UNIVERSITÉ PARIS-XII — FACULTÉ DE SCIENCES ÉCONOMIQUES ET DE GESTION
Master Professionnel *Méthodes Appliquées de la Statistique et de l'Économétrie pour la Recherche, l'Analyse et le Traitement de l'Information*
Année universitaire 2006-2007 — *François LEGENDRE*
F.Legendre@univ-paris12.fr — http://fj.legendre.free.fr

Projet « Lexicométrie »

1 Exposé du projet

Ce projet a pour objet de démontrer les possibilités du langage C++ en matière de traitement des chaînes de caractères et de statistiques simples. Il veut aussi illustrer la puissance de la *Standard Template Library* – la STL.

Il s'agit de produire une première analyse simple des mots d'un texte contenu dans un fichier. On trouve sur Internet le texte intégral de *Madame BOVARY* de Gustave FLAUBERT. C'est sur ce texte que j'applique le traitement.

La variable centrale est ***dict***, une instance du type ***map<string, size_t>***. Ce type, pour rendre le programme plus explicite, est nommé ***Dict*** par l'intermédiaire d'un ***typedef***. Les clefs de ***dict*** sont les mots du texte – en se limitant aux mots de plus ***TAILLE*** lettres. À chaque clef est associé le nombre d'occurrences du mot dans l'ensemble du texte. De même, par l'intermédiaire d'un ***typedef***, l'identificateur ***Paire_dict*** est rendu synonyme du type ***pair<string, size_t>***.

Aussi les dix premières entrées de ce tableau associatif seront-elles les suivantes, au cas particulier de *Madame BOVARY*.

Numéro	Mot (Clef)	Nombre d'occurrences (Valeur)
1	abaissa	1
2	abaissait	1
3	abaissant	4
4	abaissèrent	2
5	abandonna	9
6	abandonnait	6
7	abandonnant	3
8	abandonne	1
9	abandonner	7
10	abandonnez	2

L'utilisation de la STL s'appuie sur la notion de séquence d'entrée. Pour le conteneur ***conteneur*** de la STL, ***conteneur.begin(), conteneur.end()*** est la séquence d'entrée de l'ensemble des éléments du conteneur. Par exemple, la syntaxe « ***sort(tableau.begin(), tableau.end())*** » permet de trier l'ensemble des éléments du conteneur ***tableau*** qui serait, par exemple, un ***vector<size_t>***.

Pour l'instance ***fichier*** d'un ***istream***, la séquence d'entrée ***istream_iterator<T>(fichier)***, ***istream_iterator<T>()*** permet d'extraire une suite de variables de type ***T*** du flux d'entrée ***fichier***. Par exemple, pour extraire une à une les chaînes de caractères d'un flux d'entrée, il suffit de prendre pour ***T*** le type ***string***. L'utilisation d'un ***istreambuf_iterator*** (à la place d'un ***istream_iterator***) permet de lire le flux brut. Par exemple, pour un ***char***, le flux d'entrée est vraiment lu caractère par caractère sans aucun traitement comme en particulier le saut des espaces.

Une variable de type ***string*** peut être initialisée à partir d'une séquence d'entrée dont les éléments sont, bien évidemment, de type ***char***. La ligne suivante

```
string tout((istreambuf_iterator<char>(cin)), istreambuf_iterator<char>()) ;
```

permet donc de placer une copie du flux de caractères de ***cin*** dans une chaîne de caractères. Les parenthèses, qui semblent en plus, autour du premier argument ***istreambuf_iterator<char>(cin)*** du constructeur de ***string*** sont absolument nécessaires. En l'absence de ces parenthèses, la ligne serait interprétée comme la déclaration de la fonction ***tout*** qui retourne une variable de type ***string***.

La STL propose un certain nombre d'algorithmes comme ***transform*** ou ***for_each***. L'algorithme ***transform*** applique une fonction ou un objet-fonction sur chaque élément de la séquence d'entrée et copie le résultat sur une séquence de sortie. L'algorithme ***for_each*** permet d'exécuter une fonction ou un objet-fonction sur chaque élément de la séquence d'entrée.

Aussi la ligne de programme suivante

```
transform(tout.begin(), tout.end(), tout.begin(), recoder) ;
```

permet-elle de recoder chaque caractère de la chaîne ***tout***. Le troisième argument de l'algorithme ***transform*** est le premier élément de la séquence de sortie. En codant ***tout.begin()***, on recode chaque caractère « sur place ». La fonction ***recoder*** est définie comme suit.

```
inline char
recoder (const char & arg)
{
  static string bizarre("ÆæŒœÀàÁáÂâÇçÈèÉéÊêËëÎîÏïÔôöÖÙùÛû") ;
  if ( bizarre.find(arg) != string::npos ) {
    return arg ; }
  else if ( ispunct(arg) ) {
    return ' ' ; }
  else {
    return tolower(arg) ; }
}
```

Son argument, nécessairement, est de la forme « ***const T &*** » où ***T*** est le type d'un élément de la séquence d'entrée [1]. Ici, un élément de la séquence d'entrée est un ***char*** donc l'argument de la fonction ***recoder*** est de la forme « ***const char &*** » (une ***const*** référence sur un ***char***). La variable retournée par la fonction doit être du type d'un élément de la séquence de sortie [2]. Ici, un élément de la séquence de sortie est un ***char***. Le type retourné par la fonction ***recoder*** doit donc être un ***char***.

La fonction ***ispunct(...)*** retourne ***true*** si son argument est un caractère de ponctuation. La fonction ***tolower(...)*** retourne soit son argument en minuscule si ce dernier est une lettre majuscule soit sinon son argument. La chaîne ***"Nous étions à l'Etude, quand le Proviseur entra suivi d'un nouveau habillé en bourgeois et d'un garçon de classe qui portait un grand pupitre."*** est donc transformée en ***"nous étions à l etude quand le proviseur entra suivi d un nouveau habillé en bourgeois et d un garçon de classe qui portait un grand pupitre "***. Il sera donc particulièrement facile d'extraire les mots de cette chaîne de caractères ainsi transformée.

Le tableau associatif ***dict*** est progressivement renseigné à partir de cette extraction de mots. Ce traitement s'effectue en trois étapes.

1. Le texte recodé caractère par caractère au moyen de la fonction ***recoder***, contenu dans la ***string tout***, devient l'argument d'un ***istringstream***.
2. Les mots sont extraits à partir de ce ***istringstream*** en utilisant un ***istream_iterator*** spécialisé pour une ***string***.
3. Le conteneur ***dict*** est renseigné pour chaque mot à l'aide de l'algorithme ***for_each*** et de la fonction ***remplir***.

Cette dernière fonction est programmée comme suit.

```
inline void
remplir(const string & arg)
{
  if ( arg.size() > TAILLE ) {
    ++ dict[arg] ; }
}
```

Son argument est déclaré comme un ***const string &*** puisque chacun des éléments de la séquence d'entrée est du type ***string***. L'expression « ***++ dict[arg]*** », si la clef n'existe pas, crée une entrée en utilisant le constructeur par défaut pour la valeur. La variable ***dict*** est un ***map<string, size_t>***. La valeur est donc un ***size_t*** et le constructeur par défaut d'un ***size_t*** initialise à zéro la valeur. L'expression, ensuite, incrémente d'une unité la valeur. De cette manière, on décompte bien les occurrences de chacun des mots dans le texte.

1. Plus précisément, un élément de la séquence d'entrée doit pouvoir être converti en le type de l'argument de la fonction.
2. Plus précisément, la variable retournée doit pouvoir être convertie en le type d'un élément de la séquence de sortie.

Le fragment de programme qui s'occupe à remplir ***dict*** constitue un bloc. Aussi à la sortie du bloc les variables ***tout*** et ***mots*** seront-elles détruites. C'est bien sûr une bonne idée de chercher à réduire la durée de vie des variables pour limiter, au moment de l'exécution, les ressources utilisées par le programme.

À ce stade, nous sommes en possession d'un dictionnaire dont les clefs, de type ***string***, sont les différents mots de plus de quatre lettres qui surviennent dans le texte et dont la valeur, de type ***size_t***, associée à chaque clef est le nombre d'occurrences du mot. Il nous faut maintenant exploiter ces résultats. Tout d'abord, on veut afficher les vingt mots qui reviennent le plus souvent. Pour cela, il faudrait trier chaque entrée de ***dict*** par ordre décroissant du nombre d'occurrences. Ainsi on peut afficher les vingt mots d'intérêt. Pour trier les entrées, le plus simple est encore de toutes les recopier dans un vecteur puis de trier les éléments de ce vecteur à l'aide de l'algorithme ***sort***.

Chaque entrée dans ***dict*** est une paire. Chaque paire est recopiée dans un vecteur, de type donc ***vector< Paire_dict >***. On va ainsi coder les trois lignes suivantes.

```
vector< Paire_dict > compte ;
for (Dict::const_iterator itr = dict.begin() ; itr != dict.end() ; ++ itr) {
  compte.push_back(* itr) ; }
```

Il est préférable d'utiliser un ***const_iterator*** pour exprimer le fait que l'on va simplement accéder en lecture à chaque entrée de ***dict***. Les sémantiques des notations « ***++ itr*** » et « **** itr*** » quand ***itr*** est un itérateur sur un conteneur de la STL sont expliqués dans la partie du *Support du cours* intitulée « Les tableaux associatifs de la STL ».

Il s'avère cependant plus simple de faire appel à l'algorithme ***copy*** de la STL. Pour désigner le premier élément de la séquence de sortie, il faut coder « ***back_inserter(compte)*** » pour dire que l'on souhaite insérer l'élément à la fin du conteneur.

```
vector< Paire_dict > compte ;
copy(dict.begin(), dict.end(), back_inserter(compte)) ;
```

Le vecteur est ensuite trié. Le vecteur est initialement dans l'ordre des clefs, c'est-à-dire dans l'ordre alphabétique des clefs. Il faut le trier dans l'ordre décroissant des occurrences pour avoir, au début du vecteur, les mots qui reviennent le plus. L'algorithme ***sort*** permet de spécifier, comme troisième argument optionnel, une fonction ou un objet-fonction pour indiquer le critère du tri.

Cette fonction est codée comme suit.

```
inline bool
cmp_trier_occurrences(const Paire_dict & arg1, const Paire_dict & arg2)
{
  return arg1.second > arg2.second ;
}
```

Les deux arguments de cette fonction sont de la forme « ***const T &*** » où ***T*** est le type des éléments à trier. Ici, comme les éléments du vecteur sont tous du type ***Paire_dict***, il faut coder « ***const Paire_dict & arg1*** ». Cette fonction doit retourner ***true*** si les deux éléments sont déjà triés dans le bon ordre. Comme on veut un tri par ordre décroissant en fonction du nombre d'occurrences (le second terme de la paire), la fonction retourne l'expression logique « ***arg1.second > arg2.second*** ». On va donc coder la ligne suivante.

```
sort(compte.begin(), compte.end(), cmp_trier_occurrences) ;
```

On veut ensuite obtenir la distribution de l'occurrence des mots : savoir, par exemple, que 560 mots différents ne surviennent qu'une seule fois dans l'ensemble du texte, que 340 mots différents surviennent deux fois, etc. Différentes tactiques sont à notre disposition.

L'on peut tout d'abord construire un tableau associatif dont la clef est l'occurrence et la valeur la fréquence. C'est alors très simple à programmer. Le fragment de programme suivant s'acquitte de cette tâche.

```
map<size_t, size_t> distribution ;
for (Dict::const_iterator itr = dict.begin() ; itr != dict.end() ; ++ itr) {
  ++ distribution[itr->second] ; }
typedef map<size_t, size_t>::const_iterator Tmp_t ;
for (Tmp_t itr = distribution.begin() ; itr != distribution.end() ; ++ itr) {
  cout << itr->first << '\t' << itr->second << endl ; }
```

Ceci est cependant très peu efficace. Il n'est pas particulièrement coûteux de parcourir toutes les entrées d'un tableau associatif. En revanche, il est relativement coûteux de rechercher une clef dans un tel tableau. Il est beaucoup plus efficace, comme la clef est un entier naturel (qui varie de 1 à un nombre maximum qui reste à déterminer), d'utiliser un tableau dont les éléments servent à décompter les différentes occurrences. Si l'identificateur de ce tableau est, par exemple, ***distribution***, ***distribution[1]*** est le nombre de mots qui reviennent une fois, ***distribution[2]*** est le nombre de mots qui reviennent deux fois, etc. L'élément ***distribution[0]*** n'est pas utilisé.

Il faut dans un premier temps déterminer le nombre d'occurrences le plus élevé. Le code le plus immédiat est le suivant.

```
size_t max = 0 ;
for (Dict::const_iterator itr = dict.begin() ; itr != dict.end() ; ++ itr) {
  if ( itr->second > max ) {
    max = itr->second ; } }
```

La variable ***max*** est destinée à recevoir cette valeur maximale. Elle est initialisée à 0. Ensuite, le tableau associatif ***dict*** est parcouru en utilisant un ***const_iterator***. Cet itérateur est initialisé par la valeur ***dict.begin()***. L'incrémentation de cet itérateur permet de désigner l'élément suivant du conteneur. Ces éléments sont des ***Paire_dict***. Aussi la syntaxe « ***itr->second*** » désigne-t-elle le second élément de la paire, une variable de type ***size_t*** qui repère l'occurrence du mot dans le texte. La valeur maximale est déterminée de proche en proche. Si la valeur courante est strictement supérieure à ce maximum, le nouveau maximum est égal à cette valeur courante.

Ensuite, le vecteur ***distribution*** est convenablement dimensionné et initialisé. Ceci est l'affaire de la méthode ***assign***. Son premier argument donne la dimension du vecteur ; son second argument la valeur avec laquelle il convient d'initialiser chaque élément du vecteur. La variable ***distribution*** a été déclarée comme une variable globale. Il suffit ainsi de coder la ligne suivante.

```
distribution.assign(max+1, 0) ;
```

Attention, si le nombre maximum d'occurrences est, par exemple, 87, il convient de dimensionner le vecteur avec 88 éléments puisque l'indice 0 n'est pas utilisé. Enfin, pour renseigner le vecteur ***distribution***, il suffit d'itérer sur les éléments du tableau associatif ***dict***. Les deux lignes suivantes sont ainsi codées.

```
for (Dict::const_iterator itr = dict.begin() ; itr != dict.end() ; ++ itr) {
  ++ distribution[itr->second] ; }
```

La syntaxe « ***++ distribution[itr->second]*** », quand elle est appliquée à un ***vector***, se contente d'incrémenter l'élément dont l'indice est ***itr->second***. Si cet indice n'est pas compris dans les bornes du tableau, c'est une erreur (grave). Il est possible d'utiliser la méthode ***at(...)*** qui contrôle la validité de l'indice au prix d'une certaine inefficacité. On aurait alors codé les lignes suivantes.

```
for (Dict::const_iterator itr = dict.begin() ; itr != dict.end() ; ++ itr) {
  ++ distribution.at(itr->second) ; }
```

La STL incite à faire œuvre de « programmation fonctionelle », c'est-à-dire à utiliser les algorithmes de la STL et à appliquer des fonctions sur les éléments ainsi désignés. Par exemple, le calcul de la valeur maximale peut être obtenu en définissant, tout d'abord, un objet-fonction puis en utilisant l'algorithme ***for_each***. Cet objet-fonction encapsule la valeur maximale ; il définie l'opérateur ***operator()*** qui est appelé pour chaque élément et qui se charge du calcul du maximum et la méthode ***max()*** qui retourne cette valeur maximale. On code ainsi la définition d'un tel objet-fonction ***Calcul_max_occurrences***.

```
class Calcul_max_occurrences {
  size_t max_ ;
  public :
  Calcul_max_occurrences() : max_(0) {}
  void operator() (const Paire_dict & arg) {
    if (arg.second > max_) {
      max_ = arg.second ; } }
```

```
  size_t max() const { return max_ ; }
} ;
```

Le constructeur par défaut se contente d'initialiser l'attribut ***max_*** à `0`. La définition de l'objet-fonction est un peu verbeuse. L'utilisation de cet objet-fonction est en revanche très simple. Il suffit de coder la ligne suivante.

```
size_t max = for_each(dict.begin(), dict.end(), Calcul_max_occurrences()).max() ;
```

Le troisième argument de ***for_each*** est une instance anonyme de l'objet ***Calcul_max_occurence*** obtenue en utilisant le constructeur par défaut. C'est comme si, à la place de la « ***ma_fonction(0)*** », l'on codait « ***ma_fonction(int())*** ». En effet, ***int()*** est le constructeur par défaut pour une variable de type ***int*** et ce constructeur par défaut donne la valeur `0`. L'algorithme ***for_each*** retourne une référence sur son troisième argument. C'est la raison pour laquelle nous pouvons appliquer à cette valeur de retour la méthode ***max()***. On obtient ainsi la valeur du maximum qui est affectée à la variable ***max*** de type ***size_t***.

La STL nous offre toutefois l'algorithme ***max_element*** qui va me permettre de proposer une solution optimale : efficace et simple à mettre en œuvre. Cet algorithme est proposé en deux versions. La première version applique l'opérateur ***operator<*** entre le maximum courant et l'élément courant de la séquence d'entrée. La seconde version permet au programmeur d'indiquer le critère de comparaison. Une fonction (ou un objet-fonction) est alors le troisième argument de l'algorithme ***max_element***. Dans notre cas, la fonction de comparaison est codée comme l'était la fonction de comparaison que nous avons utilisée pour trier les entrées par ordre décroissant du nombre d'occurrences, sauf qu'il s'agit cette fois-ci de l'ordre croissant. La fonction ***cmp_max_occurrences*** est donc définie comme suit.

```
bool
cmp_max_occurrences(const Paire_dict & max, const Paire_dict & arg)
{
  return max.second < arg.second ;
}
```

La détermination du plus grand élément (plus grand au sens de la fonction de comparaison spécifiée) est alors très simple. On code la ligne suivante.

```
size_t max = max_element(dict.begin(), dict.end(), cmp_max_occurrences)->second ;
```

L'algorithme ***max_element*** retourne un itérateur sur l'élément le plus grand. Pour récupérer le nombre d'occurrences le plus élevé, il faut donc écrire « ***max_element(...)->second*** ».

La fonction de comparaison ***cmp_trier_occurrences***, utilisée pour trier par ordre décroissant le nombre d'occurrences, aurait pu être réutilisée en ayant recours à l'algorithme ***min_element***. La ligne suivante aurait été exactement équivalente à la ligne précédente.

```
size_t max = min_element(dict.begin(), dict.end(), cmp_trier_occurrences)->second ;
```

La « programmation fonctionnelle » encourage l'utilisation des algorithmes de la STL et la réutilisation de (petites) fonctions définies par le programmeur.

De la même veine, le code

```
for (Dict::const_iterator itr = dict.begin() ; itr != dict.end() ; ++ itr) {
  ++ distribution[itr->second] ; }
```

peut être remplacé par un couple algorithme/fonction. La fonction qui incrémente un élément de ***distribution*** peut être définie comme suit.

```
inline void
incrementer_par_occurrences(const Paire_dict & arg)
{
  ++ distribution[arg.second] ;
}
```

Ensuite, on utilise l'algorithme ***for_each*** sur la séquence d'entrée ***dict.begin()***, ***dict.end()***.

```
for_each(dict.begin(), dict.end(), incrementer_par_occurrences) ;
```

Un peu plus loin dans le programme, l'algorithme numérique ***accumulate*** est utilisé comme suit pour calculer la somme partielle des éléments du vecteur ***distribution*** à partir de l'élément `21`.

```
size_t borne_sup = min(max+1, size_t(21)) ;
...
size_t total = accumulate(distribution.begin()+borne_sup, distribution.end(), 0) ;
```

Le nombre total d'éléments dans ***distribution*** est égal à ***max+1***. Si le vecteur ne contient que `16` éléments, par exemple, l'expression ***min(max+1, size_t(21))*** est égale à `16` puisque la variable ***max*** est égale à `15`. Dans ce cas, la valeur de l'expression « ***distribution.begin()+ borne_sup*** » est exactement égale à ***distribution.end()*** et l'algorithme ***accumulate*** n'itère alors pas.

Il ne faut sans doute pas abuser des possibilités offertes par ce style de « programmation fonctionnelle ». Dans certains cas, le programme est plus expressif. En cette matière, C++ est handicapé du fait qu'il ne propose pas une syntaxe pour définir de manière anonyme « à la volée » une fonction. Par exemple, on voudrait, à la place des lignes suivantes,

```
for (Dict::const_iterator itr = dict.begin() ; itr != dict.end() ; ++ itr) {
  ++ distribution[itr->second] ; }
```

pouvoir disposer d'une syntaxe qui serait comme suit.

```
for_each(dict.begin(), dict.end(),
  {const Paire_dict & arg ; ++ distribution[arg.second] ;}) ;
```

La définition d'une fonction anonyme commencerait par une accolade ouvrante « *{* », la première instruction serait la déclaration des arguments de la fonction, les instructions suivantes constitueraient le corps de la fonction et ce dernier se terminerait par une accolade fermante « *}* ».

2 Le programme

Le programme du projet *« Lexicométrie »* figure ci-après. Il faut l'étudier avec soin et s'amuser avec.

```
1  #include <iostream>   // cin, cout et cerr
2  #include <fstream>    // ifstream
3  #include <sstream>    // istringstream
4  #include <iterator>   // istream_iterator
5  #include <string>     // Chaines de caractères de la STL.
6  #include <map>        // Tableaux associatifs de la STL.
7  #include <vector>     // Vecteurs de la STL.
8  #include <cctype>     // ispunct(...)
9  #include <algorithm>  // Algorithmes de la STL.
10 #include <numeric>    // Algorithmes numériques, comme accumulate(...).
11 #include <cmath>
12
13 using namespace std ;
14
15 // Taille minimum des mots.
16 const size_t TAILLE = 4 ;
17
18 // Type de la paire (mot, nombre d'occurrences du mot).
19 typedef pair<string, size_t> Paire_dict ;
20
21 // Type du map 'dict', mot et nombre d'occurrences du mot.
22 typedef map<string, size_t> Dict ;
23
24 // La variable 'dict' est globale pour être vue de la fonction 'remplir_dict'.
25 Dict dict ;
26
27 // Vecteur des différentes distributions calculées dans le projet.
28 vector<size_t> distribution ;
29
30 // Les caractères particuliers de notre belle langue sont inchangés, les caractères de
31 // de ponctuation sont remplacés par un espace ; ceux qui restent sont mis en minuscule.
32 inline char
33 recoder (const char & arg)
34 {
35   static string bizarre("ÆæŒœÀàÁáÂâÇçÈèÉéÊêËëÎîÏïÔôÖöÙùÛû") ;
36   if ( bizarre.find(arg) != string::npos ) {
37     return arg ; }
38   else if ( ispunct(arg) ) {
39     return ' ' ; }
40   else {
41     return tolower(arg) ; }
42 }
43 // Seuls les mots de plus de 'TAILLE' lettres sont inclus dans le map 'dict'.
44 inline void
45 remplir_dict(const string & arg)
46 {
47   if ( arg.size() > TAILLE ) {
48     ++ dict[arg] ; }
49 }
50 // Fonction de comparaison pour trier en ordre décroissant sur le nombre d'occurrences.
51 inline bool
52 cmp_trier_occurrences(const Paire_dict & arg1, const Paire_dict & arg2)
53 {
54   return arg1.second > arg2.second ;
55 }
56 // Fonction de comparaison pour le calcul du nombre d'occurrences maximum.
57 bool
58 cmp_max_occurrences(const Paire_dict & max, const Paire_dict & arg)
59 {
60   return max.second < arg.second ;
61 }
62 // Fonction pour établir la distribution par occurrence.
63 inline void
64 incrementer_par_occurrences(const Paire_dict & arg)
65 {
66   ++ distribution[arg.second] ;
67 }
68 // Fonction de comparaison pour le calcul de la taille maximale d'un mot.
69 bool
70 cmp_max_taille(const Paire_dict & max, const Paire_dict & arg)
71 {
72   return max.first.size() < arg.first.size() ;
73 }
74 // Fonction pour établir la distribution par taille.
75 inline void
76 incrementer_par_taille(const Paire_dict & arg)
77 {
78   ++ distribution[arg.first.size()] ;
79 }
80 int
```

```
main()
{
  {
    // La chaîne de caractères 'tout' contient la totalité du fichier. La syntaxe
    // 'string tout(...,...) ;' ne convient pas : c'est la déclaration de la fonction
    // 'total' qui retourne une 'string'. Les parenthèses qui ont été rajoutées ne
    // sont donc pas surnuméraires. L'utilisation d'un 'istream_iterator' ne convient
    // pas parce que le flux d'entrée est mis en forme (suppression des espaces, etc.).
    string tout((istreambuf_iterator<char>(cin)), istreambuf_iterator<char>()) ;

    // Transformation de toute la chaîne en recodant ses caractères au moyen de la
    // fonction 'recoder'.
    transform(tout.begin(), tout.end(), tout.begin(), recoder) ;

    istringstream mots(tout) ; // La chaîne devient un fichier en lecture en mémoire.
    // Remplissage du dict ; en entrée, les mots qui proviennent d'un itérateur
    // qui extrait les mots du fichier 'mots', en sortie le dict renseigné par
    // la fonction 'remplir'.
    for_each(istream_iterator<string>(mots), istream_iterator<string>(), remplir_dict) ;
    if ( dict.empty() ) {
      cerr << "Le texte ne contient aucun mot de plus de " << TAILLE << " lettres." ;
      exit(1) ; }
  } // Sortie du bloc : 'tout' et 'mots' sont détruits.

  // Impression des vingt premiers mots qui reviennent le plus souvent.
  // Les paires (mots, nombre d'occurrences) du dict sont recopiés dans un
  // vecteur.
  vector< Paire_dict > compte ;
  copy(dict.begin(), dict.end(), back_inserter(compte)) ;
  // Le vecteur est trié sur le nombre d'occurrences grâce à la fonction
  // 'cmp_trier_occurrences'.
  sort(compte.begin(), compte.end(), cmp_trier_occurrences) ;
  cout << "Les vingt premiers mots de " << TAILLE+1 << " lettres et plus qui "
    "reviennent le plus souvent." << endl ;
  for ( size_t i = 0 ; i < min(compte.size(), size_t(20)) ; ++ i ) {
    cout << compte[i].second << ' ' << compte[i].first << endl ; }

  // Distribution de la fréquence de l'apparition des mots.
  // Tout d'abord, on recherche la plus grande fréquence d'apparition.
  size_t max = max_element(dict.begin(), dict.end(), cmp_max_occurrences)->second ;
  // On ajuste la dimension du vecteur et on initialise chaque élément de celui-ci.
  distribution.assign(max+1, 0) ;
  // On renseigne 'distribution'.
  for_each(dict.begin(), dict.end(), incrementer_par_occurrences) ;
  cout << "Distribution de la fréquence de l'apparition des mots de " << TAILLE+1 <<
    " lettres et plus." << endl ;
  size_t borne_sup = min(max+1, size_t(21)) ;
  for ( size_t i = 1 ; i < borne_sup ; ++ i ) {
    if ( distribution[i] != 0 ) {
      cout << i << '\t' << distribution[i] << endl ; } }
  size_t total = accumulate(distribution.begin()+borne_sup, distribution.end(), 0) ;
  if ( total != 0 ) {
    cout << "de " << borne_sup << " à " << max << '\t' << total << endl ; }

  // Distribution de la fréquence de la longueur des mots ; traitement comparable.
  max = max_element(dict.begin(), dict.end(), cmp_max_taille)->first.size() ;
  distribution.clear() ; distribution.assign(max+1, 0) ;
  for_each(dict.begin(), dict.end(), incrementer_par_taille) ;
  cout << "Distribution de la fréquence de la longueur des mots de " << TAILLE+1 <<
    " lettres et plus." << endl ;
  for ( size_t i = TAILLE+1 ; i <= max ; ++ i ) {
    if ( distribution[i] != 0 ) {
      cout << i << '\t' << distribution[i] << endl ; } }
  return 0 ;
}
```

L'exécution de ce programme conduit aux sorties suivantes.

```
Les vingt premiers mots de 5 lettres et plus qui reviennent le plus souvent.
692 était
666 comme
663 avait
316 charles
236 madame
228 quand
227 cette
212 bovary
193 autre
184 quelque
168 homais
168 faire
166 alors
153 encore
143 jusqu
137 leurs
135 après
133 contre
130 rodolphe
129 trois
Distribution de la fréquence de l'apparition des mots de 5 lettres et plus.
```

1 6448
2 2165
3 1019
4 602
5 410
6 261
7 218
8 157
9 123
10 94
11 76
12 72
13 50
14 45
15 41
16 35
17 27
18 28
19 31
20 26
de 21 à 692 355
Distribution de la fréquence de la longueur des mots de 5 lettres et plus.
5 1213
6 1950
7 2240
8 2143
9 1778
10 1297
11 802
12 452
13 237
14 105
15 42
16 21
17 1
18 2