

# SMP

## TP 1

Diarra DIOPE

Eugénie ROQUAIN



12 Octobre 2025

Systèmes Embarqués Communicants 2025/2026

# PTR - TP1

<b>PTR - TP1</b>	<b>1</b>
1. Introduction	2
1.1 Objectif du TP :	2
2. Conception du lexique (Partie 1 du TP)	2
2.1. Analyse et choix de la structure de données	2
2.2. Diagramme de classe	2
2.3. Algorithmes principaux	4
3. Opérations sur le lexique	5
3.1. Sauvegarde	5
3.2. Recherche et suppression	7
3.3. Affichage et statistiques	10
3.4. Surcharge d'opérateurs	11

# 1. Introduction

## 1.1 Objectif du TP :

expliquer en quelques lignes le but du travail

L'objectif de ce TP est de mettre en œuvre les principes de structuration de données pour créer un programme C++ capable de construire et manipuler un lexique de mots à partir d'un fichier texte volumineux.

## 2. Conception du lexique (Partie 1 du TP)

### 2.1. Analyse et choix de la structure de données

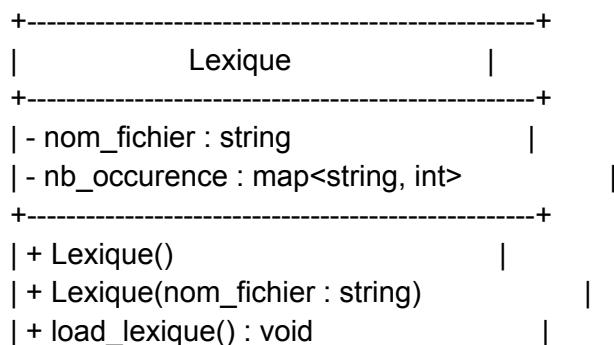
- Comparaison possible entre `map`, `unordered_map`, `vector`, etc.
- Justification du choix final (souvent `std::map<std::string, int>` pour tri et comptage simple).

### 2.2. Diagramme de classe

- Présenter la **classe Lexique** :
  - Attributs (`nom_fichier`, `nb_occurence`)
  - Méthodes principales : `load_lexique()`, `save_lexique()`, `test_presence()`, `delete_mot()`, `display()`...

La classe `Lexique` constitue le cœur du programme. Elle regroupe les données (nom du fichier et mots avec leur nombre d'occurrences) ainsi que les méthodes nécessaires à leur manipulation. Cette conception modulaire permet de faire évoluer facilement le programme, notamment pour ajouter la gestion des numéros de ligne dans une classe dérivée.

Diagramme de classe :



```

| + save_lexique(nom_fichier : string) const : void |
| + test_presence(mot : string) const : bool      |
| + delete_mot(mot : string) : int                |
| + display() const : void                        |
| + ~Lexique()                                   |
+-----+

```

Élément	Description
---------	-------------

<b>Attribut nom_fichier</b>	Contient le nom du fichier texte à analyser.
-----------------------------	--

<b>Attribut nb_occurence</b>	std::map<string,int> : structure clé/valeur où chaque clé est un mot et chaque valeur son nombre d'occurrences. Le choix d'un map (plutôt qu'un unordered_map) garantit un <b>tri lexical automatique</b> , pratique pour un affichage ordonné.
------------------------------	---

<b>Constructeurs</b>	Le constructeur par défaut initialise un lexique vide ; le second initialise le nom du fichier et prépare la map.
----------------------	---

<b>Méthode load_lexique()</b>	Lit le fichier ligne par ligne et incrémente les occurrences des mots dans la map.
-------------------------------	--

<b>Méthode</b> <b>save_lexique()</b>	Sauvegarde le contenu du lexique dans un fichier texte externe.
---	---

<b>Méthodes</b> <b>test_presence()</b> et <b>delete_mot()</b>	Permettent la recherche et la suppression d'un mot dans le lexique.
---	---

<b>Méthode</b> <b>display()</b>	Affiche sur la console le contenu du lexique et le nombre total de mots distincts.
------------------------------------	--

<b>Destructeur</b>	Ne fait rien de particulier : les objets standards (string, map) se détruisent automatiquement.
--------------------	---

### 2.3. Algorithmes principaux

- Comptage des occurrences

*fonction nb\_orccurence :*

-> Cette fonction **remplace intégralement le contenu actuel du lexique** (**nb\_occurence**) par une nouvelle map fournie en paramètre.

```
void Lexique::setNbOccurence(const std::map<std::string, int>& nouveau_map) {  
    ///copie toutes les paires mot  
    nb_occurence = nouveau_map;  
}
```

*fonction load\_lexique :*

-> Cette fonction **lit un fichier texte** (dont le nom est stocké dans l'attribut `nom_fichier`) et **construit automatiquement le lexique** en comptant le nombre d'occurrences de chaque mot.

```
void Lexique::load_lexique() {
    std::ifstream input_file(this->nom_fichier);
    if (!input_file) {
        std::cerr << "Erreur : impossible d'ouvrir le fichier " << nom_fichier << std::endl;
        return;
    }

    std::string mot;
    while (input_file >> mot) {
        ++nb_occurrence[mot]; // incrémente le compteur pour chaque mot
    }

    std::cout << "Fichier " << nom_fichier << " chargé avec " << nb_occurrence.size() << " mots uniq
}
```

`load_lexique()` permet de **remplir automatiquement le lexique à partir d'un fichier texte**,

tandis que `setNbOccurrence()` sert à **remplacer manuellement le contenu du lexique** avec une map déjà construite.

## 3. Opérations sur le lexique

### 3.1. Sauvegarde

- Explication du fonctionnement de `save_lexique()`

Cette fonction **enregistre le contenu du lexique dans un fichier texte**.

Elle ouvre un fichier en écriture (`std::ofstream`), puis parcourt la map `nb_occurrence` contenant tous les mots et leurs nombres d'occurrences.

Pour chaque paire *mot* → *occurrence*, elle écrit une ligne dans le fichier au format :

```
/**
 * @brief Sauvegarde le contenu du lexique dans un fichier texte.
 *
 * Ouvre un fichier de sortie et écrit pour chaque mot de la map
 nb_occurrence
 * une ligne contenant le mot et son nombre d'occurrences, séparés par
 un espace.
```

```

* @param nom_fichier Nom du fichier de sortie où enregistrer le
lexique.

* @note La méthode est const : elle ne modifie pas le contenu du
lexique.

* @warning Si le fichier ne peut pas être ouvert, un message d'erreur
est affiché

*          et aucune donnée n'est écrite.

*/

void Lexique::save_lexique(const std::string &nom_fichier) const
{
    std::ofstream output_file(nom_fichier, std::ios::binary);

    if (!output_file)
    {
        std::cerr << "Erreur : impossible d'ouvrir le fichier pour
écriture." << std::endl;

        return;
    }

    for (const auto &[mot, occ] : nb_occurrence)
    {
        output_file << mot << " " << occ << "\n";
    }

    std::cout << "Lexique sauvegardé avec succès dans " << nom_fichier
<< std::endl;
}

```

- Exemple de fichier de sortie :

```
"'Flock'; 1
"'Have 1
"'I 1
"'Mariage 1
"'Monsieur 2
"'No,' 1
"'The 1
"'They 1
"'Tis 1
"'Wash 1
"'We 1
"'What 1
"'Whither 1
"'With 1
"'Yes,' 1
"'You 2
"----." 1
"--And 1
"--Dier." 1
"--General 1
"----." 1
```

### 3.2. Recherche et suppression

- Présenter `test_presence()` et `delete_mot()`



```

1  /**
2   * @brief Vérifie si un mot est présent dans le lexique.
3   *
4   * Parcourt la map nb_occurrence et retourne true si la clé correspondant au mot est trouvée.
5   *
6   * @param mot Le mot à rechercher dans le lexique.
7   * @return true si le mot est présent.
8   * @return false sinon.
9   */
10
11 bool Lexique::test_presence(const std::string &mot) const
12 {
13     for (std::map<std::string, int>::const_iterator iter = this->nb_occurrence.begin();
14          iter != this->nb_occurrence.end(); iter++)
15     {
16         if (iter->first == mot)
17         {
18             return true;
19         }
20     }
21     return false;
22 }
23
24
25 /**
26 * @brief Supprime un mot du lexique s'il est présent.
27 *
28 * Recherche le mot dans nb_occurrence avec find(), puis l'efface s'il existe.
29 *
30 * @param mot Le mot à supprimer.
31 * @return int 1 si le mot a été supprimé, 0 s'il n'était pas présent.
32 */
33
34 int Lexique::delete_mot(const std::string &mot)
35 {
36     auto it = nb_occurrence.find(mot);
37     if (it != nb_occurrence.end())
38     {
39         nb_occurrence.erase(it);
40         return 1;
41     }
42     return 0;
43 }

```

- Cas de test : mot existant / non existant

Nous allons tester si le mot “Author” est présent, puis le supprimer et ainsi retester s’il est présent.



```
1  int main(){
2      Lexique lex("lesMiserables_A.txt");
3      lex.load_lexique();
4      lex.save_lexique("fichier test pour.txt");
5
6      if (lex.test_presence("Author"))
7      {
8          std::cout << "Présent" << std::endl;
9      }
10     else
11     {
12         std::cout << "Pas Présent" << std::endl;
13     }
14
15     if (lex.delete_mot("Author"))
16     {
17         std::cout << "Delete" << std::endl;
18     }
19     else
20     {
21         std::cout << "Not Delete" << std::endl;
22     }
23
24     if (lex.test_presence("Author"))
25     {
26         std::cout << "Présent" << std::endl;
27     }
28     else
29     {
30         std::cout << "Pas Présent" << std::endl;
31     }
32     return 0;
33 }
```

- Réponse test :

```
● eugenie@LAPTOP-IIU9M3DG:~/SMP/CN_2/tp1-tp1_diarra/TP1$ ./monprog
Fichier lesMiserables_A.txt chargé avec 53370 mots uniques.
Lexique sauvegardé avec succès dans fichier test pour.txt
Présent
Delete
Pas Présent
○ eugenie@LAPTOP-IIU9M3DG:~/SMP/CN_2/tp1-tp1_diarra/TP1$ █
```

### 3.3. Affichage et statistiques

- Fonction `display()`

```
/**
 * @brief Affiche le contenu complet du lexique sur la console.
 *
 * Si le lexique est vide, un message l'indique. Sinon, la fonction
 * parcourt la map nb_occurrence et affiche chaque mot avec son nombre
 * d'occurrences.
 *
 * Exemple d'affichage :
 * ```
 * Contenu du lexique :
 * amour : 12
 * guerre : 5
 * livre : 8
 * ```
 * @note La méthode est const : elle ne modifie pas les données du
 * lexique.
 */
```

```

void Lexique::display() const
{
    if (nb_occurrence.empty())
    {
        std::cout << "Le lexique est vide." << std::endl;

        return;
    }

    std::cout << "Contenu du lexique :" << std::endl;

    for (const auto &[mot, occ] : nb_occurrence)
    {
        std::cout << mot << " : " << occ << std::endl;
    }
}

```

- Nombre de mots distincts (taille de la map).

```

à : 35
âge : 1
âge, : 1
âme : 2
ædibus : 1
ædiles," : 1
æternam : 1
æternam, : 1
ça : 3
ébranlé : 1
échasses. : 1
éclaireurs_ : 1
éclos. : 1
écrit : 1
épingle : 1
épouvente : 1

```

### 3.4. Surcharge d'opérateurs

### Nouveau diagramme de classe :

```
+-----+
|           Lexique           |
+-----+
| - nom_fichier : string      |
| - nb_occurrence : map<string, int> |
+-----+
| + Lexique()                  |
| + Lexique(nom_fichier : string) |
| + load_lexique() : void      |
| + save_lexique(nom_fichier : string) const : void |
| + test_presence(mot : string) const : bool |
| + delete_mot(mot : string) : int |
| + display() const : void      |
| + operator+=(autre : Lexique) : Lexique& |
| + operator-=(autre : Lexique) : Lexique& |
| + friend operator<<(os : ostream&, lex : Lexique) |
| + ~Lexique()                  |
+-----+
```

- << : affichage d'un lexique complet
- += : fusion de deux lexiques (décrire l'algorithme)
- -= : différence entre deux lexiques
- Schéma ou pseudo-code explicatif pour chacun
- Tests :

Affichage des lexiques de départ (<<) :




```
1  int main(){
2      Lexique lex1("test1.txt");
3      Lexique lex2("test2.txt");
4      lex1.load_lexique();
5      lex1.save_lexique("fichier test_1.txt");
6      lex2.load_lexique();
7      lex2.save_lexique("fichier test_2.txt");
8
9      cout << "=== Lexique 1 ===" << endl;
10     cout << lex1 << endl;
11
12     cout << "=== Lexique 2 ===" << endl;
13     cout << lex2 << endl;
14 }
```

Résultats :

```
=== Lexique 1 ===
Contenu du lexique :
Cachouète : 1

=== Lexique 2 ===
Contenu du lexique :
Manger : 1
des : 1
```

Test de la fusion (opérateur +=) :



```

1  int main(){
2      Lexique lex1("test1.txt");
3      Lexique lex2("test2.txt");
4      lex1.load_lexique();
5      lex1.save_lexique("fichier test_1.txt");
6      lex2.load_lexique();
7      lex2.save_lexique("fichier test_2.txt");
8
9      cout << "=== Fusion lex1 += lex2 ===" << endl;
10     lex1 += lex2; // Fusion des deux lexiques
11     cout << lex1 << endl;
12 }

```

Résultats :

```

=== Fusion lex1 += lex2 ===
Contenu du lexique :
Cachouète : 1
Manger : 1
des : 1

```

Test de la différence (opérateur -=) :



```
1  int main(){
2      Lexique lex1("test1.txt");
3      Lexique lex2("test2.txt");
4      lex1.load_lexique();
5      lex1.save_lexique("fichier test_1.txt");
6      lex2.load_lexique();
7      lex2.save_lexique("fichier test_2.txt");
8
9      cout << "=== Différence lex1 -= lex2 ===" << endl;
10     lex1 -= lex2; // Suppression des mots communs
11     cout << lex1 << endl;
12 }
```

Résultats :

```
=== Différence lex1 -= lex2 ===
Contenu du lexique :
Cachouète : 1
```