second->push_back(lineNum);
63.         return true;
64.     }
65.     return false;
66. } //----- Fin de AddLine
67.
68. bool Index::HasNextIdent()
69. {
70.   if(!areItValid) resetIterators();
71.   return itIdent != refs.end();
72. } //----- Fin de HasNextIdent
73.
74. bool Index::HasNextFile()
75. {
76.   if(!areItValid) resetIterators();
77.   return itFichier != curIdent->second->end();
78. } //----- Fin de HasNextFile
79.
80. bool Index::HasNextLine()
81. {
82.   if(!areItValid) resetIterators();
83.   return itLigne != curFichier->second->end();
84. } //----- Fin de HasNextLine
85.
86. string Index::NextIdent()
87. {
88.   string retour = itIdent->first;
89.   curIdent = itIdent;
90.   ++itIdent;
91.   curFichier = curIdent->second->begin();
92.   itFichier = curFichier;
93.   return retour;
94. } //----- Fin de NextIdent
95.
96. string Index::NextFile()
97. {
98.   string retour = itFichier->first;
99.   curFichier = itFichier;
100.  ++itFichier;
101.  curLigne = curFichier->second->begin();
102.  itLigne = curLigne;
103.  return retour;
104. } //----- Fin de NextFile
105.
106. int Index::NextLine()
107. {
108.   int retour = *itLigne;
109.   curLigne = itLigne;
110.   ++itLigne;
111.   return retour;
112. } //----- Fin de NextLine
113.
114.
115. //------------------------------------------------- Surcharge d'opérateurs
116.
117. Index & Index::operator =(const Index & unIndex)
118. {
119.   refs = unIndex.refs;
120.   resetIterators();
121.   return *this;
122. } //----- Fin de operator =
123.
124.
```

```
125.  //------------------------------------------ Constructeurs - destructeur
126.
127.  Index::Index(const Index & unIndex)
128.  {
129.  #ifdef MAP
130.      cout << "Appel au constructeur de copie de <Index>" << endl;
131.  #endif
132.    refs = unIndex.refs;
133.    resetIterators();
134.  } //----- Fin de Index (constructeur de copie)
135.
136.
137.  Index::Index()
138.  {
139.  #ifdef MAP
140.      cout << "Appel au constructeur de <Index>" << endl;
141.  #endif
142.  } //----- Fin de Index
143.
144.
145.  Index::~Index()
146.  {
147.  #ifdef MAP
148.      cout << "Appel au destructeur de <Index>" << endl;
149.  #endif
150.  } //----- Fin de ~Index
151.
152.
153.  // ================================================================== PRIVE
154.
155.  //------------------------------------------------------ Méthodes protégées
156.
157.  void Index::resetIterators()
158.  {
159.    itIdent = refs.begin();
160.    itFichier = refs.begin()->second->begin();
161.    itLigne = refs.begin()->second->begin()->second->begin();
162.    curIdent = refs.begin();
163.    curFichier = refs.begin()->second->begin();
164.    curLigne = refs.begin()->second->begin()->second->begin();
165.    areItValid = true;
166.  } //----- Fin de resetIterators
```

ListeMotCles.h

```
1.  /*************************************************************************
2.   ListeMotCles - Stocke la liste des éléments.
3.   -------------------
4.   début                : 19 nov. 2010
5.   copyright            : (C) 2010 par ssignoud et tpatel
6.   *************************************************************************/
7.
8.  // Comme autorisé (après demande) par les professeurs en séance de TP,
9.  // ces fichiers sources utilisent la norme d'écriture Doxygen/Javadoc en
10. // lieu et place du « Guide de Style INSA » original.
11. // Ses conseils et ses principes de formatages (retours à la ligne et
12. // informations algorithmiques complémentaires) sont cependant conservés.
13.
14. // ===[ Interface de la classe <ListeMotCles> (fichier ListeMotCles.h) ]===
15.
16. #if ! defined ( LISTEMOTCLES_H_ )
17. #define LISTEMOTCLES_H_
18.
19. //------------------------------------------------ Interfaces utilisées
20. #include <set>
21. #include <string>
22.
23. /**
24.  * Stocke la liste des éléments et rends aisé le fait de tester si un
25.  * mot est présent dans cette liste ou non.
26.  */
27. class ListeMotCles
28. {
29.         //================================================================ PUBLIC
30.
31.     public:
32.         //------------------------------------------------------ Méthodes publiques
33.
34.         /**
35.          * Ajoute un mot clé à la liste.
36.          *
37.          * @return Si le mot clé existe déjà ou n'a pas pu être inséré, false ;
38.          * sinon, true.
39.          */
40.         bool AddMotCle(std::string unMotCle);
41.
42.         /**
43.          * Vérifie si un mot clé est déjà dans la liste.
44.          *
45.          * @return Si le mot clé est dans la liste, true ; sinon, false.
46.          */
47.         bool IsMotCle(std::string unMotCle) const;
48.
49.         /**
50.          * Indique si la liste des mots clés est vide ou non.
51.          *
52.          * @return Si la liste est vide, true ; sinon false.
53.          */
54.         bool IsVide() const;
55.
56.         //-------------------------------------------------- Surcharge d'opérateurs
57.
58.         /**
59.          * Remplace la liste des mots clés par celle d'uneAutreListe.
60.          */
```

```
61.         ListeMotCles & operator =(const ListeMotCles & uneAutreListe);
62.
63.         //------------------------------------------ Constructeurs - destructeur
64.
65.         /**
66.          * Crée une nouvelle liste de mot clés, contenant la liste de mots clés
67.          * d'uneAutreListe de mots clés.
68.          */
69.         ListeMotCles(const ListeMotCles & uneAutreListe);
70.
71.         /**
72.          * Construit une nouvelle liste de mots clés.
73.          */
74.         ListeMotCles();
75.
76.         /**
77.          * Détruit la liste de mots clés.
78.          */
79.         virtual ~ListeMotCles();
80.
81.         //================================================================== PRIVÉ
82.
83.     protected:
84.         //------------------------------------------------------ Méthodes protégées
85.
86.         //------------------------------------------------------- Attributs protégés
87.
88.         /**
89.          * Contient la liste des mots clés.
90.          */
91.         std::set<std::string> ref;
92. };
93.
94. //--------------------------- Autres définitions dépendantes de <ListeMotCles>
95.
96. #endif // LISTEMOTCLES_H_
```

ListeMotCles.cpp

```cpp
/*************************************************************************
 ListeMotCles - Stocke la liste des éléments.
 -------------------
 début                : 19 nov. 2010
 copyright            : (C) 2010 par ssignoud et tpatel
 *************************************************************************/

// Comme autorisé (après demande) par les professeurs en séance de TP,
// ces fichiers sources utilisent la norme d'écriture Doxygen/Javadoc en
// lieu et place du « Guide de Style INSA » original.
// Ses conseils et ses principes de formatages (retours à la ligne et
// informations algorithmiques complémentaires) sont cependant conservés.

//=[ Réalisation de la classe <ListeMotCles> (fichier ListeMotCles.cpp) ]=

//================================================================ INCLUDE

//---------------------------------------------------- Include système
using namespace std;
#include <iostream>
#include <string>

//------------------------------------------------------ Include personnel
#include "ListeMotCles.h"

//================================================================ PUBLIC

bool ListeMotCles::AddMotCle(string unMotCle)
{
    return ref.insert(unMotCle).second;
} //----- Fin de AddMotCle

bool ListeMotCles::IsMotCle(std::string unMotCle) const
{
    return ref.find(unMotCle) != ref.end();
} //----- Fin de IsMotCle

bool ListeMotCles::IsVide() const
{
    return ref.empty();
} //----- Fin de IsVide


//------------------------------------------------- Surcharge d'opérateurs

ListeMotCles & ListeMotCles::operator =(const ListeMotCles & uneListeMotCles)
{
    ref = uneListeMotCles.ref;
    return *this;
} //----- Fin de operator =


//-------------------------------------------- Constructeurs - destructeur

ListeMotCles::ListeMotCles(const ListeMotCles & uneListeMotCles)
{
#ifdef MAP
    cout << "Appel au constructeur de copie de <ListeMotCles>" << endl;
#endif
    ref = uneListeMotCles.ref;
```

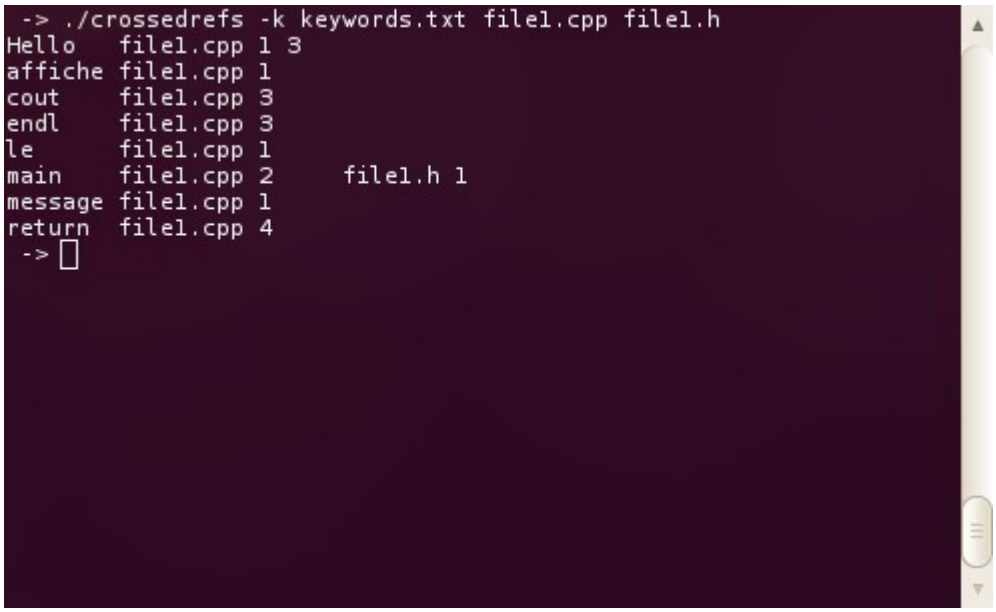
```
61.  } //----- Fin de ListeMotCles (constructeur de copie)
62.
63.  ListeMotCles::ListeMotCles()
64.  {
65.  #ifdef MAP
66.      cout << "Appel au constructeur de <ListeMotCles>" << endl;
67.  #endif
68.      ref = set<string>();
69.  } //----- Fin de ListeMotCles
70.
71.  ListeMotCles::~ListeMotCles()
72.  {
73.  #ifdef MAP
74.      cout << "Appel au destructeur de <ListeMotCles>" << endl;
75.  #endif
76.  } //----- Fin de ~ListeMotCles
77.
78.
79.  //============================================================================ PRIVE
80.
81.  //------------------------------------------------------- Méthodes protégées
```

Résultats des tests

Test A1

Fonctionnement classique avec fichier de mots clé, mots clés exclus

Avec la ligne de commande et les fichiers de tests proposés dans le *Dossier de Spécification et de Conception*, la sortie est :

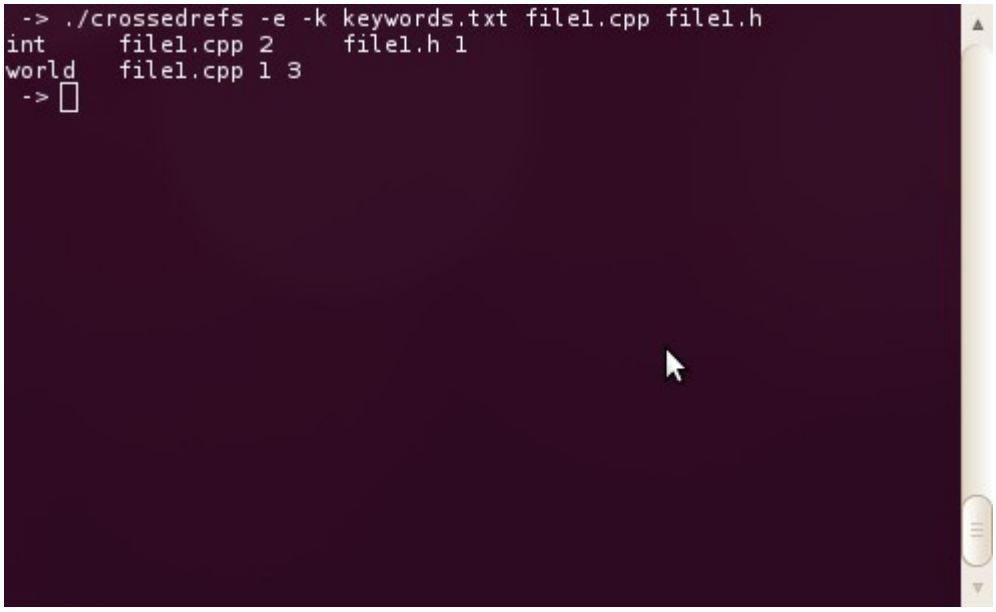


Test réussi.

Test A2

Fonctionnement classique avec fichier de mots clé, mots clés recherchés uniquement

Avec la ligne de commande et les fichiers de tests proposés dans le *Dossier de Spécification et de Conception*, la sortie est :



Test réussi.

Test B1

Fonctionnement classique avec mots clés du C++, mots clés exclus

Avec la ligne de commande et les fichiers de tests proposés dans le *Dossier de Spécification et de Conception*, la sortie est :

Test réussi.

Test B2

Fonctionnement classique avec mots clés du C++, mots clés uniquement

Avec la ligne de commande et les fichiers de tests proposés dans le *Dossier de Spécification et de Conception*, la sortie est :

Test réussi.

Test C1

Fonctionnement classique avec fichier mot clé, mots clés uniquement

Avec la ligne de commande et les fichiers de tests proposés dans le *Dossier de Spécification et de Conception*, la sortie est :

Test réussi.