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

TD 3 : Une calculette

Le thème du TD 3 est la mise en œuvre des notions présentées dans la Leçon 3 à propos de la création de fonctions et des moyens de les faire communiquer entre elles.

Comme le suggère l'image ci-contre, le prétexte de cette mise en œuvre est la réalisation d'un programme simulant le fonctionnement d'une calculette très simple, qui n'accepte d'opérer que sur des nombres entiers et n'effectue que les quatre opérations arithmétiques de base.

En dépit de cette simplicité, le programme que vous allez créer nécessite une petite réflexion préalable, et son écriture nous conduira à introduire quelques éléments du langage C++ que vous n'avez pas encore rencontrés.

L'interface utilisateur du programme du TD 03

1 - Préliminaires

Avant de se lancer dans la création d'un projet et la définition de fonctions, il est toujours salutaire de prendre quelques instants pour mettre noir sur blanc une description fonctionnelle du programme que l'on cherche à obtenir. C'est à partir de cette description que l'on peut ensuite prendre quelques décisions fondamentales concernant l'organisation générale du programme.

Cahier des charges

La calculette que nous cherchons à simuler doit pouvoir être utilisée de la façon suivante :
a) on tape un premier nombre ;
b) on choisit une des opérations disponibles ;
c) on tape un second nombre ;
d) on appuie sur la touche "=" pour obtenir le résultat.
La calculette est alors prête à recevoir le premier nombre d'une nouvelle opération.

Cette calculette ne permet de saisir que des entiers positifs, mais elle peut avoir à afficher un résultat décimal (après certaines divisions) ou négatif (après certaines soustractions).

Comment pourrait-on programmer ça ?

Il existe de nombreuses façons d'imaginer une interface utilisateur répondant à ce cahier des charges, mais le fait que nous utilisions Windows, ainsi que l'image présentée ci-dessus, suggèrent que les saisies soient réalisées en cliquant sur des boutons correspondants d'une part aux chiffres et, d'autre part, aux opérations proposées. L'affichage de la saisie en cours et du résultat obtenu sera effectué sur un petit "écran", réduit à une seule ligne.

Analyse chronologique

Si on reprend la chronologie suggérée par le cahier des charges, en tenant compte du type d'interface que nous venons de choisir, nous sommes déjà conduits à un certain nombre de constatations qui déterminent largement les variables et les fonctions que nous allons créer.

a) <u>on tape un premier nombre</u>

- Les chiffres sont tapés un à un. Chacune des "touches chiffre" devant être prise en compte au moment où l'utilisateur clique dessus, il faut "construire" progressivement la valeur du nombre.
- On tape normalement les nombres en commençant par le chiffre de gauche. A chaque saisie d'un nouveau chiffre, nous devons donc multiplier la valeur actuelle de la saisie par 10 (pour décaler tous les chiffres vers la gauche et "faire de la place" à celui qui vient d'être tapé) et lui ajouter la valeur correspondant à cette nouvelle saisie.

> Cette étape du raisonnement est souvent celle qui pose le plus de problèmes aux étudiants. Prenez quelques instants pour vous convaincre que "rajouter un chiffre à droite" d'un nombre entier, c'est le multiplier par dix et lui additionner la valeur correspondant au chiffre en question

b) <u>on choisit une des opérations disponibles</u>

- La saisie du premier nombre est donc terminée. Il faut "mettre de côté" la valeur courante de la saisie et repartir à zéro pour une nouvelle saisie.
- Le bouton qui vient d'être pressé indique quelle est l'opération souhaitée par l'utilisateur. Cette opération ne peut toutefois pas être effectuée immédiatement, puisque le second nombre n'est pas encore connu. Il nous faut donc également mettre cette information de côté.

c) on tape un second nombre

- La prise en compte des chiffres est effectuée comme dans le cas de la première saisie.

d) on appuie sur la touche "=" pour obtenir le résultat

- Il faut donc effectuer l'opération. Les deux nombres sont disponibles (l'un a été "mis de côté", l'autre est la valeur qui vient d'être saisie), et nous disposons d'une variable indiquant l'opération requise. Ce qui nous manque, en revanche, c'est un moyen nous permettant d'effectuer un traitement dont la nature dépend de la valeur de cette variable. Nous devrons donc utiliser un dispositif qui ne sera officiellement présenté qu'au cours de la Leçon 4.
- Comme le cahier des charges spécifie que "La calculette est alors prête à recevoir le premier nombre d'une nouvelle opération", il faut aussi prendre soin de la remettre dans un état permettant d'effectuer un nouveau calcul.

Variables et fonctions

Cette analyse fait apparaître la nécessité de deux variables. L'une doit stocker le premier nombre saisi et est donc de type numérique. La seconde doit stocker l'opération choisie par l'utilisateur, et doit donc ne pouvoir prendre que l'une des quelques valeurs possibles. La définition d'un type énuméré "sur mesure" va nous permettre de créer une telle variable. Il semble également clair que la valeur en cours de saisie devra être stockée dans une variable de type numérique.

Le programme devra disposer d'au moins trois variables : deux `double` (`premierNombre` et `saisieEnCours`) et une variable d'un type énuméré (`operationChoisie`).

En plus des principales variables, notre analyse doit nous conduire à identifier l'organisation générale du programme et son découpage en fonctions. Une première hypothèse serait que, puisque notre analyse fait apparaître quatre étapes dans l'utilisation du programme, celui-ci va se composer d'une fonction pour chacune de ces quatre étapes. Ce découpage mériterait sans doute d'être envisagé *si nous n'étions pas en train d'écrire une application Windows*. Il correspond en effet à une vision des choses selon laquelle c'est le programme qui décide de ce qui doit se passer, et qui sollicite en conséquence l'utilisateur. Les applications Windows ne fonctionnent pas du tout comme cela puisque, fondamentalement, elles se contentent de réagir aux actions de l'utilisateur. Concrètement, cela veut dire que nous n'aurons pas, par exemple, une fonction qui demandera un premier nombre à l'utilisateur, mais une fonction qui sera appelée lorsque l'utilisateur clique sur le bouton "1", une autre fonction appelée lorsque l'utilisateur clique sur le bouton "2", etc.

> La tâche incombant au programme est de réagir aux événements lorsqu'ils se produisent et non de les organiser en une séquence immuable et choisie par le programmeur.

Un premier découpage en fonctions est donc imposé par la gestion des événements : chaque bouton sera associé à une fonction dont il déclenchera l'exécution. Ce découpage présente l'inconvénient de créer un grand nombre de fonctions quasiment identiques. Les dix boutons correspondant aux chiffres, par exemple, ont le même travail à effectuer : multiplier la saisie par 10, lui ajouter une certaine valeur et afficher le nombre obtenu. La seule nuance entre ces dix fonctions est la valeur qui doit être ajoutée à la saisie, et notre programme sera plus facile à écrire, à lire et à modifier si les instructions communes à ces dix fonctions ne sont pas dupliquées, mais placées dans une onzième fonction, chargée d'ajouter un chiffre à la saisie et appelée par chacune des dix autres.

Il est également nécessaire, au début du programme et après chaque calcul, de placer la calculette dans l'état où elle est "prête à effectuer un nouveau calcul". Cette tâche devant être effectuée dans deux contextes différents, la confier à une fonction nommée `nouveauCalcul()` permet d'éviter de dupliquer le code correspondant.

> Un second découpage en fonctions est donc suggéré par notre désir de "mettre en facteur" les séquences d'instructions identiques, afin de pouvoir intervenir de façon centralisée et efficace si des modifications (corrections ou perfectionnements) s'avéraient nécessaires.

Le programme devra comporter au moins 17 fonctions : une pour chacun des quinze boutons, plus `ajouteChiffre()` et `nouveauCalcul()`.

Armés de cette première vision générale du programme que nous allons réaliser, nous pouvons (enfin) raisonnablement lancer l'exécution de Visual C++  .

Création du projet

Créez un "projet Qt" (en cliquant sur le bouton) et nommez-le "TD03".

Si vous rencontrez des difficultés, reportez vous à la description de la démarche de création d'un projet Qt proposée dans le premier TD.

2 - Création de l'interface

La première chose à faire est d'éliminer tous les widgets placés automatiquement par Qt Designer, de façon à obtenir un dialogue entièrement vierge .

Mise en place des widgets

Dans le menu "Tools", catégorie "Display", sélectionnez l'outil "LCDNumber" .

Placez un LCDNumber dans la partie supérieure du dialogue , et utilisez la fenêtre "Property Editor" pour lui donner propriétés indiquées ci-contre .

Name	ecran
numDigits	8
segmentStyle	flat

Les autres caractéristiques de ce widget peuvent rester telles que proposées par QT Designer.

Dans le menu "Tools", catégorie "Buttons", sélectionnez l'outil "Push button" .

Placez un bouton dans la partie gauche du dialogue, en dessous du LCDNumber .

Ce bouton va nous servir de modèle, et sa duplication va nous permettre d'obtenir rapidement la quinzaine de boutons quasi identiques dont nous avons besoin.

Le bouton étant sélectionné, utilisez la fenêtre "Property Editor" pour lui donner les caractéristiques suggérées ci-contre .

Name	b_x
Size policy / hSizeType	Preferred
Size policy / vSizeType	Preferred
Text	@

Le nom et le texte utilisés rappellent que le bouton n'est qu'un modèle qui devra être modifié, et la "politique de taille" choisie va nous permettre d'obtenir facilement un clavier de calculette d'apparence satisfaisante.

Le bouton étant toujours sélectionné, copiez-le (menu Edit, commande Copy) .

Collez-le (menu Edit, commande Paste) et placez la copie à côté de l'original .

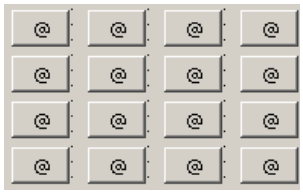
Attention, la copie créée se superpose à l'original, donnant ainsi l'impression que l'opération a échoué. Déplacez la copie, vous verrez l'original qui est dessous.

Répétez encore deux fois l'opération précédente, pour obtenir une ligne de quatre boutons alignés, comme dans l'image ci-contre .

Sélectionnez les quatre boutons[1] et copiez-les .

Collez, puis faites glisser la nouvelle ligne de boutons en dessous de la première .

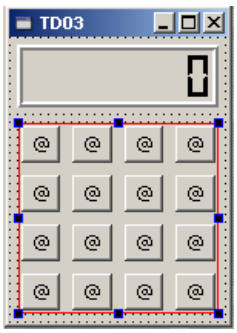
Répétez deux fois de plus l'opération précédente, de façon à obtenir un ensemble de seize boutons alignés, comme dans l'image ci-contre .



Supprimez l'un des deux boutons centraux de la ligne du bas, et augmentez la taille de l'autre de façon à occuper l'espace libéré .

Sélectionnez les quinze boutons et, dans le menu "Layout", choisissez la commande "Lay out in a grid" .

[1] Rappel : pour réaliser une sélection multiple, cliquer sur un premier élément, enfoncez la touche Shift et cliquez successivement sur les autres éléments avant de relâcher la touche Shift. Lorsque les éléments à sélectionner sont contigus, il est également possible d'utiliser le rectangle de sélection en cliquant à côté du premier d'entre eux et en faisant glisser la souris pour agrandir le rectangle jusqu'à ce qu'il les englobe tous.



Cette commande a pour effet de regrouper tous les boutons sélectionnés dans une unité que Qt Designer appelle un "layout" et représente sous la forme d'un encadré rouge.

En déplaçant et en redimensionnant cet encadré (et, éventuellement, le LCDNumber et le dialogue lui-même), essayez de donner à votre calculette une apparence semblable à celle représentée ci-contre.

Sélectionnez successivement chacun des boutons et utilisez la fenêtre "Property Editor" pour leur donner des noms (`b_0` à `b_9` pour les touches "chiffres", `b_plus`, `b_moins`, `b_fois`, `b_divise` et `b_egal` pour les autres) et des textes conformes à leur fonction.

Votre dialogue devrait maintenant ressembler beaucoup à celui représenté page 2.

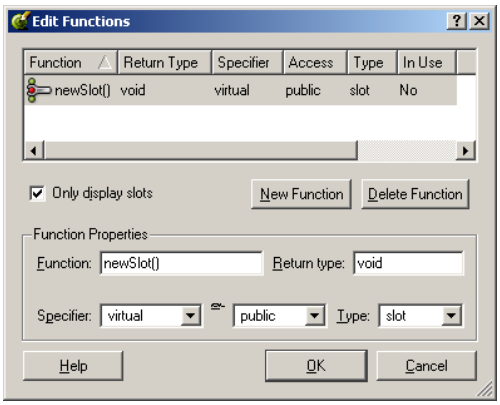
Création des slots

Comme nous l'avons vu dans le TD 1, le déclenchement d'un traitement par un widget exige l'établissement d'un lien entre un événement concernant ce widget et un slot (déplacement du curseur du slider et modification de la valeur affichée par le LCDNumber, dans le TD 1).

Dans le cas présent, les widgets concernés sont les 15 boutons, et chacun d'entre eux doit déclencher un traitement lorsqu'il est cliqué.

A la différence du TD 1, toutefois, les traitements que nous souhaitons voir effectuer ne correspondent pas à des slots prédéfinis (tels que l'affichage d'une valeur par un LCDNumber) mais seront effectués par des fonctions que nous allons écrire. Il nous faut donc annoncer l'existence de ces fonctions, de façon à pouvoir préciser ensuite laquelle doit correspondre à chacun des boutons.

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



Dans la fenêtre qui s'ouvre alors (cf. ci-contre), cliquez sur le bouton "New function" pour obtenir la création d'un `newSlot()`.

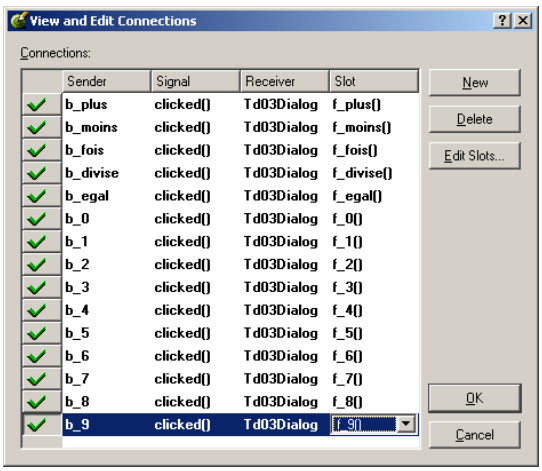
Utilisez la zone d'édition "Function" du groupe "Function Properties" pour rebaptiser ce slot "`f_0()`".

Cliquez ensuite sur le bouton "New function" pour obtenir la création d'un nouveau slot, que vous renommerez "`f_1()`".

Procédez de même pour créer successivement les slots `f_2()`, `f_3()`, `f_4()`, `f_5()`, `f_6()`, `f_7()`, `f_8()`, `f_9()`, `f_plus()`, `f_moins()`, `f_fois()`, `f_divise()` et `f_egal()`.

Association des slots aux boutons

Sélectionnez la commande "Connections..." du menu "Edit".



- Pour créer une nouvelle connexion, cliquez sur le bouton "New".
- Déroulez ensuite la liste "Sender" et choisissez le widget `b_0`.
- Dans la liste "Signal", choisissez "`clicked()`".
- Dans la liste "Receiver", choisissez "TD03Dialog".
- Dans la liste "Slot", choisissez la fonction `f_0()`.

Répétez ces opérations jusqu'à ce que votre liste de connexions soit conforme à celle présentée ci-contre.

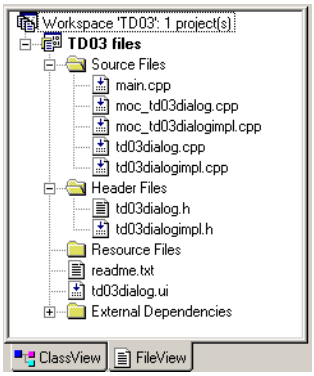
Une fois que toutes les connexions nécessaires sont établies, n'oubliez pas d'enregistrer votre travail (menu "File", commande "Save all") avant de revenir à Visual C++.

3 - Ecriture du code

Si vous compilez (F7) et exécutez (F5) le programme dans son état actuel, vous pouvez constater qu'il se présente bien comme nous le souhaitons, mais que la calculette n'est pas utilisable. Le code que nous allons ajouter pour faire fonctionner la calculette va prendre place au sein du code généré par Qt Designer, et il est préférable de commencer par examiner celui-ci pour comprendre comment il est organisé. Pour cela, refermez la calculette et revenez à Visual C++.

L'œuvre de Qt Designer

La première chose à comprendre est que, pour réaliser l'interface que nous lui avons demandée, Qt Designer a créé une classe. Si vous avez bien nommé votre projet TD03, cette classe a été baptisée `TD03Dialog`.



Comme nous l'avons vu au cours du TD 2, la définition d'une classe est habituellement placée dans un fichier portant le nom de la classe en question et l'extension .h L'examen des fichiers créés par Qt Designer (onglet "File View" de la fenêtre "Workspace", cf ci-contre) montre que Qt Designer respecte cette tradition.

> Si la fenêtre "Workspace" n'est pas visible, rouvrez-la à l'aide de la commande correspondante du menu "View".

Si vous les ouvrez (en double-cliquant sur leur nom), vous constaterez que les fichiers TD03Dialog.h et TD03Dialog.cpp débutent par un commentaire comportant cet avertissement :

```
** WARNING! All changes made in this file will be lost!
```

Ces fichiers sont en effet sous la responsabilité de Qt Designer, qui sera conduit à les re-créer pour tenir compte des modifications effectuées sur le dessin de l'interface.

> Cet avertissement signifie donc qu'il est inutile de se fatiguer à taper du code dans ces fichiers (pour définir les fonctions associées aux boutons, par exemple) car le moindre déplacement d'un widget à l'aide de Qt Designer se traduirait par l'écrasement du code péniblement mis au point.

Dans ces conditions, où allons-nous pouvoir placer le code que nous allons rédiger ?

Qt Designer a prévu pour cela une seconde classe, nommée `TD03DialogImpl`. Si vous ouvrez le fichier TD03DialogImpl.h où se trouve définie cette classe, vous pouvez y reconnaître l'architecture de la définition d'une classe :

```
1  #include "td03dialog.h"
2  class TD03DialogImpl : public TD03Dialog
3  {
4     Q_OBJECT
5  public:
6     TD03DialogImpl(QWidget* parent=0, const char* name=0, bool modal=FALSE, WFlags f=0);
7  };
```

Outre ces éléments familiers, la définition de la classe `TD03DialogImpl` comporte, sur la ligne *2*, une information capitale : cette classe *dérive* de la classe `TD03Dialog`. Nous aurons l'occasion, dans une prochaine Leçon, d'étudier en détail ce processus de dérivation et le mécanisme d'*héritage* qu'il implique. Il vous suffit, pour l'instant, de retenir que, du fait de cette relation de dérivation, la classe `TD03DialogImpl` comporte toutes les caractéristiques de la classe `TD03Dialog` (décrites par les fichiers TD03Dialog.h et TD03Dialog.cpp), plus des caractéristiques qui lui sont propres (décrites par les fichiers TD03DialogImpl.h et TD03DialogImpl.cpp). C'est cette dualité qui va nous permettre de faire cohabiter sans conflit le code généré automatiquement par Qt Designer et le code que nous allons rédiger "à la main" :

Nous n'utiliserons jamais Visual C++ pour modifier la classe TD03Dialog.

Toutes les variables ou fonctions que nous souhaiterons ajouter à notre dialogue seront, au contraire, ajoutées à la classe `TD03DialogImpl`.

> La définition de la classe `TD03DialogImpl` comporte également (ligne *4*) une instruction rendue nécessaire par le fait que TD03Dialog dérive elle-même d'une classe définie dans la librairie Qt et (ligne *6*) la déclaration d'un *constructeur*, fonction membre qui présente la particularité d'être exécutée automatiquement lorsqu'une instance de la classe est créée.

La création de classes (c'est à dire de types) ne donne pas naissance à un programme. S'il se passe quelque chose lorsque vous pressez F5, c'est parce que Qt Designer a généré (dans le fichier "main.cpp") du code ayant pour effet d'instancier la classe `TD03Dialog` et d'appeler, au titre de cette instance, une fonction de "mise en marche" du dialogue que cette classe décrit :

```
1  #include <qapplication.h>
2  #include "td03dialog.h"
3  int main( int argc, char** argv )
4  {
5     QApplication app( argc, argv );
6     TD03Dialog dialog( 0, 0, TRUE );
7     app.setMainWidget(&dialog);
8     dialog.exec();
9     return 0;
10 }
```

> Il s'agit là de la fonction `main()`, dont nous avons signalé qu'elle était automatiquement exécutée lors du lancement du programme.

L'examen de ce code révèle un problème : la classe instanciée (ligne *6*) est la classe `TD03Dialog`, c'est à dire celle qui ne comporte que les caractéristiques spécifiées par Qt Designer, à l'exclusion des variables et fonctions que nous allons être conduits à créer.

Modification du type du dialogue créé

Modifiez les lignes *2* et *6* du fichier main.cpp, de façon à ce que la fonction `main()` instancie la classe `TD03DialogImpl` ☐.

> Si la fonction `main()` générée automatiquement n'instancie pas la classe `TD03DialogImpl`, c'est parce que cette façon de procéder n'est que l'une de celles rendues possibles par Qt Designer.

Vérifiez que le programme compile (F7) sans erreur et que son exécution (F5) donne le même résultat que précédemment ☐.

> Pour l'instant la classe `TD03DialogImpl` ne comporte aucune caractéristique propre, mais seulement celles héritées de la classe `TD03Dialog`. Instancier l'une ou l'autre ne peut donc donner lieu à aucune différence notable lors de l'exécution du programme.

Création du type énuméré `operation`

Nous savons que nous allons devoir utiliser une variable d'un type énuméré pour "mettre de côté" l'information concernant la nature de l'opération à effectuer. Comme nous l'avons vu au cours du TD 2, il est préférable de faire figurer la définition d'un type dans un fichier spécifique qui peut ensuite être "injecté" là où il est nécessaire grâce à la directive `#include`.

Dans le menu "File", choisissez la commande "New" ☐. Dans l'onglet "Files" du dialogue qui s'ouvre alors, sélectionnez la catégorie "C/C++ Header File" et donnez comme nom du fichier à créer "operation.h", puis cliquez sur le bouton "OK" ☐.

Donnez à ce fichier le contenu suivant ☐ :

```
enum operation {ERREUR, ADDITION, SOUSTRACTION, MULTIPLICATION, DIVISION};
```

> Il est souvent judicieux de prévoir une valeur spéciale permettant aux variables d'indiquer qu'elles ne contiennent aucune valeur significative. Cette valeur (ERREUR, dans le cas présent) donne aux variables du type concerné une possibilité qui ressemble à celle dont toutes les variables sont fondamentalement dépourvues : la possibilité d'être "vide".

Déclaration des fonctions et variables

Quinze des 17 fonctions dont notre analyse a révélé la nécessité vont voir leur exécution déclenchée par un clic sur un bouton. Elles ont donc été signalées à Qt Designer comme étant des slots, et ceci implique que ces fonctions soient membres de la classe `TD03DialogImpl`.

Les trois variables que nous avons prévues doivent être accessibles à plusieurs de ces fonctions slots. Il faut donc qu'elles soient elles-mêmes membre de la classe `TD03DialogImpl`.

> La valeur de `premierNombre` doit pouvoir être fixée par n'importe laquelle des fonctions "opération" (`f_plus()`, `f_moins()`, `f_fois()` et `f_divise()`), et elle sera utilisée par `f_egal()`.

La variable `saisieEnCours`, pour sa part, sera utilisée tantôt par l'une des fonctions "opération" (pour fixer la valeur de `premierNombre`), tantôt par `f_egal()` (pour effectuer le calcul). Enfin, `operationChoisie` doit évidemment recevoir sa valeur de l'une des fonctions "opération", et cette valeur doit pouvoir être utilisée par `f_egal()`.

La fonction `ajouteChiffre()` doit être en mesure de modifier le contenu de `saisieEnCours`, et la fonction `nouveauCalcul()` devra fixer des valeurs aux trois variables. Ces deux fonctions doivent donc, elles aussi, être membres de la classe `TD03DialogImpl`.

Modifiez le fichier "TD03dialogImpl.h" pour qu'il comporte les déclarations nécessaires ☐ :

```
1  #include "TD03dialog.h"
2  #include "operation.h"
3  class TD03DialogImpl : public TD03Dialog
4  {
5     Q_OBJECT
6  public:
7     TD03DialogImpl( QWidget* parent=0, const char* name=0, bool modal=FALSE, WFlags f=0 );
   //variables
8     double premierNombre;
9     double saisieEnCours;
10    operation operationChoisie;
   //slots
11    void f_0();
12    void f_1();
13    void f_2();
14    void f_3();
15    void f_4();
16    void f_5();
17    void f_6();
18    void f_7();
19    void f_8();
20    void f_9();
21    void f_plus();
22    void f_moins();
23    void f_fois();
24    void f_divise();
25    void f_egal();
   // autres fonctions
26    void ajouteChiffre();
27    void nouveauCalcul();
28 };
```

Définition des fonctions

Comme nous le savons, le code définissant le corps de ces fonctions va prendre place dans le fichier "TD03DialogImpl.cpp". Ce fichier contient déjà la définition d'une fonction (le constructeur évoqué page 6). C'est donc après cette définition que nous allons ajouter les nôtres.

Les fonctions "opération"

Le seul aspect de la tâche dévolue à une fonction "opération" que nous n'ayons pas encore évoqué est l'affichage de la valeur courante de la saisie. Cet affichage a lieu dans le `LCDNumber`, et Qt Designer a créé pour nous une variable portant le même nom que ce widget (`ecran`, si vous avez respecté scrupuleusement les instructions données précédemment) qui permet de s'adresser à lui pour lui dire ce qu'il doit afficher :

```
ecran->display(saisieEnCours);
```

Cette expression met en œuvre un passage de paramètre à une fonction, une technique que nous n'étudierons que dans la Leçon 5. Pour qui connaît le sens du verbe anglais "to display" (montrer, afficher), la signification de la ligne ci-dessus devrait cependant être assez évidente pour que cette anticipation ne soit pas trop gênante.

Les `LCDNumber` n'étant de toute évidence pas des objets faisant partie du langage C++, une telle familiarité avec l'un d'entre eux exige toutefois qu'ils aient été préalablement présentés au compilateur au moyen d'une directive `#include "QlcdNumber.h"`. La même remarque s'applique à la classe `QMessageBox`, que nous allons rencontrer dans un instant.

L'ajout de la définition de la fonction `f_plus()` devrait donc vous conduire à un fichier "TD03dialogImpl.cpp" ayant le contenu présenté ci-dessous ☐.

Le code bleu est celui créé par Qt Designer, le reste correspond à ce que vous devez rajouter.

```
1  #include "TD03dialogImpl.h"
2  #include "qlcdnumber.h"
3  #include "qmessagebox.h"
4  TD03DialogImpl::TD03DialogImpl( QWidget* parent, const char* name, bool modal, WFlags f )
5     : TD03Dialog( parent, name, modal, f )
6  {
7  }
   //Cette fonction est exécutée lorsque l'utilisateur clique sur la touche [+]
8  void TD03DialogImpl::f_plus()
9  {
10 operationChoisie = ADDITION;       //mémorisation de l'opération choisie
11 premierNombre = saisieEnCours;     //stockage de la première valeur saisie
   //préparation de la seconde saisie
12 saisieEnCours = 0;
13 ecran->display(saisieEnCours);
14 }
```

Les autres fonctions "opération" ne diffèrent guère de celle-ci, et un peu de réflexion devrait vous permettre de définir vous-mêmes `f_moins()` ☐, `f_fois()` ☐ et `f_divise()` ☐.

La fonction `f_egal()`

Cette fonction est au cœur du programme, puisque c'est elle qui effectue le traitement qui justifie l'existence de celui-ci. Le `premierNombre` et la `saisieEnCours` étant disponibles, la seule difficulté est de les utiliser dans une opération qui dépend de la valeur contenue dans `operationChoisie`. Là encore, nous sommes conduits à anticiper légèrement sur la suite du cours pour utiliser une structure de contrôle qui n'est présentée que dans la Leçon 4. Une fois de plus, le sens du code ci-dessous ne devrait cependant pas vous paraître trop obscur pendant que vous le recopiez dans votre fichier "TD03dialogImpl.cpp" ☐.

```
   //Cette fonction est exécutée lorsque l'utilisateur clique sur la touche [=]
1  void TD03DialogImpl::f_egal()
2  {
3  double resultat;
4  switch(operationChoisie)
5     {
6     case ADDITION :
7        resultat = premierNombre + saisieEnCours ;
8        break ;
9     case SOUSTRACTION :
10       resultat = premierNombre – saisieEnCours ;
11       break ;
12    case MULTIPLICATION :
13       resultat = premierNombre * saisieEnCours ;
14       break ;
15    case DIVISION :
16       resultat = premierNombre / saisieEnCours ;
17       break ;
18    default: //ça ne devrait jamais arriver…
19       QMessageBox::critical(0, "Calculette", "Opération non spécifiée !");
20    }
21 ecran->display(resultat);
22 nouveauCalcul();
23 }
```

Remarquez que la variable `resultat` n'est utile à aucune des autres fonctions. Il n'est donc pas nécessaire d'en faire une variable membre, une variable locale à `f_egal()` suffit. La ligne *19* contient une instruction qui ne devrait jamais être exécutée et qui sert, justement, à signaler que le programme n'est pas au point. Remarquez, enfin, l'appel à `nouveauCalcul()` (ligne *22*) qui garantit que, comme l'exige le cahier des charges, la calculette est prête à reprendre le travail.

La fonction `nouveauCalcul()`

Les opérations nécessaires pour préparer la calculette à une nouvelle opération sont simples :

```
//Cette fonction est exécutée au début du programme et lorsqu'on vient
//d'achever un calcul
1 void TD03DialogImpl::nouveauCalcul()
2 {
3 saisieEnCours = 0;            //mise à zéro pour préparer une saisie
4 premierNombre = -1;           //signale l'absence de contenu significatif
5 operationChoisie = ERREUR;    //signale l'absence de contenu significatif
6 }
```

> La ligne 4 affecte à `premierNombre` une valeur qu'il ne peut normalement avoir (la calculette ne permet pas la saisie de nombres négatifs). Cette valeur n'est jamais envisagée dans la version du programme que nous sommes en train d'écrire, mais donner systématiquement, lorsque c'est possible, des valeurs repérables aux variables dont l'état est dépourvu de signification est une excellente habitude à prendre, car cela facilite grandement la détection et la correction des erreurs de programmation.

Après avoir défini la fonction `nouveauCalcul()` ☐, il reste à tenir l'une des promesses faites dans son commentaire introductif : cette fonction doit être appelée lorsque le programme commence.

Il serait possible de placer un appel à `nouveauCalcul()` dans la fonction `main()` rencontrée page 7 (et dont nous savons qu'elle constitue le "point de départ" du programme), mais une meilleure option s'offre à nous : lors de l'instanciation de la classe `TD03DialogImpl`, son constructeur est automatiquement exécuté. Pour tenir notre promesse, il nous suffit donc de compléter le code généré par Qt Designer ☐ :

```
1 #include "TD03dialogImpl.h"
2 #include "QlcdNumber.h"
3 #include "QMessageBox.h"
4 TD03DialogImpl::TD03DialogImpl(QWidget* parent, const char* name, bool modal, WFlags f )
5    : TD03Dialog( parent, name, modal, f )
6 {
7 nouveauCalcul(); //prépare la calculette pour la première opération
8 }
```

La fonction `ajouteChiffre()` et les fonctions "chiffre"

Toutes les fonctions "chiffre" ont, nous l'avons vu, le même travail à faire. Nous avons décidé de reporter les instructions décrivant ce travail dans la fonction `ajouteChiffre()`, que nous pouvons définir ainsi ☐ :

```
//Cette fonction est appelée par les fonctions "chiffre"
1 void TD03DialogImpl::ajouteChiffre()
2 {
3 saisieEnCours = saisieEnCours * 10; //on décale le nombre vers la gauche
4 saisieEnCours = saisieEnCours + leNouveauChiffre; //on ajoute le nouveau chiffre
5 ecran->display(saisieEnCours); //on affiche le nouvel état de la saisie
6 }
```

La fonction `f_0()`, par exemple, peut alors être définie comme ceci ☐ :

```
1 //Cette fonction est exécutée lorsque l'utilisateur clique sur la touche [0]
2 void TD03DialogImpl::f_0()
3 {
4 leNouveauChiffre = 0;
5 ajouteChiffre();
6 }
```

Il devrait donc vous être facile de définir les dix fonctions "chiffre" ☐.

Toutefois, pour que votre programme fonctionne, il vous faudra encore trouver où et comment la variable `leNouveauChiffre` (dont nous n'avions jamais parlé) doit être introduite ... ☐

4 - Questions

1 - Dans ce programme, quelle est la ligne de code dont l'exécution se traduit par la création d'une variable de type `operation` ?

2 - Pourquoi le fichier TD03DialogImpl.h doit-il comporter la ligne `#include "operation.h"` ?

3 - Quelle erreur pourriez vous introduire dans le programme pour qu'il arrive que le message d'alerte prévu dans la fonction `f_egal()` apparaisse ?

4 - Si nous avions choisi d'appeler `nouveauCalcul()` depuis la fonction `main()`, comment l'appel à cette fonction se serait-il présenté ?

Attention, `main()`, n'est pas une fonction membre de `TD03DialogImpl`...

5 - Comment pourrait-on modifier le programme pour que le code commun aux fonctions "opération" soit reporté dans une fonction nommée `passageALaSecondeSaisie()`, ce qui éviterait qu'il figure quatre fois dans le programme ?

5 - Qu'avons nous appris ?

Nous venons, pour la première fois, de réaliser un programme qui comporte à la fois :

- une interface graphique créée avec Qt Designer (le TD 2 en était dépourvu) ;

- des fonctions que nous avons écrites en C++ et dont l'exécution est déclenchée à l'aide de l'interface graphique (le TD 1 en était dépourvu).

La mise en place d'une organisation de ce genre sera le point de départ de tous les projets que vous allez réaliser au cours des mois qui viennent. La marche à suivre est récapitulée dans la fiche aide-mémoire disponible ici.