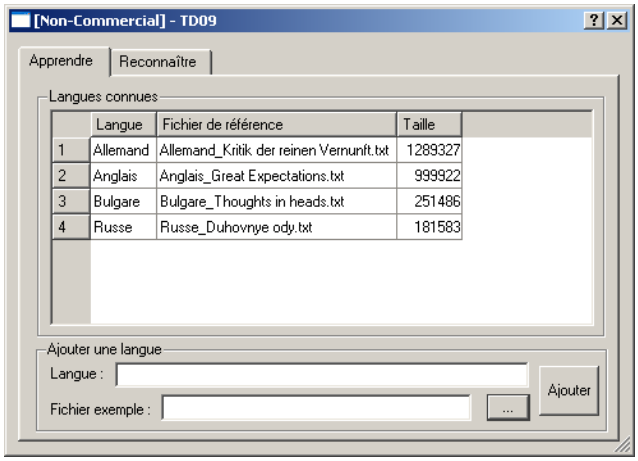
*C*entre *I*nformatique pour les *L*ettres
et les *S*ciences *H*umaines

TD 9 : Reconnaître la langue d'un texte

Déterminer dans quelle langue un texte est rédigé est une tâche dont l'importance s'accroît avec le nombre de documents électroniques accessibles, et qui constitue souvent une étape préalable à des traitements plus ambitieux.

Le programme que nous allons réaliser base sa décision sur une analyse des fréquences relatives des différents caractères composant le texte. Cette stratégie exige de disposer de points de comparaison, et notre programme va donc comporter deux facettes : il doit être capable d'une part d'apprendre à associer le nom d'une langue à une certaine distribution des fréquences des caractères et, d'autre part, de déterminer, parmi les langues connues, quelle est celle dont la distribution des fréquences des caractères est la plus proche de la distribution observée dans un fichier dont la langue est inconnue.



L'interface utilisateur du programme réalisé au cours du TD 9

1 - Analyse préalable

Outre la gestion de l'interface utilisateur et l'accès aux fichiers de données, le programme repose essentiellement sur l'établissement de descriptions statistiques des textes utilisés. Cet aspect du travail sera confié à une classe `CDescription`, dont nous allons, comme à l'accoutumé, découvrir progressivement les caractéristiques au cours de la mise au point de la classe de dialogue.

2 - Création du projet et dessin de l'interface

Créez, en suivant la procédure habituelle, un projet nommé TD09 ☐.

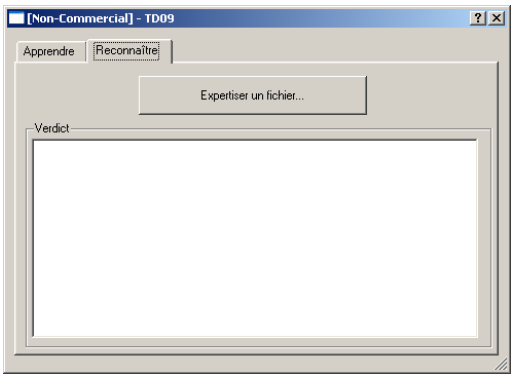
Placez sur le dialogue un `QTabWidget` (Menu Tools, catégorie Containers) ☐.

Sur la première page de ce `QTabWidget`, disposez :

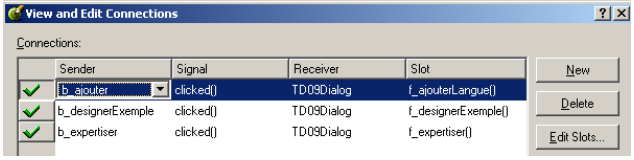
- une `QTable` (Menu Tools, catégorie Views) baptisée `tableauLangues` ☐ ;
- deux `QLineEdit` (Menu Tools, catégorie Input) respectivement nommées `nomLangue` et `exempleLangue` ☐ ;
- deux `QPushButton` (Menu Tools, catégorie Buttons) respectivement nommés `b_ajouter` et `b_designerExemple` ☐ ;
- deux `QGroupBox` et deux `QLabel` explicitant le rôle des différents widgets (cf. image ci-dessus) ☐.

Sur la seconde page du `QTabWidget`, disposez :

- un `QPushButton` nommé `b_expertiser` ☐ ;
- un `QTextEdit` (Menu Tools, catégorie Input) nommé `ecran` ☐ ;
- un `QGroupBox` donnant un titre au `QTextEdit` (cf. image ci-contre) ☐.



Créez trois slots nommés `f_ajouterLangue`, `f_designerExemple` et `f_expertiser`, et connectez-les aux boutons correspondants ☐.



3 - La classe de dialogue

Notre interface comporte une `QTable` et quelques instructions sont nécessaires pour attribuer des titres aux colonnes. Cette opération peut être effectuée une fois pour toutes, dès le lancement du programme. Les instructions correspondantes prennent donc place dans...

Le constructeur

```
1  TD09DialogImpl::TD09DialogImpl(QWidget* parent, const char* name, bool modal,
                                  WFlags f ): TD09Dialog( parent, name, modal, f )
2  {
3  tableauLangues->setNumRows(0);
4  tableauLangues->setNumCols(3);
5  QHeader * titres = tableauLangues->horizontalHeader();//on récupère le QHeader
6  titres->setLabel(0, "Langue");//on lui indique les titres des colonnes
7  titres->setLabel(1, "Fichier de référence");
8  titres->setLabel(2, "Taille");
9  }
```

Dans le fichier TD09DialogImpl.cpp, ajoutez les lignes *3-8* au constructeur de la classe ☐.

La fonction `f_designerExemple()`

Cette fonction ne peut pas entreprendre quoi que ce soit de significatif, car il n'est pas certain que, au moment où elle est invoquée, le nom de la langue correspondant à l'exemple désigné ait déjà été saisi. Elle se contente donc d'afficher le chemin du fichier désigné, ce qui, du même coup, rend celui-ci disponible pour la fonction `f_ajouterLangue()`.

```
1  void TD09DialogImpl::f_designerExemple()
2  {
3  exempleLangue->setText(QFileDialog::getOpenFileName());
4  }
```

Ajoutez la fonction `f_designerExemple()` à votre classe `TD09DialogImpl` ☐ et donnez lui le contenu suggéré ci-dessus ☐.

La fonction `f_ajouterLangue()`

Le rôle de cette fonction est d'effectuer les calculs nécessaires pour ajouter à la "base de connaissances" du programme la description de la nouvelle langue. Avant de se lancer dans cette opération, la fonction doit cependant s'assurer que certains cas triviaux sont filtrés. Il faut ainsi vérifier que l'utilisateur a bien spécifié un nom *(5-9)* et un fichier d'exemple *(10-15)* pour la nouvelle langue :

```
1  void TD09DialogImpl::f_ajouterLangue()
2  {
3  QString nom = nomLangue->text();
4  QString chemin = exempleLangue->text();
5  if(nom == "")
6     {
7     QMessageBox::warning(this, "TD09", "Donnez un nom à cette langue");
8     return;
9     }
10 QFile leFichier(chemin);
11 if(!leFichier.exists())
12    {
13    QMessageBox::warning(this, "TD09", "Veuillez désigner un fichier contenant "
                                         "un échantillon de " + nom);
14    return;
15    }
```

Il faut ensuite créer une instance de la classe `CDescription` qui servira à stocker les fréquences des différents caractères rencontrés dans le fichier d'exemple. Cette instance doit avoir connaissance du nom de la langue et du fichier d'exemple correspondant :

```
16  CDescription aAjouter;
17  if(! aAjouter.miseEnRoute(nom, chemin))
18     {
19     QMessageBox::warning(this,"TD09", "Le fichier " + chemin + " est inexploitable");
20     return;
21     }
```

1 : La classe `CDescription` doit comporter une fonction membre nommée `miseEnRoute()`dont les deux paramètres permettent de spécifier respectivement le nom de la langue décrite et le chemin d'accès au fichier exemple. Cette fonction renvoie un booléen indiquant si l'instance au titre de laquelle elle a été exécutée accepte la mission qui lui est proposée.

La "base de connaissances" de notre programme n'est qu'une vulgaire collection d'instances de `CDescription`. Comme le seul usage que nous allons en faire est de la parcourir pour trouver l'instance qui "ressemble" le plus à la description de notre fichier "mystère", une `QValueList` semble tout à fait suffisante.

Ajoutez à votre fichier TD09DialogImpl.h une ligne déclarant une variable membre de type `QValueList <CDescription>` portant le nom `lesLangues` .

Avant d'ajouter la description de la nouvelle langue à notre collection, il convient de vérifier que celle-ci ne comporte pas déjà une description d'une langue portant le même nom. Si c'est le cas, il faut éliminer cette ancienne description.

> La présence de deux descriptions différentes d'une même langue n'est pas à proprement parler inconcevable dans le contexte du présent projet. Il semble toutefois plus clair d'exiger que leur désignation reste unique ("Anglais 1" et "Anglais 2", par exemple).

```
   //on élimine une éventuelle entrée précédente pour cette langue
22 QValueList<CDescription>::Iterator uneLangue = lesLangues.begin();
23 while(uneLangue != lesLangues.end())
24    if((*uneLangue).nom() == nom)
25       uneLangue = lesLangues.remove(uneLangue);
26    else
27       ++uneLangue;
```

2 : Une `CDescription` doit être capable de révéler le nom de la langue qu'elle décrit.

Il suffit ensuite de demander à la nouvelle `CDescription` de procéder aux calculs requis *(28)*, puis de l'ajouter à notre collection :

```
   //on ajoute la nouvelle langue
28 QApplication::setOverrideCursor(waitCursor);
29 aAjouter.analyse();
30 lesLangues.append(aAjouter);
```

> L'analyse du fichier d'exemple peut prendre "un certain temps" (nous ne savons rien de sa taille...) et la ligne *27* donne au pointeur de la souris la forme qui indique qu'une opération longue est en cours (sous Windows, il s'agit par défaut d'un sablier).

3 : La classe `CDescription` doit comporter une fonction `analyse()` ordonnant à l'instance au titre de laquelle elle est exécutée de procéder à l'analyse du fichier d'exemple (qui doit lui avoir été préalablement désigné par un appel à la fonction `miseEnRoute()`).

> Pourquoi la fonction `miseEnRoute()` ne procède-t-elle pas directement à l'analyse du fichier d'exemple ? Parce qu'il s'agit d'une opération potentiellement très longue, et qu'il est préférable que de tels traitements ne soient entrepris que lorsqu'ils sont explicitement demandés.

La fonction `f_ajouterLangue()` doit enfin afficher la liste des langues connues, de façon à ce que l'utilisateur sache où il en est. Plutôt que d'essayer d'ajouter une ligne à `tableauLangues` (ce qui exigerait des mesures particulières dans le cas où l'une des entrées a été supprimée), il est préférable de vider la `QTable` *(30)* et d'en reconstituer intégralement le contenu en demandant aux `CDescription` de fournir elles-mêmes les renseignements nécessaires :

```
//on affiche la liste des langues connues
tableauLangues->setNumRows(0);
for(uneLangue = lesLangues.begin(); uneLangue!= lesLangues.end(); ++uneLangue)
   {
   int ligne = tableauLangues->numRows();
   tableauLangues->setNumRows(ligne + 1);
   tableauLangues->setText(ligne, 0, (*uneLangue).nom());
   QString c = (*uneLangue).chemin();
   tableauLangues->setText(ligne, 1, c.mid(c.findRev("/")+1, 100));
   tableauLangues->setText(ligne, 2, QString::number((*uneLangue).taille()));
   }
int colonne;
for(colonne = 0 ; colonne < tableauLangues->numCols() ; ++colonne)
   tableauLangues->adjustColumn(colonne); //règle la largeur des colonnes
nomLangue->setText("");
exempleLangue->setText("");
QApplication::restoreOverrideCursor();
}
```

4 : Une `CDescription` doit pouvoir dire quel fichier elle a analysé.

5 : Une `CDescription` doit pouvoir dire combien de caractères elle a analysé.

Les lignes *44-45* éliminent le risque qu'un utilisateur impatient procède à plusieurs ajout de la même langue en cliquant plusieurs fois de suite sur le bouton [Ajouter]. La ligne *44* rend à la souris son curseur normal (sous Windows, il s'agit par défaut d'une flèche). L'utilisation de ces fonctions exige la présence d'une directive `#include "qapplication.h"`.

Ajoutez la fonction `f_ajouterLangue()` à votre classe `TD09DialogImpl` ☐ et donnez lui le contenu suggéré ci-dessus ☐.

La fonction `f_expertiser()`

Cette fonction doit, elle aussi, commencer par filtrer les cas dans lesquels elle ne peut rien accomplir d'utile :

```
void TD09DialogImpl::f_expertiser()
{
if(lesLangues.isEmpty())
   {
   QMessageBox::warning( this,"TD09","Je ne connais aucune langue !");
   return;
   }
QString chemin = QFileDialog::getOpenFileName();
QFile leFichier(chemin);
if(!leFichier.exists())
   {
   QMessageBox::warning(this, "TD09", "Veuillez désigner un fichier contenant un texte");
   return;
   }
CDescription aIdentifier;
if(! aIdentifier.miseEnRoute("Inconnue", chemin))
   {
   QMessageBox::warning(this, "TD09", "Echec d'analyse de " + chemin);
   return;
   }
```

Une fois ces précautions prises, l'expertise proprement dite peut commencer. Il s'agit de parcourir notre collection de descriptions en cherchant celle qui se rapproche le plus du texte à expertiser : la langue (connue) de cette description est la réponse la plus vraisemblable que le programme puisse fournir à la question posée.

Le calcul de la distance entre les descriptions de deux textes sera assuré par une fonction membre de `CDescription`, mais notre programme va comporter un petit raffinement supplémentaire : il va analyser progressivement le texte mystérieux, et nous dire à quel moment il

a atteint sa conclusion finale quant à la langue de celui-ci. Ce raffinement impose de nouvelles contraintes à la classe `CDescription` :

6 : La classe `CDescription` doit comporter une fonction `estFini()` renvoyant un booléen qui indique si l'analyse est parvenue à la fin du texte.

7 : La fonction `CDescription::analyse()` doit accepter un paramètre lui indiquant combien de caractères elle doit traiter.

Sans oublier que

8 : La classe `CDescription` doit comporter une fonction `distance()` renvoyant une mesure de l'écart entre les statistiques contenues dans l'instance au titre de laquelle elle est invoquée et celles contenues dans l'instance qui lui est passée comme paramètre.

```
21 QApplication::setOverrideCursor(waitCursor);
22 QString verdict = "Erreur";
23 unsigned long changementAvis;
24 double distanceMini = DBL_MAX;
25 while (!aIdentifier.estFini())
26    {
27    aIdentifier.analyse(100); //lit une "tranche" du texte mystérieux
28    //cherche la langue qui ressemble le plus aux statistiques actuelles
29    QValueList<CDescription>::Iterator uneLangue;
30    for(uneLangue=lesLangues.begin(); uneLangue != lesLangues.end(); ++uneLangue)
31       {
32       double distance = aIdentifier.distance(*uneLangue);
33       if(distance < distanceMini)
34          {//uneLangue vient de battre le record de proximité
35          distanceMini = distance;
36          if( (*uneLangue).nom() != verdict)
37             changementAvis = aIdentifier.taille();//le verdict va changer
38          verdict = (*uneLangue).nom();
39          }
40       }
41    }
42 QString message = "\nMon avis: %1\nDécision stabilisée après analyse de %2 caractères\n";
43 ecran->setText(ecran->text() + chemin +
                        message.arg(verdict).arg(QString::number(changementAvis)));
44 QApplication::restoreOverrideCursor();
45 }
```

Remarquez, ligne 24, l'utilisation d'une constante (définie dans le fichier float.h) permettant de donner à `distanceMini` une valeur déraisonnablement grande qui garantit que la première distance calculée va être considérée comme un record de proximité.

Ajoutez la fonction `f_expertiser()` à votre classe `TD09DialogImpl` ☐ et donnez lui le contenu suggéré ci-dessus (sans oublier les `#include` nécessaires) ☐.

4 - La classe `CDescription`

Ajoutez une classe `CDescription` à votre projet ☐.

Le fichier description.h

La mise au point de la classe de dialogue nous a conduit à faire huit "promesses" concernant cette classe, et l'interface (ie. la section `public:`) de `CDescription` s'en trouve quasiment dictée :

```
1 class CDescription
2 {
3 public:
4    CDescription();
5    virtual ~CDescription();
```

Les lignes 4 et 5 ont été générées par Visual C++ lorsque vous avez créé la classe. Elles déclarent un constructeur et le destructeur de la classe (cf. Leçon 10). Si elles vous gênent, vous pouvez les effacer, mais il vous faudra alors effacer aussi les définitions (vides) des fonctions correspondantes que Visual C++ a placées dans le fichier description.cpp.

```
6    bool miseEnRoute(QString nomFich, QString cheminFich);        //point 1
7    QString nom() const {return m_nom;}                           //point 2
```

Comme doivent le faire toutes les fonctions qui ne font que communiquer à l'appelant une information concernant l'état de l'instance au titre de laquelle elles sont appelées (sans modifier cet état), la fonction `nom()` affiche clairement cette caractéristique en se déclarant constante.

Ces fonctions sont généralement très brèves, et peuvent donc être définies en même temps que la classe. Dans le cas présent, il est clair que `nom()` a besoin que `miseEnRoute()` ait stocké le nom de la langue dans une variable membre.

```
8    void analyse(unsigned long longueur = ULONG_MAX);             //points 3 et 7
```

L'utilisation d'un paramètre de type "entier long positif" permet de placer la valeur maximale admissible aussi haut qu'il est possible de le faire sans complications excessives (et bien inutiles dans ce cas). Comme dans le de la variable `distanceMini` rencontrée plus haut, une constante prédéfinie permet de donner à ce paramètre une valeur par défaut maximale, qui garantit l'analyse de l'intégralité du fichier si l'appelant n'a spécifié aucune `longueur`. La même fonction tient donc à la fois les promesses 3 et 7.

```
9    QString chemin() const {return m_chemin;}                     //point 4
10   unsigned long taille() const {return m_nbCarAnalyses;}       //point 5
```

Deux nouvelles variables membres s'imposent donc. La `taille()` dont il s'agit ici n'est pas celle du fichier, mais bien le nombre de caractères utilisés pour établir les statistiques actuelles.

```
11   bool estFini() const {return m_texteAAnalyser.isEmpty();}     //point 6
```

Derrière cette ligne se cache une décision fondamentale : nous allons placer le contenu du fichier dans une variable membre (une `QString`). Après chaque analyse partielle, nous retirerons du `texteAAnalyser` les caractères nouvellement pris en compte dans les statistiques. L'analyse sera donc bien terminée lorsque la `QString` en question sera vide.

```
12   double distance (const CDescription & unAutre) const;        //point 8
```

Les `CDescription` sont des objets volumineux (elles contiennent une fréquence pour chacun des caractères apparaissant dans le texte analysé et une copie du texte non encore analysé...). Il faut donc éviter de transmettre une valeur de ce type à une fonction, ce qui l'obligerait à créer une instance locale pour y copier toutes ces données. L'utilisation d'une référence à un objet constant évite cette copie sans pour autant permettre à la fonction de modifier un objet qui ne lui appartient pas.

La partie `protected:` de `CDescription` comporte évidemment la déclaration des variables membre dont nous venons de découvrir la nécessité :

```
13   protected:
14      QString m_nom;
15      QString m_chemin;
16      double m_nbCarAnalyses;
17      QString m_texteAAnalyser;
```

Il faut aussi prévoir le stockage des statistiques. Etant donné qu'il s'agit de mémoriser un nombre d'occurrences pour chaque caractère, une `QMap` semble tout à fait adaptée :

```
18      QMap <QChar, unsigned long> m_frequence;
19   };
```

Les clés de cette `QMap` ne sont pas des `char`, mais des `QChar`. Outre la simplification que ceci promet (les `QString` sont composées de `QChar`, et non de `char`), il serait dommage de se priver de l'indépendance que les classes Qt nous offrent vis à vis du type de codage utilisé dans les fichiers (ASCII, Unicode sur 8 ou 16 bits...). Grâce à cette indépendance, notre programme va en effet être capable de reconnaître la langue utilisée dans un fichier, même lorsque le seul exemple de cette langue qui lui a préalablement été montré utilisait un codage différent.

La fonction `miseEnRoute()`

Cette fonction assure la "pseudo-initialisation" des variables membre *(3-6)* et assure la lecture du fichier *(7-11)* :

```
1  bool CDescription::miseEnRoute(QString n, QString chemin)
2  {
3  m_nom = n;
4  m_chemin = chemin;
5  m_frequence.clear();
6  m_nbCarAnalyses = 0;
7  QFile data(m_chemin);
8  if(! data.open(IO_ReadOnly | IO_Translate))
9     return false;
10 QTextStream leFichier(&data);
11 m_texteAAnalyser = leFichier.read();
12 return true;
13 }
```

> La lecture *(11)* de l'intégralité du fichier dans une `QString` n'est pas généralement recommandable, car elle suppose implicitement que le fichier n'est pas démesurément grand. Dans le cas présent, les fichiers "naturels" ne poseront aucun problème (même un roman très long ne fera jamais plus de quelques millions de caractères). Il pourrait en aller autrement, bien sûr, en cas de création d'un fichier spécialement destiné à faire des statistiques, où seraient concaténés des milliers de textes d'origines diverses.

Ajoutez cette définition de la fonction `miseEnRoute()` à votre fichier description.cpp ☐.

La fonction `analyse()`

```
1  void CDescription::analyse(unsigned long nbCarAAjouter)
2  {
3  int position = 0;
4  while (position < nbCarAAjouter && position < m_texteAAnalyser.length() )
5     {
6     QChar aComptabiliser = m_texteAAnalyser[position];
7     ++m_frequence[aComptabiliser];
8     ++position;
9     }
10 m_nbCarAnalyses += position;
11 m_texteAAnalyser.remove(0, position);
12 }
```

> Attention, aux lignes *10* et *11*, c'est bien la variable `position` qu'il faut utiliser, et non `nbCarAAjouter`. En effet, cette dernière ne contient le nombre de caractères effectivement traités que si le `texteAAnalyser` était encore assez long pour satisfaire complètement la demande d'analyse.

Ajoutez cette définition de la fonction `analyse()` à votre fichier description.cpp ☐.

La fonction `distance()`

La distance entre deux séries ordonnées de nombres peut être calculée en faisant la somme des carrés des écarts. Ainsi, la distance entre 2, 3, 1 et 1, 5, 1 sera $(2-1)^2 + (3-5)^2 + (1-1)^2 = 1 + 4 + 0$, soit 5.

Dans le cas des `CDescription`, l'application de ce principe doit tenir compte de deux particularités prévisibles : les deux textes analysés ne sont sans doute pas de la même longueur, et certains caractères n'apparaissent peut-être que dans l'un des deux textes.

La question de la longueur des textes est facilement résolue en considérant les fréquences relatives des caractères plutôt que leurs fréquences absolues.

La question des "caractères manquants" impose de parcourir les deux collections de fréquences. Etant donné que ces deux collections sont "read only" (celle de l'instance au titre de laquelle la fonction est exécutée ne peut être modifiée parce que la fonction est déclarée constante, et celle de

l'autre instance ne peut être modifiée car la fonction y accède via une "référence à un objet constant"), leur parcours exige un "itérateur sur objet constant" :

```
1  double CDescription::distance(const CDescription & autre)const
2  {
3  double distanceCalculee = 0;
4  QMap<QChar, unsigned long>::ConstIterator it;
   //on commence par les caractères qui apparaissent chez nous
5  for(it = m_frequence.begin() ; it != m_frequence.end() ; ++it)
6     {
7     double ecart = it.data() / double(taille());
8     if(autre.m_frequence.contains(it.key())) //il apparait aussi chez l'autre
9        ecart -= autre.m_frequence [it.key()] / double(autre.taille());
10    distanceCalculee += ecart * ecart; //on fait la somme des carrés des écarts
11    }
```

Si un caractère n'apparaît pas dans l'autre texte, sa fréquence y est nulle et l'écart est donc égal à sa fréquence (relative) dans le premier texte *(7)*. Si le caractère apparaît dans les deux textes, il faut en revanche calculer *(9)* la différence entre ses deux fréquences (relatives).

```
   //il ne faut pas oublier ceux qui apparaissent chez l'autre mais pas chez nous
12 for(it = autre.m_frequence.begin() ; it != autre.m_frequence.end() ; ++it)
13    {
14    double ecart = it.data() / double(autre.taille());
15    if(!m_frequence.contains(it.key()))
16       distanceCalculee += ecart * ecart;
17    }
18 return distanceCalculee;
19 }
```

Lors du parcours des fréquences observées dans l'autre texte, seuls nous intéressent les caractères qui n'apparaissent pas dans le premier *(15)*. Par définition, l'écart correspondant est donc égal à la fréquence (relative) du caractère dans le second texte *(14)*.

Ajoutez cette définition de la fonction `distance()` à votre fichier description.cpp ☐.

Vous pouvez maintenant compiler le programme ☐ et, si vous n'avez oublié aucune des (nombreuses) directives `#include` nécessaires, procéder à quelques tests.

Vous pouvez trouver ici ou là des textes rédigés dans diverses langues.