



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

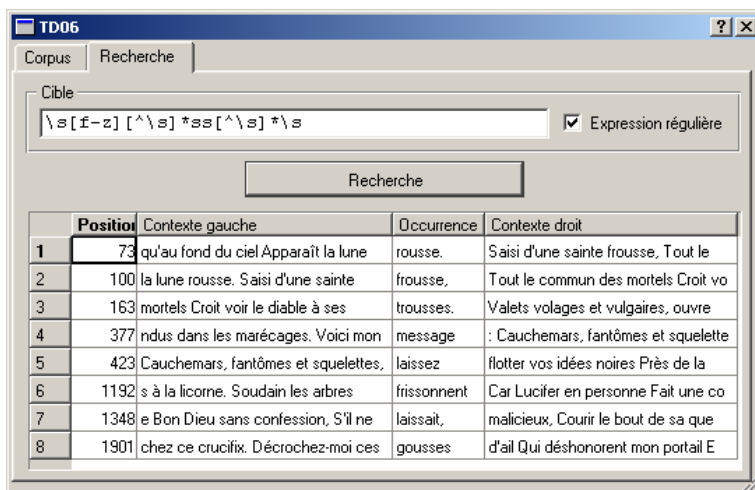
## TD 6 : Un concordancier

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 <code>f_chercher()</code> .....	4
Préparatifs .....	5
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
La fonction <code>f_clicSurOccurrence()</code> .....	8

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 "multilineEdit", 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 .


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

Gardez une petite marge autour du `tabWidget`, elle vous servira pour établir la connexion signal/slots...

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 `multilineEdit` (menu "Tools", catégorie "Input") occupant totalement la page "Corpus" du `tabWidget` .

Renommez le `multilineEdit` pour qu'il s'appelle "corpus"  et modifiez sa propriété "wordWrap" pour lui donner la valeur "WidgetWidth".

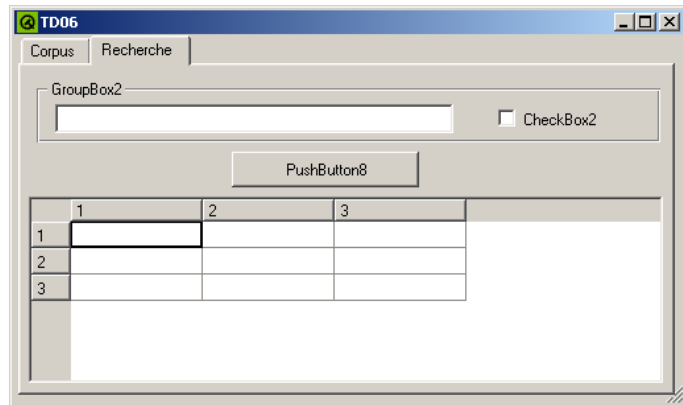
Dans le dessin du dialogue, cliquez sur l'onglet "Tab2" pour faire passer la seconde page au premier plan .

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

Dans le menu "Edit", choisissez la commande "Slots..." ☐.

Dans le dialogue d'édition des slots qui apparaît alors, cliquez sur le bouton "New slot" ☐.

Dans la zone d'édition nommée "Slot:", tapez `f_chercher()`, puis cliquez sur "OK" ☐.

Dans le menu "Tools", choisissez la commande "Connect signal/slots" ☐.

Dans votre projet de dialogue, cliquez sur le bouton "Chercher" et, tout en tenant le bouton enfoncé, faites glisser le pointeur de la souris vers une zone du dialogue non occupée par le tabWidget, puis relâchez le bouton de la souris ☐.

Sélectionnez le signal `clicked()` et le slot `f_chercher()`, puis cliquez sur OK ☐.

La fonction `f_chercher()` sera donc appelée lorsque l'utilisateur cliquera sur le bouton `b_chercher`.

Enregistrez votre travail (Menu "File", commande "Save") ☐, puis revenez à Visual C++ ☐.

### 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 `TD06Dialog`.

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 `TD06Dialog.cpp`, pour qu'il devienne :

```

1 #include "td06dialog.h"
2 #include "QTable.h" //explique ce que sont les QTable et les QHeader
3 TD06Dialog::TD06Dialog( QWidget* parent, const char* name, bool modal, WFlags f )
4 : TD06DialogBase( parent, name, modal, f )
5 { //Préparation de la QTable
6   resultats->setNumCols(4); //on indique à la QTable qu'elle aura 4 colonnes
7   QHeader * titres = resultats->horizontalHeader(); //on veut parler au QHeader...
8   titres->setLabel(0, "Position"); //on lui indique les titres des 4 colonnes
9   titres->setLabel(1, "Contexte gauche");
10  titres->setLabel(2, "Occurrence");
11  titres->setLabel(3, "Contexte droit");
12 }
```

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

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

#### La fonction `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 TD06Dialog::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 rendre facilement disponibles certaines des informations que celle-ci devra utiliser :

```
1 void TD06Dialog::f_chercher()  
2 {  
3     const unsigned int T_CONTEXTE = 35;  
4     QString texte = corpus->text();  
5     QString aChercher = cible->text();  
6     QRegExp expression = aChercher;  
7     int longueur = aChercher.length();  
8     resultats->setNumRows(0); //effacement du contenu antérieur du tableau
```

La création d'une constante (3) 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.

Sans être rigoureusement indispensable, l'utilisation de QString locales contenant le corpus (4) et la cible fixée par l'utilisateur (5) 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 (5 et 6) 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 longueur (7) : lorsque la recherche n'utilise pas d'expression régulière, la taille du texte 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 découvert. Stocker systématiquement la longueur dans une variable permet donc d'éviter de créer inutilement deux cas différents au moment de l'affichage de l'occurrence découverte.

La ligne 8, 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 (9).

```
9 int position = 0;  
10 do {  
11     if (expReg->isChecked())  
12         position = expression.match(texte, position, & longueur);  
13     else  
14         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 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 (12). Dans le cas contraire, un simple appel à QString::find() suffit (14), et le contenu de la variable longueur restera inchangé.

Dans les deux 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 a 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 longueur, la longueur du fragment trouvé, si un fragment a été trouvé (dans le cas contraire, le contenu de longueur 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 recherche 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 (15).

Copier ce fragment de texte (17) ne présente aucune difficulté, puisque nous en connaissons la position et la longueur.

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 (19). Si c'est le cas, le `début du contexteGauche` se situe, par définition, `T_CONTEXTE` caractères avant le début du fragment découvert (20). 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()` (22).

```

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

```

La copie 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 (26). Dans le cas contraire, il faut calculer sa `taille`, et la fonction `right()` permet ensuite de le recopier.

#### 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 ligne, ce nombre est supérieur d'une unité au nombre actuel de ligne, lui-même obtenu en appelant la fonction `numRows()` :

```

29     //ajout d'une ligne au tableau de résultats
30     int ligne = resultats->numRows();
31     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 :

```
31 //remplissage les cellules
32 resultats->setText(ligne, 0, QString::number(position));
33 resultats->setText(ligne, 1, contexteGauche.simplifyWhiteSpace());
34 resultats->setText(ligne, 2, occurrence.simplifyWhiteSpace());
35 resultats->setText(ligne, 3, contexteDroit.simplifyWhiteSpace());
36 position = position + 1;
37 }
} 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 ligne qu'avait le tableau AVANT cette création, car les lignes du tableau sont, comme toujours en C++, numérotées à partir de zéro.

La première colonne affiche la position de l'occurrence dans le corpus. La fonction `QString::number()` est utilisée pour créer "à la volée" une chaîne représentant cette position dans le système positionnel, en base dix, en utilisant les chiffres arabes (31).

Plutôt que de copier directement `contexteGauche`, `occurrence` et `contexteDroit` dans les cellules du tableau, les lignes 32 à 34 insèrent la copie "nettoyée" renvoyée par `simplifyWhiteSpace()` 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 5 à 37 sont celles dont nous avons 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 `adjustColumn()` effectue précisément cette tâche sur la colonne dont on lui passe le numéro comme argument :

```
38 int colonne;
39 for(colonne = 0 ; colonne < 4 ; colonne = colonne + 1)
40     resultats->adjustColumn(colonne);
41 } //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"
```

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 `multiLineEdit`. 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 `QMultiLineEdit` que nous n'avons pas encore rencontrées.

### Exercices

- 1) Ajoutez au programme un dispositif permettant à l'utilisateur de modifier la taille des contextes présentés dans le tableau de résultats.
- 2) Ajoutez au programme un dispositif permettant à l'utilisateur d'accumuler les résultats de plusieurs recherches successives.



### 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.


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

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 cette occurrence 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 multiLineEdit présent sur cette page affiche alors la zone du texte qui contient l'occurrence désignée, la ligne concernée 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 "Connect signal/slots" du menu "Tools" .




Cliquez sur le tableau de résultats et, tout en tenant le bouton de la souris enfoncé, faites glisser le pointeur vers une zone du dialogue libre de tout widget .

Attention, une zone du tabWidget non occupée par le tableau ne convient pas... Il faut vraiment aller dans une zone du dialogue qui soit en dehors du tabWidget.

Relâchez le bouton de la souris  et, 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 slot"  et entrez le texte suivant dans la zone d'édition "slot:" `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 &)" 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  (Menu "File", commande "Save") et revenez à Visual C++ .



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()`.

### La fonction `f_clicSurOccurrence()`

Cette fonction reçoit quatre paramètres qui 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<sup>1</sup> 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.

Dans l'onglet "Class View" de la fenêtre "Workspace", cliquez avec le bouton droit sur le nom de la classe `TD06Dialog`. Dans le menu qui s'ouvre alors, choisissez "Add member function..." .

Dans le dialogue de création d'une fonction membre, indiquez que la fonction est de type `void`  et que sa déclaration est `f_clicSurOccurrence(int ligne, int, int, const QPoint &)` .

Attention, le **premier paramètre** ne doit pas rester ici anonyme, car les instructions que nous allons placer dans le corps de la fonction auront besoin d'accéder à sa valeur.

Cliquez ensuite sur OK pour refermer le dialogue de création d'une fonction membre .

Le problème qui se pose à la fonction `f_clicSurOccurrence()` peut être décomposé en trois sous-problèmes. Il lui faut en effet :

- déterminer quelle partie du corpus doit être montrée à l'utilisateur ;
- retrouver cette partie dans le multiLineEdit qui contient le corpus ;
- assurer la visibilité de cette partie du multiLineEdit.

<sup>1</sup> Les cellules d'une table sont en effet capables de contenir des widgets (des boutons, par exemple)



La ligne du tableau de résultats sur laquelle l'utilisateur a cliqué contient, dans sa première colonne, un texte qui représente la position de l'occurrence dans le corpus. Pour connaître cette position, il suffit donc d'extraire le contenu de la **Première Colonne** et de **retrouver l'équivalent numérique** de cette chaîne de caractères :

```
1 void TD06Dialog::f_clicSurOccurrence(int ligne, int, int, const QPoint& )
2 {
3 //où doit-on aller ?
4 QString contenuDeLaPremiereColonne = resultats->text(ligne, 0);
5 int positionSouhaitee = contenuDeLaPremiereColonne.toInt();
```

La fonction `QTable::text()` est en quelque sorte l'inverse de la fonction `QTable::setText()` que nous avons utilisée 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()`.

Connaître la `positionSouhaitee` ne permet pas directement d'assurer la visibilité du texte correspondant. En effet, la manipulation du contenu d'un `multilineEdit` se fait en termes de numéros de lignes, et non en termes de numéros de caractères par rapport au début du texte. Il nous faut donc déterminer quelle ligne du widget contient le texte recherché.

Nous allons donc rechercher la première ligne du widget dont la fin corresponde, dans le corpus, à une position plus éloignée du début que ne l'est la `positionSouhaitee`.

Cette ligne est en effet celle qui contient le caractère qui est à la `positionSouhaitee` et dont nous voulons assurer la visibilité.

Cette recherche est effectuée en déterminant, la **position dans le corpus** du dernier caractère de **chaque ligne du widget**. Dès que cette position **excède la positionSouhaitee**, nous avons trouvé la ligne recherchée.

```
//où est-ce ?
5 QString texte = corpus->text();
6 int limite = corpus->numLines();
7 int finLigne = 0;
8 for(ligne=0 ; ligne < limite && finLigne < positionSouhaitee ; ligne = ligne+1)
9 {
10     QString laLigne = corpus->textLine(ligne);
11     int debutLigne = texte.find(laLigne);
12     finLigne = debutLigne + laLigne.length();
13 }
```

Remarquez que, du fait des règles de fonctionnement de la boucle `for( ; ; )`, l'exécution du fragment de code présenté ci-dessus laisse dans la variable `ligne` une valeur supérieure d'une unité au numéro de la ligne recherchée (**l'augmentation** de la valeur de `ligne` *précède* l'évaluation de **l'expression qui met fin** à la boucle).

Une fois connu le numéro de la ligne du `multilineEdit` qui contient l'occurrence recherchée, il ne reste plus qu'à en assurer la visibilité :

```
//allons-y
14 lesPages->setCurrentPage(0);
15 corpus->setCursorPosition(ligne - 1, 0);
16 corpus->setCursorPosition(ligne, 0, true);
17 } //fin de la fonction f_clicSurOccurrence()
```

La fonction `QTabWidget::setCurrentPage()` permet au programme d'obtenir ce que l'utilisateur obtient en cliquant sur l'un des onglets : la page désignée passe au premier plan.

Le premier appel (15) à la fonction `QMultiLineEdit::setCursorPosition()` place le curseur au **début** de la ligne à mettre en évidence. Le second le fait passer au **début** de la **ligne suivante**, mais en indiquant que ce déplacement doit s'accompagner de la **sélection** du texte parcouru (comme si l'utilisateur faisait passer le curseur d'une position à l'autre en maintenant enfoncé le bouton de la souris).