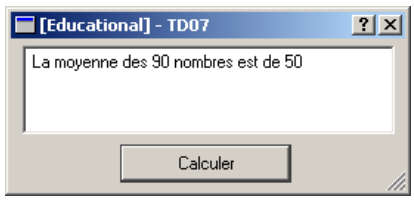
Centre Informatique pour les Lettres et les Sciences Humaines

TD 7 : Calculs sur des données numériques contenues dans un fichier

L'objectif du TD 07 est d'illustrer l'utilisation des techniques permettant à un programme de manipuler des fichiers de données. Le programme que nous allons réaliser n'effectue qu'un traitement très élémentaire : le calcul de la moyenne d'une série de nombres contenue dans un fichier texte.



L'interface utilisateur du programme réalisé au cours du TD 07

1 - Création du projet et dessin de l'interface

En vous inspirant de la procédure décrite dans le TD 3, créez un projet nommé TD07 ☐.

Eliminez tous les widgets présents dans le dialogue, et remplacez les par un textEdit (menu "Tools", catégorie "Input") et par un pushButton (menu Tools, catégorie "Buttons") ☐.

Rebaptisez le textEdit `affichage` et le bouton `b_calculer` ☐.

Créez un slot nommé `f_calculer()` et associez-lui l'événement `clicked()` du bouton ☐.

Enregistrez votre travail (Menu "File", commande "Save") ☐, puis revenez à Visual C++ ☐ et procédez aux modifications de rigueur sur la fonction `main()`, de façon à ce qu'elle instancie la classe `TD07DialogImpl` ☐.

2 - Préparation du fichier de données

La principale difficulté de mise au point des programmes utilisant des fichiers de données réside dans le fait que leur bon fonctionnement ne dépend pas seulement de leur exactitude, mais aussi du respect par les fichiers de données du format attendu par le programme. Pour éviter les surprises, nous allons dans un premier temps traiter une série de données "propre" :

```
64.18   8.34   3.95  15.00  65.80   7.82  99.52  62.32  21.69  75.04 55.21 87.94 26.41 56.60  8.40
99.41  61.34  12.47  37.71  34.43  90.47  59.40   4.59  56.74  82.45 51.15 39.98 74.73 98.90 49.13
36.35  35.72  59.14  10.58  26.59  92.21  48.28  49.21  42.36  62.71 70.88 45.44 82.61 47.75 50.70
40.32  67.22  50.42  98.39  22.24   1.68  60.96  95.40  97.64  28.33 51.92 91.20  6.40 22.57 68.96
15.92   9.64  40.52  72.28  56.60  40.88  93.21   7.30  67.12  39.91 79.80  3.76 71.28 25.57 77.66
28.06  59.42   4.62  32.12  13.81  49.62  82.26  66.27  55.96  83.55 22.42 42.53 34.28 80.47 67.86
```

Vous pouvez trouver ici un fichier nommé listing_TD07.txt contenant ces données, soit 6 lignes composées chacune de 15 nombres séparés par des tabulations.

Enregistrez ce fichier dans votre dossier projet ☐.

> Si vous n'arrivez pas à accéder au fichier proposé, vous pouvez en créer un de toutes pièces à l'aide d'un éditeur de texte quelconque. Veillez simplement à respecter l'organisation en 6 lignes de 15 colonnes, et à l'enregistrer au format texte.

3 - Ecriture du code

Ajoutez à la classe `TD07DialogImpl` une fonction nommée `f_calculer()` ☐.

Cette fonction doit tout d'abord ouvrir le fichier de données et s'assurer qu'un message intelligible est adressé à l'utilisateur si cette ouverture s'est avérée impossible. Le message émis dans ce type de circonstances devrait toujours mentionner le chemin d'accès utilisé lors de la tentative d'ouverture de fichier qui a échoué. Il est également préférable que ce message mentionne le nom du programme concerné. En effet, l'exécution d'un programme n'est pas toujours déclenchée explicitement par l'utilisateur mais peut être provoquée par d'autres programmes. L'utilisateur peut donc très légitimement n'avoir aucune idée de quel est le programme qui est en train d'essayer d'ouvrir un fichier.

> Vos messages ont peu de chances d'être réellement utiles si l'utilisateur ne sait ni qui les émet ni de quoi ils parlent. Un message tel que celui présenté ci-contre est une proclamation d'incompétence grave de la part du programmeur qui en est responsable.

Les premières lignes de la fonction seront donc :

```
1  void TD07DialogImpl::f_calculer()
2  {
3  QFile fichier("listing_TD07.txt");
4  bool toutVaBien = fichier.exists();
5  if(toutVaBien)
6     toutVaBien = fichier.open(IO_ReadOnly | IO_Translate);
7  if(! toutVaBien)
8     {
9     QMessageBox::critical(0,"TD07.exe", "Impossible d'ouvrir " + fichier.name());
10    return;
11    }
   //à suivre…
```

Une fois le fichier ouvert, il peut être utilisé comme source de données pour notre calcul. Bien entendu, aucun calcul ne peut être effectué directement sur le contenu du fichier (le processeur ne peut manipuler que des informations représentées en mémoire, cf. Leçon 1). Il faut donc que chaque valeur contenue dans le fichier soit d'abord transférée dans une variable (grâce à l'opérateur d'extraction) pour pouvoir être utilisée.

Il serait, évidemment, possible de créer une variable pour chacune des données à traiter. Cependant, nous savons que le fichier contient 90 valeurs, et l'idée de définir 90 variables ne semble guère attrayante.

Fort heureusement, le calcul de la moyenne n'exige pas que toutes les valeurs soient simultanément représentées en mémoire : si on ajoute *(20)* à une première variable initialement nulle *(14)* le contenu d'une seconde variable qui vient de recevoir une donnée provenant du fichier *(19)*, cette seconde variable redevient disponible pour recevoir une nouvelle donnée en provenance du fichier, qui sera à son tour ajoutée à la première variable. Cette première variable contient donc la somme des données lues, et, une fois le fichier épuisé, il suffit de la diviser par le nombre de données pour obtenir la `moyenne` recherchée *(22)*.

```
   //suite
12 QString message = "La moyenne des 90 nombres est de %1";
13 QTextStream source(&fichier);
14 double somme = 0;          //première variable
15 double valeurExtraite;     //seconde variable
16 int n;
17 for (n = 0 ; n < 90 ; ++n)
18    {
19    source >> valeurExtraite;
20    somme += valeurExtraite;
21    }
22 double moyenne = somme / 90;
23 affichage->setText(message.arg(moyenne));
24 } //fin de la fonction f_calculer()
```

Si, par ailleurs, vous insérez en tête de fichier les directives

```
#include "qfile.h"
#include "qmessagebox.h"
#include "qtextEdit.h"
```

vous pouvez compiler et exécuter votre programme.

Remarquez le recours à la fonction `arg()` pour injecter dans le texte affiché la représentation textuelle de la valeur de la moyenne *(23)*. Il aurait été possible, mais moins judicieux, d'écrire :

```
QString message = "La moyenne des 90 nombres est de ";
//calculs…
affichage->setText(message + QString::number(moyenne));
```

4 - Prolongements

L'extrême simplicité de mise au point de notre programme s'accompagne malheureusement de limitations qui le rendent d'une utilité pratique absolument nulle. Si on laisse de côté le calcul réalisé (notre propos n'est pas de développer un logiciel de statistiques), il reste un certain nombre de problèmes du point de vue qui nous intéresse, à savoir l'utilisation de fichiers de données.

- Le programme n'est capable de travailler que sur un fichier de données particulier.

  Pour être utilisable, le programme devrait offrir la possibilité de désigner le fichier contenant les données dont nous voulons calculer la moyenne. Modifier le texte source et recompiler le programme chaque fois qu'un nouveau fichier doit être traité n'est pas très pratique...

- Le programme n'est capable de traiter que des séries de 90 valeurs.

  Si on envisage d'appliquer le traitement sur divers fichiers, il y a fort à parier que, tôt au tard, nous rencontrerons des séries d'une longueur différente. Une fois encore, modifier le texte source et recompiler le programme n'est une méthode réellement acceptable.

- Le programme ignore l'organisation des données en un tableau de 6 lignes de 15 colonnes.

  Si les données sont présentées ainsi, c'est sans doute parce que cette organisation à un rapport avec la signification qu'ont les valeurs. Il serait donc préférable que le programme puisse en tenir compte (pour calculer séparément la moyenne de chacune des lignes, par exemple).

- Le résultat obtenu est simplement affiché à l'écran.

  Tant que notre programme ne calcule qu'une seule moyenne, cette limitation n'est pas très gênante. En revanche, si le programme devient capable de produire les moyennes de chacune des lignes d'un tableau en comportant des centaines, il serait plus pratique de pouvoir créer un fichier contenant ces résultats.

Désigner le fichier à traiter

Modifiez la fonction `f_calculer()` pour qu'elle commence par proposer à l'utilisateur un dialogue permettant à celui-ci de désigner le fichier contenant les données ☐.

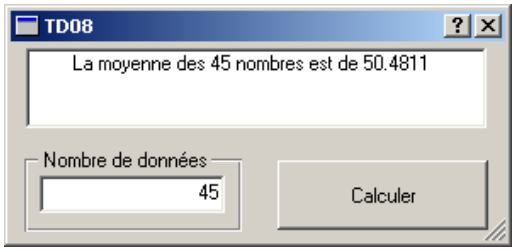
Vérifiez que la compilation ☐ et l'exécution ☐ du programme se déroulent comme prévu.

Traiter des séries de données d'une longueur quelconque

Les techniques envisageables pour rendre notre programme capable de s'adapter au nombre de données présentes dans le fichier peuvent être classées en quatre catégories.

Demander à l'utilisateur d'indiquer le nombre de données

Modifiez l'interface utilisateur du programme, de façon à y introduire un lineEdit dans lequel l'utilisateur pourra fournir l'information requise ☐.



N'oubliez pas d'enregistrer votre travail avant de revenir à Visual C++

Modifiez la fonction `f_calculer()` pour qu'elle tienne compte du nombre de données indiqué par l'utilisateur ☐.

Si le lineEdit s'appelle `saisieNb`, le nombre spécifié par l'utilisateur est donné par l'expression

```
saisieNb->text().toInt()
```

Vérifiez que votre programme est capable de calculer la moyenne des premiers nombres du fichier (50.4811 dans le cas des 45 premières valeurs de la série proposée précédemment) ☐.

La simplicité de mise en œuvre de cette approche la rend séduisante aux yeux de certains programmeurs, mais il faut savoir que les applications ainsi conçues sont généralement peu appréciées des utilisateurs, car elles exigent de ceux-ci qu'ils fournissent le nombre de données à un moment où ils n'ont pas nécessairement cette information présente à l'esprit. Ce manque de confort d'utilisation s'accompagne, en outre, d'un important risque d'erreur.

Remarquez aussi que, lorsque toutes les précautions sont prises pour vérifier que la saisie du nombre a bien été effectuée et que le texte présent dans le lineEdit correspond bien à une valeur numérique admissible, la "simplicité de mise en œuvre" de cette approche n'est plus qu'un souvenir assez lointain...

Utiliser un fichier dont la première ligne indique le nombre de données

Cette approche est, elle aussi, très facile à programmer : il suffit d'extraire le nombre de valeurs du fichier avant d'entrer dans la boucle qui extrait les données proprement dites.

> Le nombre de données doit figurer dans le fichier de données, ce qui est à la fois un avantage (si c'est l'utilisateur lui-même qui prépare le fichier, il a les données sous les yeux et peut donc savoir combien il y en a) et un inconvénient (si le fichier de données est créé par un autre programme, il faut que celui-ci respecte la convention). Utiliser cette méthode laisse deux sources d'erreurs principales : en cas d'ajout ou de suppression de données à l'aide d'un éditeur de texte, il ne faut pas oublier de mettre l'effectif à jour et, lorsque le fichier de données est traité par un autre programme qui n'utilise pas la même méthode, le nombre de valeurs risque fort d'être pris pour la première d'entre-elles, ce qui conduira à un résultat faux

Modifiez l'interface du programme, la fonction `f_calculer()` et le fichier de données, de façon à utiliser cette méthode ☐.

Vérifiez que votre programme trouve toujours les mêmes résultats, aussi bien pour une série complète ☐ que lors d'un calcul limité aux premières valeurs présentes dans le fichier ☐.

Utiliser un fichier comportant une "sentinelle"

Cette méthode repose sur la présence, à la fin de la série, d'une valeur particulière (la "sentinelle"), que le programme interprétera comme une sorte de "point final", et non comme une donnée.

Cette approche est un peu plus difficile à programmer que les précédentes, car elle nécessite qu'un test soit effectué avant que la `valeurExtraite` soit ajoutée à la somme. L'échec de ce test doit entraîner la fin de la boucle d'extractions des données, et chaque passage dans celle-ci doit s'accompagner de la mise à jour d'un "compteur de valeurs".

> Les programmes ainsi conçus présentent le double avantage d'être moins vulnérables en cas d'ajout ou de suppression de données (puisqu'il n'y a pas d'effectif à mettre à jour) et d'utiliser des fichiers de données compatibles avec les programmes utilisant une méthode de la première catégorie (si le programme "sait" déjà combien de données il doit lire, il ignorera simplement la dernière valeur présente dans le fichier). Le plus gros défaut de cette méthode est qu'elle repose sur l'existence d'une valeur dont on sait par avance qu'elle ne figurera jamais dans la série (non seulement le programme ne la prendrait pas en compte dans son calcul, mais sa rencontre mettrait prématurément fin à l'extraction des données). Cette technique n'est donc pas très générale, mais elle peut convenir lorsqu'on connaît la nature des données traitées (s'il s'agit de durées, par exemple, une valeur négative est inconcevable).

Modifiez la fonction `f_calculer()` et le fichier de données, de façon à utiliser comme sentinelle la valeur -1 ☐.

Vérifiez que votre programme trouve toujours les mêmes résultats, aussi bien pour une série complète ☐ que lors d'un calcul limité aux premières valeurs présentes dans le fichier ☐.

Que se passe-t-il si la valeur sentinelle n'apparaît pas dans le fichier de données ?

> Utilisez la commande "Stop debugging" du menu "Debug" pour arrêter le programme...

Détecter la fin du fichier

Modifiez la fonction `f_calculer()` de façon à ce que le test arrêtant la boucle d'extraction des données utilise la fonction `atEnd()` ☐.

Vérifiez que votre programme trouve toujours la même moyenne pour l'ensemble des données contenues dans le fichier ☐.

Si le programme semble extraire plus de valeurs qu'il n'en existe, vérifiez que votre fichier ne se termine pas par des lignes vides (difficiles à remarquer dans un éditeur de textes...)

Pour ne pas être perturbé par la présence de lignes vides, le programme doit extraire les données en direction d'une `QString` et vérifier la taille de celle-ci avant de la convertir en un `double`.

Modifiez la fonction `f_calculer()` de façon à ce qu'elle procède ainsi ☐ et vérifiez le résultat produit par votre programme en cas d'introduction de lignes vides à différents endroits dans le fichier de données ☐.

Les programmes capables de détecter eux-mêmes l'épuisement des données sont agréables à utiliser, puisque la longueur de la série ne vient polluer ni l'interface utilisateur (comme dans le cas de la première méthode) ni le fichier de données (comme dans le cas des deuxième et troisième méthodes). Il est cependant conseillé de toujours leur faire afficher le nombre de données qu'ils ont lues et traitées, car le confort qu'ils offrent peut donner une trompeuse impression d'infaillibilité.

Tenir compte de l'organisation lignes/colonnes

Lorsqu'on utilise l'opérateur d'extraction pour attribuer à une variable une valeur provenant d'un fichier, on ne dispose d'aucun moyen permettant de déterminer si la séquence de caractères représentant cette valeur dans le fichier s'achève par un passage à la ligne, une tabulation, ou tout autre caractère de séparation. Si le programme doit tenir compte de l'organisation en lignes, il lui faut donc procéder autrement.

La classe `QTextStream` offre, nous l'avons vu, une fonction `readLine()` qui permet, comme son nom l'indique, de lire l'intégralité d'une ligne (ie. tous les caractères non encore extraits, jusqu'au prochain saut de paragraphe). La fonction `f_calculer()` pourrait donc procéder ainsi :

```
//la première partie de la fonction reste inchangée
12 QString message = "La moyenne des %1 nombres est de %2";
13 QTextStream source(&fichier);
14 double somme = 0;
15 int nbValeurs = 0;
16 while (!source.atEnd() )
17    {
18    QString ligneExtraite = source.readLine();
19    if(! ligneExtraite.isEmpty())
20       traiteLigne(ligneExtraite, somme, nbValeurs);
21    }
22 double moyenne = somme / nbValeurs;
23 affichage->setText(message.arg(nbValeurs).arg(moyenne));
24 } //fin de la fonction f_calculer()
```

Une boucle contrôlée par la fin du fichier *(16-21)* procède à la lecture des données ligne par ligne. Les lignes non vides *(19)* sont confiées *(20)* à une fonction nommée `traiteLigne()` qui doit mettre à jour les variables `somme` et `nbValeurs`, en fonction des données présentes dans la ligne.

La fonction `traiteLigne()` devant modifier les valeurs contenues dans `somme` et `nbValeurs`, il faut que ses deuxième et troisième paramètres soient des références. Elle sera donc définie ainsi :

```
1  void TD07DialogImpl::traiteLigne(QString ligne, double & s, int & nb)
2  {
3  QTextStream uneLigne(&ligne, IO_ReadOnly);
4  QRegExp separateurs = "[ \t]";
5  int nbVal = 1 + ligne.contains(separateurs);
6  int n;
7  for (n = 0 ; n < nbVal ; ++n)
8     {
9     double valeurExtraite;
10    uneLigne >> valeurExtraite;
11    s += valeurExtraite;
12    ++nb;
13    }
14 }
```

La première caractéristique remarquable de cette fonction est l'usage qu'elle fait de la classe `QTextStream`. La variable `uneLigne` n'est en effet pas connectée *(3)* à un fichier, mais à une `QString` (celle qui contient la ligne transmise par `f_calculer()`). Une fois cette connexion établie, l'extraction des données est faite *(10)* comme si `uneLigne` était connectée à un fichier.

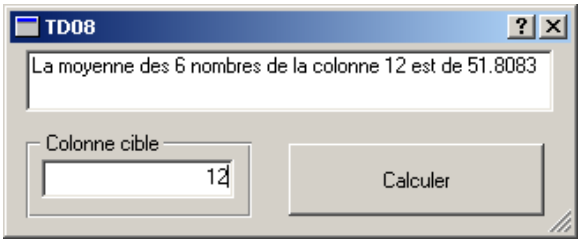
La seconde particularité de la fonction `traiteLigne()` est qu'elle utilise le dénombrement des "occurrences" d'une expression régulière dans une `QString`. La quantité ainsi obtenue correspond au nombre de séparateurs (espaces ou tabulations) figurant dans la ligne, quantité inférieure d'une unité au nombre de données contenues dans la ligne. Du fait de cette utilisation d'une expression régulière, la compilation de la fonction `traiteLigne()` n'est possible que si une directive `#include "qregexp.h"` figure en début de fichier.

Modifiez la fin de la fonction `f_calculer()`, conformément au code présenté ci-dessus ☐.

Ajoutez à la classe `TD07DialogImpl` une fonction membre nommée `traiteLigne()` et définissez celle-ci comme suggéré précédemment ☐.

Vérifiez que votre programme peut être compilé sans erreurs ni avertissements ☐ et que son exécution donne toujours le même résultat ☐.

Modifiez l'interface graphique et la définition de la fonction `traiteLigne()`, de façon à ce que le programme effectue le calcul de la moyenne des valeurs figurant dans une colonne choisie par l'utilisateur (cf. exemple ci-contre) ☐.



Attention : pour un utilisateur "normal", la première colonne est la colonne 1, et non la colonne 0...

Souvenez-vous qu'il n'est pas possible de changer l'ordre de lecture des données : pour obtenir la prochaine valeur qui vous intéresse, vous devez nécessairement extraire toutes celles qui la précèdent, même s'il s'agit de valeurs dont vous n'avez pas besoin pour votre calcul. Il ne faut donc pas remettre en cause l'organisation générale de la fonction `traiteLigne()`, mais simplement rendre conditionnels les traitements effectués par les lignes *11* et *12*.

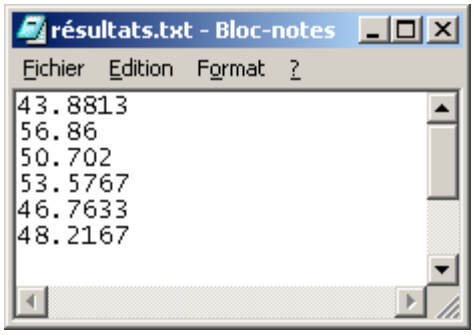
Enregistrer les résultats dans un fichier

Ajoutez un bouton "Moyennes des lignes" à l'interface utilisateur du programme ☐.

Ajoutez à la classe `TD07DialogImpl` une fonction exécutée lorsque l'utilisateur clique sur ce bouton ☐.

Définissez cette fonction de façon à ce qu'elle place dans un fichier nommé résultats.txt les moyennes de chacune des lignes non vides rencontrées dans le fichier de données ☐.

Si vous utilisez les données proposées page 2, le contenu du fichier résultats.txt devrait apparaître ainsi lorsque ce fichier est ouvert avec le bloc note :



5 - Qu'avons-nous appris ?

1 - Sous sa forme rudimentaire, l'utilisation d'un fichier de données n'ajoute que quelques minutes à la durée de mise au point d'un programme.

2 - Un programme qui se contente d'une gestion rudimentaire des fichiers de données est BEAUCOUP moins agréable à utiliser qu'un programme qui fait ça proprement.

3 - S'il faut "faire propre", la mise au point de la gestion des fichiers par le programme peut prendre plus de temps que la mise au point du reste du programme.