Centre Informatique pour les Lettres et les Sciences Humaines

TD 6 : Un concordancier

1 - Réflexions préliminaires 2

2 - Création du projet et dessin de l'interface 2

3 - Ecriture du code 3

Préparation du tableau de résultats 3
La fonction `f_chercher()` 4
Préparatifs 4
Recherche 5
Affichage : récupération des fragments de texte concernés 6
Affichage : insertion des fragments de texte dans le tableau 6
Finitions 7

4 - Prolongements 7

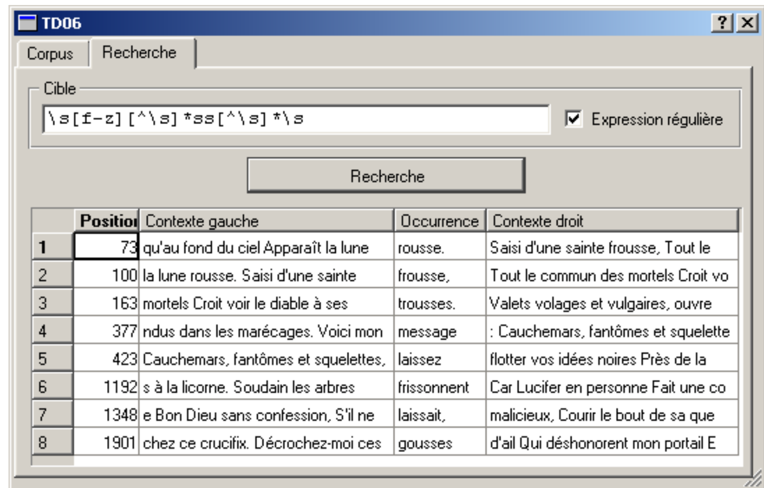
Exercices 7
Montrer une occurrence particulière dans le texte intégral 8
Détection de l'évènement "double-clic" dans une ligne du tableau 8
Analyse 8
Ajout de deux colonnes au tableau de résultats 9
Modification de la fonction `f_chercher()` 9
La fonction `f_clicSurOccurrence()` 10

5 - Qu'avons nous appris ? 10

Le programme réalisé au cours de ce TD est un *concordancier*, c'est à dire un outil permettant de mettre en évidence toutes les apparitions, dans un texte (le *corpus*), d'un passage répondant à des critères exprimés préalablement par l'utilisateur.

L'exemple ci-contre correspond à la recherche, dans les paroles de la chanson "Champagne" de J. Higelin, des mots commençant par une lettre comprise entre 'f' et 'z' et comportant un redoublement de la lettre 's'.

Ça n'a aucun intérêt ? Sans doute. Mais si vous savez faire ça, vous saurez aussi résoudre les nombreux problèmes du même genre que vous risquez de rencontrer réellement.



L'interface utilisateur du programme réalisé au cours du TD 06

1 - Réflexions préliminaires

Le travail effectué par le concordancier correspond assez directement aux fonctionnalités de la classe `QString` présentées dans la Leçon 6. L'essentiel du problème est donc d'une part de placer dans une `QString` le texte que l'utilisateur du programme souhaite explorer et, d'autre part, de présenter d'une façon agréable les résultats obtenus.

Etant donné que l'écriture de programmes traitant des données stockées dans un fichier ne nous est pas encore permise (c'est l'objet de la prochaine Leçon...), nous allons exiger de l'utilisateur qu'il ouvre le fichier concerné avec un éditeur de texte quelconque, qu'il copie l'intégralité du texte, puis qu'il colle ces données dans un widget capable de les recevoir (un "textEdit", en l'occurrence).

Traditionnellement, la présentation des résultats obtenus par un concordancier adopte une mise en page centrée sur les occurrences répondant aux exigences de la recherche, les contextes amont et aval de ces occurrences étant respectivement présentés à leur gauche et à leur droite. Ce type de présentation est facilement réalisable à l'aide d'un widget de type "table".

La visualisation du texte sur lequel porte la recherche n'est pas nécessaire en permanence. Etant donné qu'elle occupe beaucoup de place, il semble préférable de ne la proposer à l'utilisateur que comme une alternative à l'affichage des résultats. Un "tabWidget" dispose de plusieurs pages sur lesquelles on peut placer d'autre widgets, et permet donc d'offrir sans difficultés ce genre de présentation.

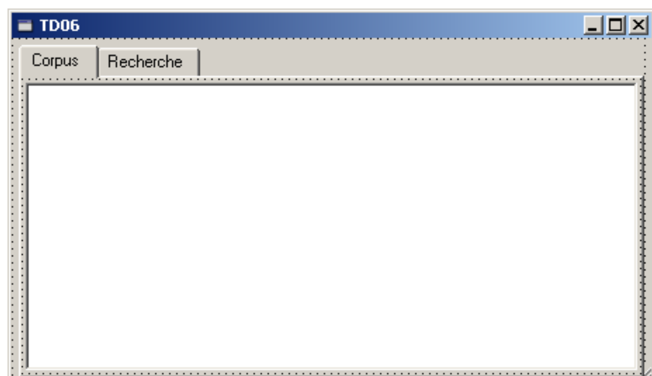
2 - Création du projet et dessin de l'interface

En vous inspirant de la procédure décrite dans le TD 3, créez un projet nommé TD06 ☐.

Eliminez tous les widgets présents dans le dialogue, et remplacez-les par un tabWidget (menu "Tools", catégorie "Containers") occupant l'essentiel de l'espace disponible ☐.

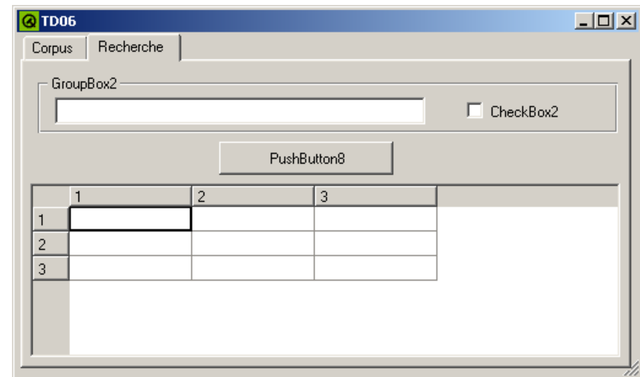
Utilisez le "Property editor" pour renommer ce widget "lesPages" ☐ et pour remplacer, dans la zone intitulée "PageTitle", l'intitulé "Tab1" par "Corpus" ☐.

Insérez dans votre projet de dialogue un widget textEdit (menu "Tools", catégorie "Input") occupant totalement la page "Corpus" du tabWidget ☐, et renommez ce textEdit pour qu'il s'appelle "corpus" ☐.



Dans le dessin du dialogue, cliquez sur l'onglet "Tab2" pour faire passer la seconde page au premier plan ☐.

Modifiez la "PageTitle" de cette page pour qu'elle devienne "Recherche" ☐.



Disposez, sur cette seconde page :

- une groupBox (menu "Tools", catégorie "Containers") ☐,
- un lineEdit (catégorie "Input") ☐,
- une checkBox (catégorie "Buttons") ☐,
- un pushButton (catégorie "Buttons") ☐
- une table (catégorie "Views") ☐.

Votre projet de dialogue devrait maintenant ressembler à celui représenté ci-contre.

Modifiez les propriétés des widgets citées dans le tableau ci-contre, de sorte qu'elles adoptent les valeurs suggérées ☐.

Widget	Propriété	Valeur
groupBox	Title	Cible
lineEdit	Name	cible
	Font	Courier New 10
checkbox	Name	expReg
	Text	Expression régulière
pushButton	Name	b_chercher
	Text	Chercher
	Default	true
table	Name	resultats

Créez un slot nommé `f_chercher` ☐ et associez-le au clic sur le bouton `b_chercher` ☐.

Enregistrez votre travail ☐, puis revenez à Visual C++ ☐.

Procédez aux modifications de rigueur sur la fonction `main()`, de façon à ce qu'elle instancie la classe `TD06DialogImpl` ☐.

3 - Ecriture du code

Le programme comporte deux sortes d'instructions : celles qui doivent être effectuées à chaque fois que l'utilisateur clique sur le bouton [Rechercher] et qui prendront naturellement place dans la fonction `f_rechercher()`, et celles qui ne doivent être exécutées qu'une seule fois, en début de programme, et prendront place dans le constructeur de la classe `TD06DialogImpl`.

La seule tâche qu'il est nécessaire d'accomplir en début de programme est la préparation du tableau de résultats : il faut créer les colonnes et leur donner un titre.

Préparation du tableau de résultats

La création des colonnes se fait en appelant une fonction membre de la classe `QTable` (la classe qui fournit les fonctionnalités du widget du même nom) nommée `setNumCols()`. Cette fonction est munie d'un paramètre de type `int`, qui permet de lui indiquer combien de colonnes doit désormais avoir l'instance au titre de laquelle elle est appelée.

Bien entendu, l'appel d'une fonction membre de `QTable` implique la présence d'une directive

```
#include "qtable.h"
```

Les titres des colonnes ne sont pas gérés directement par la `QTable`, mais par une de ses composantes, de type `QHeader`. Pour spécifier ces titres, il nous faut donc nous adresser à cet objet, dont la `QTable` nous fournit gracieusement l'adresse en réponse à l'appel de la fonction `horizontalHeader()`.

Une fois cette adresse connue, elle peut être utilisée pour appeler une fonction membre de la classe QHeader nommée `setLabel()`, qui attend deux arguments : le numéro de la colonne concernée, et le nom qu'elle doit adopter.

Modifiez ☐ le contenu du fichier TD06DialogImpl.cpp, pour qu'il devienne :

```
1  #include "td06dialogImpl.h"
2  #include "qtable.h"          //explique ce que sont les QTable et les QHeader
3  const int NB_COLONNES = 3;   //le nombre de colonnes de notre QTable
4  TD06DialogImpl::TD06DialogImpl( QWidget* parent,const char* name, bool modal, WFlags f )
5     : TD06Dialog( parent, name, modal, f )
6  {
   //Préparation de la QTable
7  resultats->setNumCols(NB_COLONNES);
8  QHeader * titres = resultats->horizontalHeader(); //on veut parler au QHeader
9  titres->setLabel(0, "Contexte gauche");//on lui indique les titres des colonnes
10 titres->setLabel(1, "Occurrence");
11 titres->setLabel(2, "Contexte droit");
12 resultats->setNumRows(0); //la table est pour l'instant vide
13 }
```

Compilez (F7) et exécutez (F5) votre programme ☐.

Vérifiez que l'onglet "Recherche" propose bien un tableau correctement configuré ☐.

La fonction `f_chercher()`

Ajoutez à la classe `TD06DialogImpl` une fonction membre nommée `f_chercher()` ☐.

Cette fonction ne doit pas rechercher *une* occurrence de la cible choisie par l'utilisateur, mais *toutes* les occurrences de cette cible figurant dans le corpus. Si elle peut s'appuyer sur les fonctions de recherche proposées par les classes `QString` et `QRegExp`, il lui reste donc à organiser une boucle garantissant que, après la découverte d'une occurrence (et son affichage dans le tableau), une nouvelle recherche sera effectuée sur la suite du corpus. Lorsqu'une de ces recherches s'avèrera infructueuse, c'est que toutes les occurrences auront été découvertes.

Nous savons que les fonctions de recherche renvoient la position à laquelle elles ont trouvé la cible, et signalent leur échec éventuel en renvoyant une position égale à –1.

L'architecture générale de la fonction `f_chercher()` sera donc :

```
1  void TD06DialogImpl::f_chercher()
2  {
3  //préparatifs
4  int position = 0;
5  do {
6     //position = cherche la cible à partir de position
7     if(position != -1)
8         {
9         //affiche l'occurrence découverte
10        position = position + 1;
11        }
12    } while (position != -1);
13 //finitions
14 }
```

Le rôle joué par `position` en fait la variable principale de cette fonction. Il est donc important de bien comprendre ses différents aspects :

- D'une part position contient la réponse de la fonction de recherche utilisée à la ligne 6. Cette réponse détermine non seulement la fin de la boucle (ligne 12), mais aussi l'opportunité des opérations d'affichage (ligne 7-9).
- D'autre part, `position` indique où commence la partie du corpus qui n'a pas encore été explorée, information qui doit être communiquée à la fonction de recherche. Pour que celle-ci ne re-détecte pas l'occurrence qu'elle vient de signaler, il faut éviter de lui passer la position qu'elle a renvoyée lors de l'appel précédent, et c'est là le rôle de la ligne 10.

Préparatifs

Avant d'entrer dans la boucle, il est préférable de vérifier que l'utilisateur a bien spécifié une cible et de rendre facilement disponibles certaines des informations nécessaires au traitement :

```
1  void TD06DialogImpl::f_chercher()
2  {
3  QString aChercher = cible->text();
4  if(aChercher.isEmpty())
5    return;
6  QString texte = corpus->text();
7  QRegExp expression = aChercher;
8  int tailleOccurrence = aChercher.length();
9  resultats->setNumRows(0); //effacement du contenu antérieur du tableau
```

Sans être rigoureusement indispensable, l'utilisation de `QString` locales contenant la cible fixée par l'utilisateur *(3)* et le corpus *(6)* allègent l'écriture des lignes de code qui utilisent ces données. En l'absence de ces variables, ces lignes de code devraient en effet répéter l'appel de la fonction `text()` qui permet d'obtenir l'information nécessaire.

Les variables `aChercher` et `expression` *(3 et 7)* contiennent le même texte, mais sont de types différents. Selon le souhait exprimé par l'utilisateur, c'est l'une ou l'autre de ces variables qui sera utilisée pour effectuer la recherche, mais il est sans doute plus clair de disposer systématiquement des deux plutôt que de rejeter leur création dans la partie de la fonction qui est contrôlée par un `if()` basé sur l'option choisie par l'utilisateur.

La même remarque s'applique à la variable `tailleOccurrence` *(8)* : lorsque la recherche n'utilise pas d'expression régulière, cette taille est fixe et ne mérite pas de figurer dans une variable. Lorsque les expressions régulières sont utilisées, en revanche, la variable est indispensable puisque son adresse doit être communiquée à la fonction `match()`, qui l'utilisera pour y stocker la longueur du fragment de texte "compatible" avec la cible qu'elle aura découverte. Stocker systématiquement la longueur dans une variable permet donc d'éviter de créer inutilement deux cas différents.

La ligne *9*, pour sa part, permet d'éliminer les résultats produits par une éventuelle recherche précédente.

Recherche

Comme nous l'avons vu, la première chose que doit faire le code compris dans la boucle est d'effectuer la recherche de la cible dans la partie du corpus non encore explorée. Lors de la première itération, cette partie inexplorée commence au premier caractère du corpus, ce qui est exprimé par l'initialisation de `position` *(10)*.

```
10 int position = 0;
11 do {
12   if (expReg->isChecked())
13     position = expression.match(texte, position, & tailleOccurrence);
14   else
15     position = texte.find(aChercher, position);
```

Le choix exprimé par l'utilisateur est consulté en appelant la fonction `isChecked()` au titre de la variable chargée d'assurer le fonctionnement de la "case à cocher" que nous avons placé dans le volet "Recherche" du dialogue. Si cette fonction renvoie `true`, le texte figurant dans le lineEdit doit être traité comme une expression régulière. Comme nous avons besoin de connaître la longueur du fragment découvert, nous ne pouvons pas utiliser la fonction `QString::find()` et c'est donc la fonction `QRegExp::match()` qui est appelée *(13)*. Dans le cas contraire, un simple appel à `QString::find()`suffit *(15)*, et le contenu de la variable `tailleOccurrence` restera inchangé.

Dans l'un et l'autre cas, l'exécution de cette séquence d'instructions s'achève donc avec :

- dans la variable `position`, une valeur de –1 si la fin du corpus à été atteinte sans qu'un fragment correspondant au critère de recherche ait été trouvé, ou une valeur égale à la position de ce fragment dans le cas contraire.
- dans la variable `tailleOccurrence`, la longueur du fragment trouvé, si un fragment a été trouvé (dans le cas contraire, le contenu de `tailleOccurrence` est sans importance, puisque aucun affichage n'aura lieu).

Une conséquence importante de cette identité des résultats produits dans les deux cas est que la suite du code ne dépendra pas du type de la recherche qui a été effectuée.

Affichage : récupération des fragments de texte concernés

Les instructions assurant l'affichage ne doivent évidemment être effectuées que si un fragment de texte répondant au critère de recherche a été trouvé *(16)*.

Copier ce fragment de texte *(18)* ne présente aucune difficulté, puisque nous en connaissons la `position` et la taille.

Copier le `contexteGauche` est un peu plus délicat. En effet, si le fragment découvert est proche du début du corpus, il est possible qu'il n'y ait pas suffisamment de caractères disponibles pour fournir un contexte de la taille choisie. La taille du `contexteGauche` n'est donc égale à `T_CONTEXTE` que si `position` est supérieure ou égale à cette constante *(21)*. Si c'est le cas, le début du `contexteGauche` se situe, par définition, `T_CONTEXTE` caractères avant le début du fragment découvert *(22)*. Dans le cas contraire, le `contexteGauche` s'étend du début du texte au début du fragment découvert, ce qui signifie que sa taille est donnée par la `position` de ce dernier et qu'il peut être recopié à l'aide de la fonction `left()` *(24)*.

```
16    if (position != -1)
17      {
        //extraction de l'occurrence
18      QString occurrence = texte.mid(position, tailleOccurrence);
        //extraction du contexte amont
19      const unsigned int T_CONTEXTE = 35; //taille du contexte
20      QString contexteGauche;
21      if(position >= T_CONTEXTE)
22        contexteGauche = texte.mid(position - T_CONTEXTE, T_CONTEXTE);
23      else
24        contexteGauche = texte.left(position);//tout le contexte disponible
        //extraction du contexte aval
25      QString contexteDroit;
26      int debutContexteDroit = position + tailleOccurrence;
27      if(debutContexteDroit + T_CONTEXTE <= texte.length())
28        contexteDroit = texte.mid(debutContexteDroit, T_CONTEXTE);
29      else
30        contexteDroit = texte.right(texte.length() - debutContexteDroit);
```

La recopie du `contexteDroit` suit une logique similaire : si l'occurrence découverte est trop proche de la fin du corpus, il faudra se contenter des caractères disponibles. La position du début du `contexteDroit` est facile à déterminer, et elle permet d'extraire facilement celui-ci lorsqu'il n'est pas tronqué par la fin du corpus *(28)*. Dans le cas contraire (*30*), il faut calculer sa taille, et la fonction `right()` permet ensuite de le recopier.

> La création d'une constante *(19)* fixant la taille des fragments de textes amont et aval présentés dans le tableau de résultats permet à la fois de modifier facilement cet aspect du programme et d'améliorer la lisibilité du code concerné.

Affichage : insertion des fragments de texte dans le tableau

Une fois extraits les fragments de textes correspondants à l'occurrence et à son contexte, la mise à jour de l'affichage ne pose plus vraiment de problème : il suffit d'utiliser les fonctions prévues pour ajouter une ligne et donner aux cellules le contenu souhaité.

On modifie le nombre de lignes d'un tableau à l'aide de la fonction `setNumRows()`, à laquelle il convient de passer comme argument le nombre de lignes désirées. Comme nous voulons simplement ajouter une line, ce nombre est supérieur d'une unité au nombre actuel de lignes, lui-même obtenu en appelant la fonction `numRows()` :

```
        //ajout d'une ligne au tableau de résultats
31      int ligne = resultats->numRows();
32      resultats->setNumRows(ligne + 1);
```

L'affectation d'un contenu à une cellule se fait par l'intermédiaire de la fonction `setText()`. Cette fonction reçoit trois arguments qui indiquent respectivement les numéros de ligne et de colonne de la cellule visée, et le contenu qu'elle doit recevoir :

```
33        //remplissage des cellules
34        resultats->setText(ligne, 0, contexteGauche.simplifyWhiteSpace());
35        resultats->setText(ligne, 1, occurrence.simplifyWhiteSpace());
36        resultats->setText(ligne, 2, contexteDroit.simplifyWhiteSpace());
37        position = position + 1;
38        }
39      } while (position != -1);
```

Les fragments de texte sont affichés sur la dernière ligne du tableau, celle qui vient d'être créée. Le numéro de cette ligne est égal au nombre de lignes qu'avait le tableau AVANT cette création, car les lignes du tableau sont, comme toujours en C++, numérotées à partir de zéro.

Plutôt que de copier directement `contexteGauche`, `occurrence` et `contexteDroit` dans les cellules du tableau, les lignes 34 à 36 insèrent la copie "nettoyée" renvoyée par `simplyfyWhiteSpace()` lorsqu'elle est invoquée au titre de chacune de ces variables. Il serait en effet peu compatible avec une présentation en tableau de chercher à afficher dans une cellule un texte comportant des passages à la ligne.

Les lignes 37 à 39 sont celles dont nous avions prévu la présence dès notre première réflexion sur l'architecture générale de la fonction.

Finitions

Pour améliorer la qualité de la présentation des résultats, une dernière opération peut être effectuée : ajuster la largeur des colonnes à leur contenu. La fonction `adjustColum()` effectue précisément cette tâche sur la colonne dont on lui passe le numéro comme argument :

```
40  int colonne;
41  for(colonne = 0 ; colonne < NB_COLONNES ; ++colonne)
42     resultats->adjustColumn(colonne);
43  } //fin de la fonction f_chercher()
```

Par ailleurs, l'utilisation de fonctions membre des classes correspondant aux différents widgets exige l'ajout, en début du fichier "TD06Dialog.cpp", des directives suivantes :

```
#include "QMultiLineEdit.h"
#include "QLineEdit.h"
#include "QCheckBox.h"
#include "QRegExp.h"
```

Après avoir donné à la fonction `f_chercher()` le contenu que nous venons de décrire, compilez (F7) ☐ puis exécutez (F5) ☐ votre programme.

Pour tester votre programme, vous pouvez taper un texte bref dans le textEdit. Si vous voulez jouer avec un corpus plus conséquent, un simple copier/coller à partir d'un éditeur de textes quelconque remplace avantageusement la saisie au clavier. Une intéressante collection de corpus potentiels peut être trouvée sur http://abu.cnam.fr

4 - Prolongements

Parmi les nombreuses améliorations possibles pour notre concordancier, nous en envisagerons ici trois. Les deux premières sont assez faciles à réaliser pour faire l'objet de simples exercices, alors que la troisième mérite une présentation détaillée, car elle fait appel à des fonctions membre de la classe `QTextEdit` que nous n'avons pas encore rencontré.

Exercices

1) Ajoutez au programme un dispositif permettant à l'utilisateur d'accumuler dans le tableau les résultats de plusieurs recherches successives.

2) Ajoutez au programme un dispositif permettant à l'utilisateur de modifier la taille des contextes présentés dans le tableau de résultats.

Montrer une occurrence particulière dans le texte intégral

Quelle que soit la taille du contexte affiché dans le tableau de résultats, il arrive que l'utilisateur en souhaite d'avantage[1]. Plutôt que de chercher à augmenter la quantité d'information présentée dans le tableau, il est donc préférable de permettre à l'utilisateur de désigner l'occurrence qui l'intéresse et de lui présenter alors celle-ci dans son contexte intégral (le corpus lui-même).

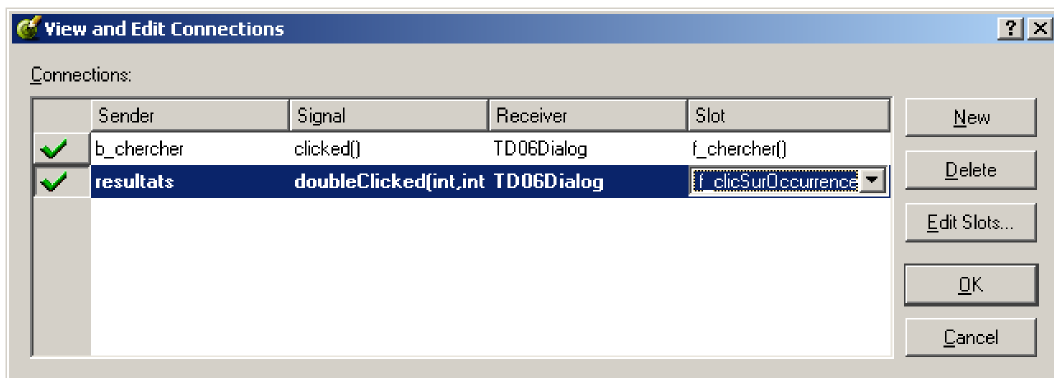
Nous allons donc faire en sorte que, lorsque l'utilisateur double-clique sur une des lignes du tableau, l'affichage bascule sur la page "Corpus" du dialogue. Nous nous assurerons, en outre, que le textEdit présent sur cette page affiche alors la zone du texte qui contient l'occurrence désignée, celle-ci étant sélectionnée (ce qui la rend visuellement très saillante).

Détection de l'évènement "double-clic" dans une ligne du tableau

Dans Qt Designer, choisissez la commande "Connections..." du menu "Edit" ☐. Dans le dialogue qui s'ouvre alors, cliquez sur le bouton "Edit slots"☐ pour ouvrir le dialogue de création de slots.

Cliquez sur le bouton "New function" ☐ et entrez le texte suivant dans la zone d'édition "Function:" `f_clicSurOccurrence(int, int, int, const QPoint &)` ☐.

Cliquez sur OK pour refermer le dialogue de création de slots ☐, et, dans le dialogue de connexions, sélectionnez le signal "`doubleClicked(int, int, int, const QPoint &)` de la QTable `resultats`, le "receiver" `TD06Dialog`, et le slot que nous venons de créer, `f_clicSurOccurrence(int, int, int, const QPoint &)` ☐ :



Cliquez enfin sur OK pour refermer le dialogue de connexion signal/slots ☐, n'oubliez pas de sauver le résultat de vos efforts ☐.

> Nous venons simplement de faire en sorte que tout double-clic sur une des lignes du tableau résultat se traduise par l'appel de la fonction `f_clicSurOccurrence()`.

Revenez à Visual C++ ☐ et ajoutez à la classe `TD06DialogImpl` une fonction nommée `f_clicSurOccurrence()` acceptant comme paramètres trois `int` et une référence à un `QPoint` constant ☐.

> Ces quatre paramètres indiquent respectivement la ligne et la colonne du tableau sur lesquelles a eu lieu le double-clic, le bouton sur lequel l'utilisateur a éventuellement cliqué[2] et les coordonnées exactes du pointeur de la souris au moment du double-clic. Seule la première de ces informations (le numéro de ligne) nous intéresse, mais il faut néanmoins que nous créions une fonction acceptant tous les arguments qui vont lui être transmis.

Analyse

La tâche incombant à la fonction `f_clicSurOccurrence()` est de retrouver, dans le corpus, l'occurrence décrite par la ligne du tableau de résultats sur laquelle l'utilisateur a cliqué. On pourrait penser qu'il suffit de récupérer le contenu de cette ligne et de procéder à sa recherche dans le contenu du `textEdit`. Plusieurs facteurs rendent cependant cette approche inutilisable :

- Le contenu des cellules du tableau n'est pas une copie littérale du corpus (nous utilisons la fonction `simplifyWhiteSpace()` pour garder au tableau un aspect agréable).

[1] La présentation en tableau ne se prête de toutes façons guère à un affichage visuellement confortable d'un extrait vraiment long.

[2] Les cellules d'une table sont en effet capables de contenir des widgets (des boutons, par exemple)

- Même si le tableau contenait une copie parfaite d'un extrait du corpus, rien ne garantirait l'unicité de cet extrait. Si le texte constitué par la réunion du contexte gauche, de l'occurrence et du contexte droit apparaît plusieurs fois dans le corpus, comment peut-on déterminer laquelle de ces occurrences correspond à celle désignée par l'utilisateur ?

Le tableau de résultats ne fournit donc pas suffisamment d'informations pour permettre à la fonction `f_clicSurOccurrence()`d'effectuer facilement sa tâche, et nous allons devoir compléter son contenu.

Le widget dans lequel nous souhaitons visualiser l'occurrence désignée est un `textEdit`. Ce type de widget gère son contenu en numérotant les paragraphes qu'il contient et en désignant chaque caractère par sa position à l'intérieur du paragraphe. La tâche de `f_clicSurOccurrence()` sera donc très simplifiée si le tableau de résultats comportent deux colonnes supplémentaires, l'une donnant le numéro du paragraphe où figure l'occurrence décrite par la ligne, et l'autre donnant la position de cette occurrence à l'intérieur de ce paragraphe.

Ajout de deux colonnes au tableau de résultats

Il suffit de modifier la valeur de la constante `NB_COLONNES` ☐ et d'ajouter deux lignes (*10* et *11*) au constructeur de la classe `TD06DialogImpl` ☐ :

```
1   const int NB_COLONNES = 5;    //le nombre de colonnes de notre QTable
2   TD06DialogImpl::TD06DialogImpl( QWidget* parent, const char* name, bool modal, WFlags f )
3      : TD06Dialog( parent, name, modal, f )
4   {
    //Préparation de la QTable
5   resultats->setNumCols(NB_COLONNES);
6   QHeader * titres = resultats->horizontalHeader(); //on veut parler au QHeader
7   titres->setLabel(0, "Contexte gauche");//on lui indique les titres des colonnes
8   titres->setLabel(1, "Occurrence");
9   titres->setLabel(2, "Contexte droit");
10  titres->setLabel(3, "Paragraphe");
11  titres->setLabel(4, "Position");
12  resultats->setNumRows(0); //la table est pour l'instant vide
13  }
```

Modification de la fonction `f_chercher()`

Lors de la création d'une ligne de résultats, ces deux colonnes doivent évidemment recevoir un contenu. La position dans le corpus de l'occurrence à décrire est donnée par la variable `position`. Il est donc facile (*1*) d'isoler la partie du corpus qui précède l'occurrence. Le numéro du paragraphe est alors simplement le nombre de passages à la ligne contenus dans cette partie du corpus (*2*), alors que la position de l'occurrence à l'intérieur de ce paragraphe est la différence entre la longueur de cette partie du corpus et la position du dernier passage à la ligne qu'elle contient (*3*).

```
        //calcul de la position de l'occurrence (paragraphe + offset)
1       QString avant = texte.left(position);
2       int numParagraphe = avant.contains("\n");
3       int posPar = position - avant.findRev("\n") - 1;
        //remplissage des cellules
4       resultats->setText(ligne, 0, contexteGauche.simplifyWhiteSpace());
5       resultats->setText(ligne, 1, occurrence.simplifyWhiteSpace());
6       resultats->setText(ligne, 2, contexteDroit.simplifyWhiteSpace());
7       resultats->setText(ligne, 3, QString::number(numParagraphe));
8       resultats->setText(ligne, 4, QString::number(posPar));
9       position = position + 1;
```

La fonction `QString::number()` est utilisée (*7* et *8*) pour créer "à la volée" les chaînes représentant, dans le système positionnel, en base dix, en utilisant les chiffres arabes, les valeurs devant être affichées dans les colonnes 3 et 4.

La fonction `f_clicSurOccurrence()`

Dans ces conditions, la fonction `f_clicSurOccurrence()` peut récupérer les informations nécessaires dans le tableau de résultat, en veillant toutefois à retransformer les chaînes représentant des valeurs numériques en véritables valeurs numériques :

```
1  void TD06DialogImpl::f_clicSurOccurrence(int ligne, int, int, const QPoint &)
2  {
3  QString duTableau = resultats->text(ligne, 3); //le numéro de paragraphe
4  int numParagraphe = duTableau.toInt();
5  duTableau = resultats->text(ligne, 4); //la position dans le paragraphe
6  int posPar = duTableau.toInt();
7  QString cible = resultats->text(ligne, 1) ;
```

La fonction `QTable::text()` est en quelque sorte l'inverse de la fonction `QTable::setText()` que nous avons utilisé pour remplir le tableau de résultat. Il faut simplement lui préciser la `ligne` et la colonne concernée, et elle renvoie le contenu de la cellule ainsi désignée. Dans le cas présent, la ligne qui nous intéresse est celle sur laquelle l'utilisateur a cliqué et dont le numéro a été passé comme premier argument à la fonction `f_clicSurOccurrence()`.

Une fois ces données obtenues, il faut sélectionner l'occurrence cible, ce qui assurera sa visibilité. Les `QTextEdit` offrent une fonction `setSelection()`, mais elle est ici peu adaptée car elle exige de préciser la position de début et la position de fin du fragment à sélectionner.

La cible peut comporter des sauts de ligne, ce qui signifie nous ignorons le numéro du paragraphe contenant la fin de la sélection, et, a fortiori, la position de cette fin à l'intérieur de ce paragraphe. Des calculs seraient donc nécessaires pour pouvoir utiliser `setSelection()`.

Il est plus simple d'utiliser un autre membre de `QTextEdit()`, la fonction `find()`. Cette fonction attend six paramètres, qui indiquent respectivement le texte à chercher, s'il faut ou non distinguer majuscules et minuscules, s'il faut ignorer les occurrences non entourées de séparateurs, s'il faut chercher du début vers la fin et, finalement, dans quelles variables doivent être placées les informations indiquant à quelle position la cible a été découverte.

Dans le cas présent, nous connaissons parfaitement la position où nous allons trouver notre cible, mais ce sont d'autres caractéristiques de la fonction `find()` qui nous intéressent : elle s'achève en ayant sélectionné l'occurrence qu'elle a découverte, et elle est capable de commencer sa recherche à une position indiquée par le contenu initial des variables pointées par ses deux derniers paramètres. L'utilisation des variables contenant les valeurs extraites du tableau de résultats garantit donc que c'est bien la bonne occurrence qui sera trouvée (puisque la recherche débutera exactement sur cette occurrence), et la fonction `find()` se chargera elle-même de la sélection de la cible.

```
8  corpus->find(cible, true, false, true, &numParagraphe, &posPar);
9  resultats->selectRow(ligne);
10 lesPages->setCurrentPage(0);
11 }//fin de la fonction f_clicSurOccurrence()
```

La ligne *9* a pour effet de sélectionner la ligne du tableau de résultats sur laquelle a eu lieu le double-clic, ce qui améliore un peu le confort d'utilisation. La ligne *10* fait passer la page "Corpus" au premier plan, de façon à ce que l'utilisateur bénéficie directement du positionnement du `textEdit` sur l'occurrence désignée.

5 - Qu'avons nous appris ?

Bien connaître la librairie peut vous épargner des semaines de travail...