Centre Informatique pour les Lettres et les Sciences Humaines

TD 4 : Boucles et tests

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

2 - Un peu de magie noire 2

3 - Boucles simples 3

La fonction `f_1a10()` 3
La fonction `f_10a1()` 4
La fonction `f_2en2()` 4
La fonction `f_alphabet()` 4

4 - Boucles comportant un test 5

La fonction `f_zigzag()` 5
La fonction `f_y()` 5

5 - Boucles enchâssées 6

La fonction `f_crescendo()` 6
La fonction `f_escalier()` 6

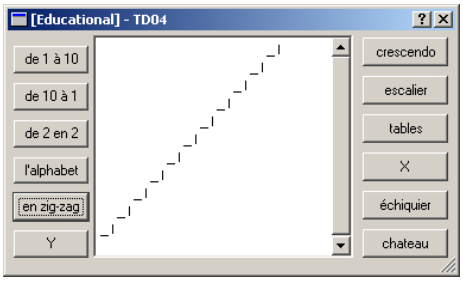
6 - Utiliser un appel de fonction pour éclaircir un enchâssement 6

La fonction `f_table()` 7
La fonction `f_x()` 8

7 - A vous de jouer... 8

Le programme que nous allons réaliser n'effectue aucune tâche réellement intéressante, puisqu'il se contente d'afficher des séries de nombres ou de lettres respectant des contraintes totalement arbitraires, qui ne sont qu'un prétexte à la mise en œuvre des différentes structures de contrôle du flux d'exécution introduites dans la Leçon 4.

Ce Td vous offre donc une occasion de "faire vos gammes" avant de vous lancer dans des œuvres plus ambitieuses.



L'interface utilisateur du programme réalisé au cours du TD 04

1 - Création du projet et dessin de l'interface

En vous inspirant de la procédure décrite lors du TD 3, créez un projet nommé TD04 dont le dialogue ressemble à l'image proposée ci-dessus. (Le widget occupant le centre du dialogue est un "TextEdit", inséré grâce à une commande de la catégorie "Input" du menu "Tools").

Donnez aux boutons les noms `b_1a10`, `b_10a1`, `b_2en2`, `b_alphabet`, `b_zigzag`, `b_y`, `b_crescendo`, `b_escalier`, `b_table`, `b_x`, `b_echiquier` et `b_chateau`.

Donnez au textEdit le nom `leTextEdit` et spécifiez (propriété "Font") qu'il doit utiliser la police "Courier " .

Créez 12 slots nommés `f_1a10()`, `f_10a1()`, `f_2en2()`, `f_alphabet()`, `f_zigzag()`, `f_y()`, `f_crescendo()`, `f_escalier()`, `f_table()`, `f_x()`, `f_echiquier()` et `f_chateau()`.

Associez chaque bouton au slot correspondant.

Enregistrez votre travail, revenez à Visual C++ et procédez aux modifications de rigueur sur la fonction `main()`, de façon à ce qu'elle instancie la classe `TD04DialogImpl`.

2 - Un peu de magie noire

Sans être spécialement difficile, l'usage normal d'un textEdit fait appel à des notions que nous n'avons pas encore abordées (passage de paramètres à des fonctions et utilisation de la classes `QString`, notamment). Plutôt que de répéter ces opérations (actuellement incompréhensibles pour vous) dans chacune des fonctions que nous allons écrire, je vous propose d'insérer dans votre fichier TD04DialogImpl.cpp les quelques lignes suivantes, dont la présence vous permettra d'afficher très facilement du texte et des valeurs numériques :

```
1  #include "td04dialogimpl.h"
2  //Opérateurs d'insertion dans un QTextEdit
3  template <typename T> QTextEdit & operator << (QTextEdit & e, T n)
4  {e.setText(e.text() + QString::number(n)); return e;}
5  template <> QTextEdit & operator << (QTextEdit & e, char *t)
6  {e.setText(e.text() + t); return e;}
7  template <> QTextEdit & operator << (QTextEdit & e, char c)
8  {e.setText(e.text() + c); return e;}
9  TD04DialogImpl::TD04DialogImpl(QWidget* parent, const char* name,bool modal, WFlags f)
10   : TD04Dialog( parent, name, modal, f ), afficheur(*leTextEdit)
11 { // Add your code
12 }
```

Le code écrit en bleu est celui que Qt Designer a écrit pour vous, le reste correspond à ce que vous devez insérer (un copier/coller à partir de ce fichier texte peut vous éviter quelques fautes de frappe...). N'oubliez pas de compléter la ligne *10*.

Pour être efficaces, ces incantations nécessitent aussi l'ajout, dans le fichier TD04DialogImpl.h, de la déclaration d'un membre de type référence :

```
QTextEdit & afficheur;
```

Cette déclaration exige à son tour la présence d'une directive #include "`QTextEdit.h`" en tête du fichier TD04DialogImpl.h

3 - Boucles simples

Comme nous l'avons vu dans le TD 3, l'ajout d'une fonction membre à la classe TD04Dialog implique la déclaration de la fonction (dans le fichier TD04Dialog.h) et sa définition (dans le fichier TD04Dialog.cpp). Visual C++ offre un moyen d'accélérer ces opérations fastidieuses.

Dans l'onglet "ClassView" de la fenêtre "Workspace", cliquez avec le bouton droit sur le nom de la classe "TD04DialogImpl" ☐.

C'est bien dans **`TD04DialogImpl`** qu'il faut ajouter la fonction, et non dans `TD04Dialog`.

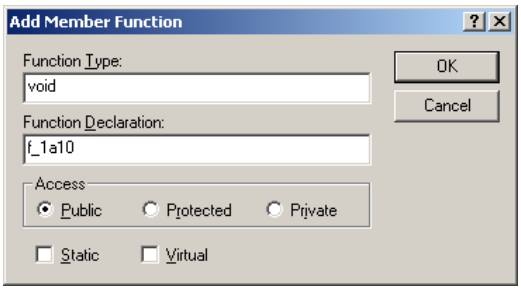
> Il arrive parfois que l'onglet "ClassView" ne propose pas la classe à laquelle vous souhaitez ajouter un membre. Pour que cet onglet soit remis à jour, il suffit d'ajouter (ou enlever) une ligne vide dans la définition de la classe absente. (Vous pouvez accéder au fichier .h contenant cette définition en double-cliquant sur son nom, dans l'onglet "FileView".)

Dans le menu qui apparaît alors, choisissez la commande "Add member function..." ☐.

Cette commande fait apparaître le dialogue d'ajout d'une fonction membre (cf. ci-contre).

Complétez les deux zones d'édition destinées à indiquer le type de la fonction ☐ et son nom ☐, puis cliquez sur le bouton "OK" ☐.

> Cette façon de procéder permet d'obtenir simultanément la déclaration de la fonction (dans le fichier .h) et la création d'une définition rudimentaire (dans le fichier .cpp).



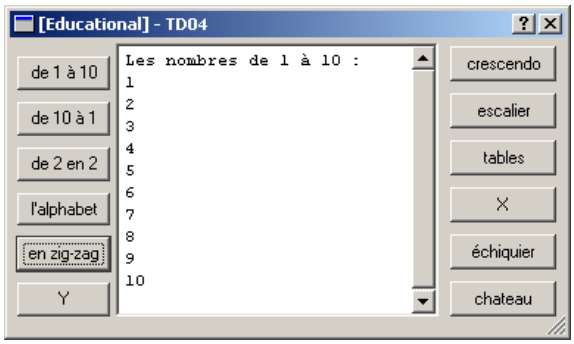
Le dialogue "Ajout d'une fonction membre"

La fonction créée doit porter le même nom que le slot connecté au bouton qui doit l'appeler.

La fonction `f_1a10()`

La fermeture du dialogue d'ajout d'une fonction membre vous conduit directement dans le fichier TD04Dialog.cpp, à l'endroit où Visual C++ a défini la nouvelle fonction. Il ne reste plus qu'à ajouter les instructions nécessaires dans le corps de celle-ci, de façon à ce que son exécution produise le résultat suggéré ci-contre.

La fonction `f_1a10()` pouvant servir de modèle pour toutes les suivantes, elle mérite que nous la discutions ligne par ligne.



Exécution de la fonction `f_1a10()`

```
//Cette fonction est appelée lorsque l'utilisateur clique sur [De 1 à 10]
1  void TD04DialogImpl::f_1a10()
2  {
3  afficheur.clear();
4  afficheur << "Les nombres de 1 à 10 : \n";
5  int nombre;
6  for(nombre = 1 ; nombre < 11 ; nombre = nombre + 1)
7     afficheur << nombre << "\n";
8  }
```

La première opération effectuée par la fonction (ligne *3*) a pour effet d'effacer (en anglais : to clear) un éventuel contenu antérieur du widget que nous utilisons pour l'affichage.

> Il peut sembler curieux que la première opération cherche à effacer quelque chose puisque, manifestement, nous n'avons encore rien écrit dans le widget. Regardez bien, toutefois, le commentaire qui précède la définition de la fonction. Rien ne prouve que ce bouton sera le premier sur lequel l'utilisateur va cliquer, ni même qu'il sera cliqué une seule fois...
>
> D'un point de vue syntaxique, la fonction `clear()` est appelée au titre d'une instance de la classe `QTextEdit` (la classe de la librairie Qt qui fait fonctionner ce type de widget).

La ligne suivante *(4)* illustre trois points simples mais importants :
- on obtient l'affichage de quelque chose en "l'envoyant" dans l'afficheur à l'aide de l'opérateur d'insertion, noté <<
- le texte qui doit être affiché "tel quel" est placé entre guillemets
- par dérogation à la règle précédente, les caractères précédés d'une barre oblique inverse prennent une signification spéciale. La séquence "\n" permet ainsi d'obtenir un passage à la ligne ("new ligne" en anglais) et "\\" désigne une unique barre oblique.

La ligne *5* définit une variable entière, qui fait ensuite *(6-7)* l'objet d'une boucle au cours de laquelle son contenu passe de 1 à 11 (cette dernière valeur provoquant la fin de la boucle).

> Remarquez, sur la ligne *7*, l'utilisation de deux opérateurs d'insertion pour envoyer successivement dans l'afficheur la valeur courante de `nombre` et un passage à la ligne.

Recopiez ces instructions dans le corps de la fonction `f_1a10()` ☐, compilez le programme ☐ et vérifiez qu'il fonctionne comme prévu ☐.

> Si rien ne se passe lorsque vous cliquez sur le bouton "De 1 à 10", retournez dans Qt Designer, vérifiez vos connexions signal/slots (Menu "Edit", commande "Connections"), et n'oubliez pas de sauver votre travail avant de revenir à Visual C++ pour recompiler.

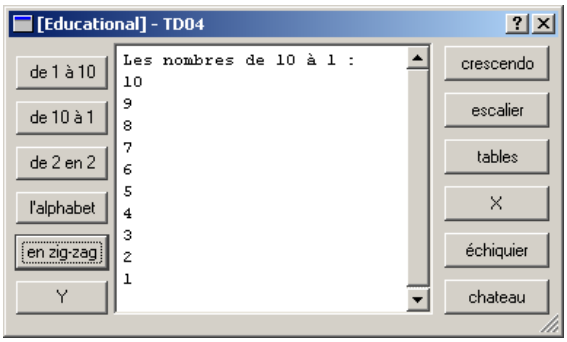
Que se passe-t-il si, sur la ligne *7*, vous placez le nom de la variable `nombre` entre guillemets ?

La fonction `f_10a1()`

Créez, en suivant la procédure décrite plus haut, une fonction membre nommée `f_10a1()` ☐.

Placez, dans le corps de cette fonction, les instructions qui permettront à son exécution de provoquer l'affichage représenté ci-contre ☐.

> N'essayez pas d'afficher en commençant par le bas, c'est impossible. Vous ne pouvez écrire qu'une ligne après l'autre, en commençant par le haut.



Exécution de la fonction `f_1a10()`

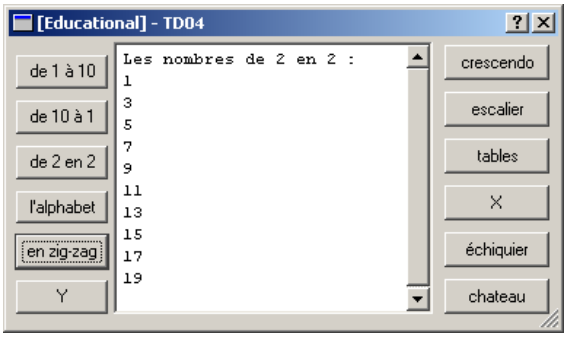
La fonction `f_2en2()`

Créez, en suivant la procédure décrite plus haut, une fonction membre nommée `f_2en2()` ☐.

Placez, dans le corps de cette fonction, les instructions qui permettront à son exécution de provoquer l'affichage représenté ci-contre ☐.

Modifiez votre fonction pour qu'elle utilise une boucle `while()` ☐.

> Si votre fonction utilise déjà une boucle `while()`, modifiez-la pour qu'elle utilise une boucle `do { } while()`.



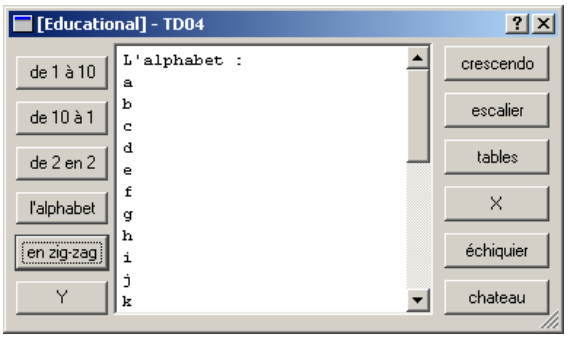
Exécution de la fonction `f_2en2()`

La fonction `f_alphabet()`

Créez, en suivant la procédure décrite plus haut, une fonction membre nommée `f_alphabet()` ☐.

Placez, dans le corps de cette fonction, les instructions qui permettront à son exécution de provoquer l'affichage représenté partiellement ci-contre ☐.

> L'affichage sera réalisé au moyen d'une boucle portant sur une variable de type `char` (nommée `lettre`), dont le contenu passera de `'a'` à `'z' + 1`, valeur qui provoquera la fin de la boucle.



Exécution de la fonction `f_alphabet()`

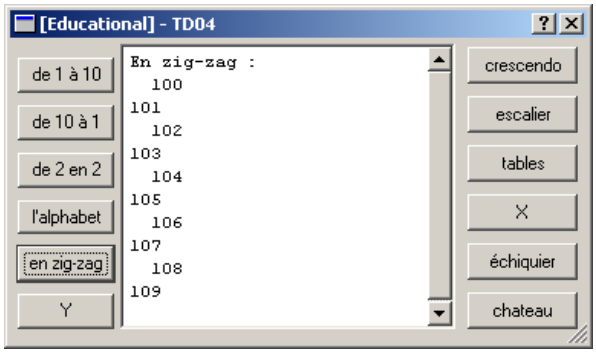
4 - Boucles comportant un test

Dans un programme réel, les structures de contrôle du flux d'exécution sont très souvent utilisées en conjonction les unes avec les autres. Les deux fonctions qui suivent exigent qu'un test soit effectué lors de chaque passage dans une boucle.

La fonction `f_zigzag()`

Créez, en suivant la procédure décrite plus haut, une fonction membre nommée `f_zigzag()` ☐.

Comme le montre l'image ci-contre, cette fonction ne traite pas tous les nombres de la même façon : les nombres pairs sont précédés de deux espaces, et leur alternance avec les nombres impairs provoque donc l'effet recherché.



Exécution de la fonction `f_zigzag()`

Comment peut-on déterminer si un nombre est pair ?

Nous avons vu (Leçon 3) que, lorsqu'une division implique des nombres de types entiers, elle donne un résultat entier. Lorsqu'un nombre impair est divisé par deux, la valeur obtenue n'est donc pas égale à la moitié du nombre. Il suffit, par conséquent, de multiplier ce résultat par deux : si on retrouve la valeur d'origine, c'est qu'elle est paire, sinon, c'est qu'elle est impaire. Nous écrirons donc ☐ :

```
//Cette fonction est appelée lorsque l'utilisateur clique sur [Zig-zag]
1  void TD04DialogImpl::f_zigzag()
2  {
3  afficheur.clear();
4  afficheur << "En zig-zag : \n";
5  int nombre;
6  for(nombre = 100 ; nombre < 110 ; nombre = nombre + 1)
7     {
8     if (nombre / 2 * 2 == nombre) //si le nombre est pair…
9        afficheur << "  ";              //on affiche deux espaces
10    afficheur << nombre << "\n";  //on affiche le nombre et on passe à la ligne
11    }
12 }
```

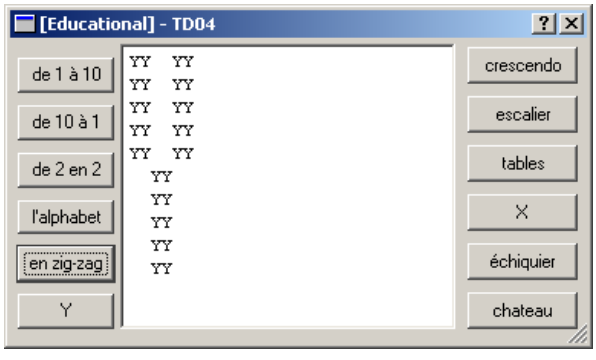
L'inclusion d'une structure de contrôle (un `if()`, dans cet exemple) à l'intérieur d'une autre (un `for( ; ; )`, dans le cas présent) ne change rien au fonctionnement de chacune de ces deux structures : le `if()` va simplement devoir faire son travail autant de fois que le `for( ; ; )` l'exigera.

La fonction `f_y()`

Cette fonction se distingue des précédentes par le fait qu'elle n'affiche pas des nombres, mais trente Y, dans une disposition qui fait apparaître cette lettre une 31ème fois.

Même si aucun `nombre` n'apparaît dans l'afficheur, il reste préférable d'organiser une boucle affichant les 10 lignes les unes après les autres : les cinq premières comportent deux blocs de deux Y, séparés par deux espaces, les cinq autres comportent deux espaces suivis de deux Y.

En d'autres termes, il faut, selon la ligne en cours d'affichage, insérer dans l'afficheur soit "YY YY", soit " YY".



Exécution de la fonction `f_y()`

Définissez la fonction `f_y()` de façon à ce que son exécution produise l'affichage représenté ci-dessus ☐.

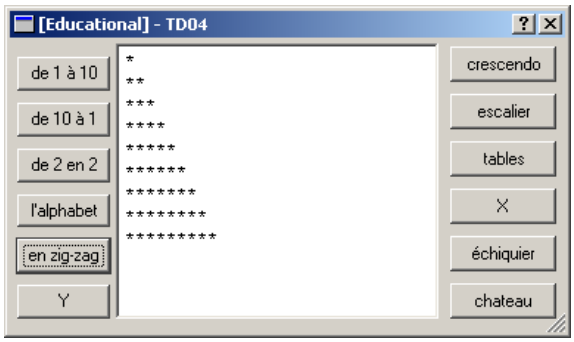
5 - Boucles enchâssées

La présence d'un test à l'intérieur d'une boucle est une situation banale en programmation. Toute aussi banale est la présence d'une seconde boucle à l'intérieur d'une première...

La fonction `f_crescendo()`

Créez, en suivant la procédure décrite plus haut, une fonction membre nommée `f_crescendo()` ☐.

Comme le montre l'image ci-contre, cette fonction affiche des lignes d'une longueur croissante.



Exécution de la fonction `f_crescendo()`

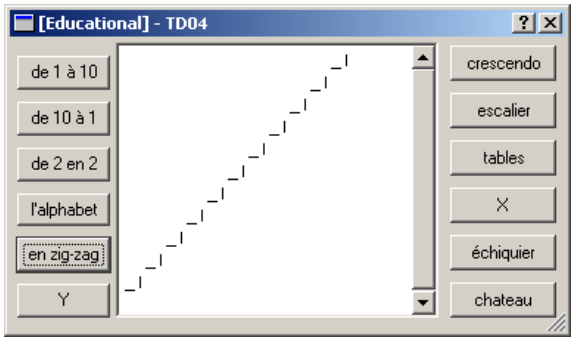
Chaque ligne étant créée par une boucle, il est clair que cette boucle doit prendre place à l'intérieur d'une autre (qui assure la création de plusieurs lignes).

Etant donné que la longueur d'une ligne est précisément le numéro de la ligne en question (un caractère sur la première ligne, deux sur la deuxième, etc), la boucle affichant les caractères doit adopter pour limite la valeur actuelle de la variable contrôlant la boucle créant les lignes :

```
//Cette fonction est appelée lorsque l'utilisateur clique sur [Crescendo]
void TD04DialogImpl::f_crescendo()
{
afficheur.clear();
int numeroLigne;
for(numeroLigne = 1 ; numeroLigne < 10 ; numeroLigne = numeroLigne + 1)
   {
   int nbCar;
   for(nbCar = 0 ; nbCar < numeroLigne; nbCar = nbCar + 1)
      afficheur << "*";
   afficheur << "\n";
   }
}
```

La fonction `f_escalier()`

L'effet que doit produire l'exécution de cette fonction est représenté ci-contre.



Exécution de la fonction `f_escaliers()`

Les marches sont obtenues par affichage de couples "_|", repoussés à la bonne distance de la marge de gauche par l'insertion préalable d'un certain nombre d'espaces.

> N'oubliez pas que vous devez nécessairement commencer l'affichage par la ligne du haut.

Le code de cette fonction n'est donc, en définitive, pas très différent de celui présent dans la fonction `f_crescendo()`.

Définissez la fonction `f_escaliers()` de façon à ce que son exécution produise l'affichage recherché ☐.

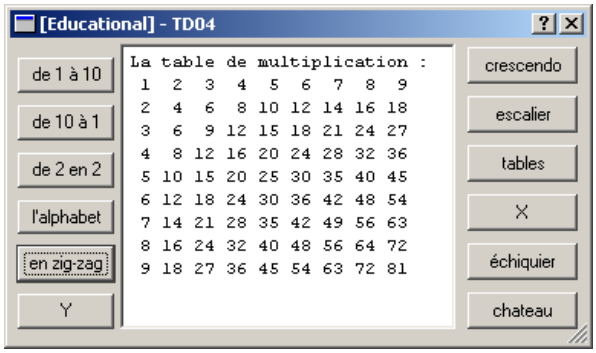
6 - Utiliser un appel de fonction pour éclaircir un enchâssement

Lorsque les choses se compliquent un peu trop, il est souvent salutaire d'isoler une partie du problème en lui donnant un nom judicieusement choisi. Il devient alors possible de manipuler cette partie du problème en faisant abstraction de sa complexité interne, ce qui permet, pour un effort identique, d'aller plus loin dans le raisonnement.

La fonction `f_table()`

L'objectif de cette fonction est, comme le montre l'image ci-contre, d'afficher les tables de multiplications.

Si l'on disposait d'une fonction capable d'afficher une ligne de la table, la tâche serait facile : il suffirait d'appeler 10 fois cette fonction, en lui indiquant à chaque fois quelle ligne elle doit créer (cette indication est nécessaire, puisque toutes les lignes de la table ne sont pas identiques).



Exécution de la fonction `f_table()`

Comment peut-on indiquer quelque chose à une fonction appelée ? Tant que nous n'avons pas étudié la Leçon 5, le seul moyen dont nous disposons est le recours à une variable membre de la classe dont sont membres les deux fonctions concernées.

La fonction `f_table()` est membre de la classe `TD04Dialog`. C'est donc dans cette classe que nous allons ajouter une fonction et une variable permettant de communiquer avec la nouvelle fonction. Cette nouvelle fonction doit créer une ligne de la table. Nous l'appellerons donc `ligneTable()`. La variable membre permet de préciser quelle ligne doit être créée. Nous l'appellerons donc `numeroLigne`. Dans ces conditions, la fonction `f_table()` devient :

```
1  void TD04DialogImpl::f_table()
2  {
3  afficheur.clear();
4  afficheur<< "La table de multiplication :\n";
5  for(numeroLigne = 1 ; numeroLigne < 10 ; numeroLigne = numeroLigne + 1)
6     ligneTable(); //appel de la fonction "auxiliaire"
7  }
```

ce qui, vous en conviendrez, est d'une simplicité rassurante.

> La fonction `f_table()` appelle `ligneTable()` sans indiquer au titre de quelle instance l'exécution de celle-ci doit avoir lieu. Comme nous l'avons vu dans la Leçon 3, la fonction appelée est, dans ce cas, exécutée au titre de l'instance pour laquelle la fonction appelante est elle-même exécutée. Ceci garantit bien que `f_table()` et `ligneTable()` accèdent à la même variable membre, une sous-variable de l'instance en question. Reste un problème : de quelle instance s'agit-il ? Il n'existe en fait qu'une seule instance de la classe `TD04Dialog`, celle qui est créée dans `main()` et au titre de laquelle la fonction de "mise en route" du dialogue est appelée (cf. TD 3). Toutes les fonctions s'appellent les unes les autres à partir de là, et elles sont donc toutes exécutées au titre de cette instance.

La fonction `ligneTable()` doit, pour sa part, afficher dix nombres séparés par des espaces. Par rapport aux "boucles simples" avec lesquelles nous avons commencé ce TD, `ligneTable()` ne doit prendre en compte que deux contraintes supplémentaires : le nombre `aAfficher` est le produit du numéro de ligne et du numéro de colonne et, lorsqu'il ne comporte qu'un seul chiffre, il faut le faire précéder d'un espace supplémentaire pour garantir l'alignement avec les nombres à deux chiffres :

```
1  void TD04DialogImpl::ligneTable()
2  {
3  int colonne;
4  for(colonne = 1 ; colonne < 10 ; colonne = colonne + 1)
5     {
6     int aAfficher = numeroLigne * colonne;
7     if(aAfficher < 10)
8        afficheur<< ' ';
9     afficheur<< aAfficher << ' ';
10    }
11 afficheur<< "\n";
12 }
```

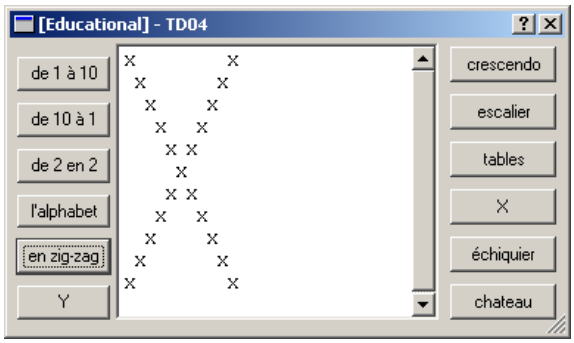
Créez les fonctions et la variable nécessaires à l'affichage des tables de multiplications ☐.

La fonction f_x()

L'affichage d'une grande lettre X composée de 21 petites fait, lui aussi, appel à une boucle enchâssée dans une autre.

Puisque nous devons afficher ligne par ligne, il n'est pas possible d'afficher les lignes obliques l'une après l'autre. Le dessin d'une ligne devra donc insérer deux X dans l'afficheur.

Si on analyse l'image ligne par ligne, on constate que le lien entre le numéro d'une ligne et les positions horizontales sur lesquelles elle comporte des X est assez simple.



Exécution de la fonction f_x()

numeroLigne	Position horizontale des X
1	1 et 11
2	2 et 10
3	3 et 9
4	4 et 8
5	5 et 7
6	6
7	5 et 7
8	4 et 8
9	3 et 9
10	2 et 10
11	1 et 11

Une première régularité remarquable est que toutes les lignes comportent un X à la position horizontale égale à leur numéro.

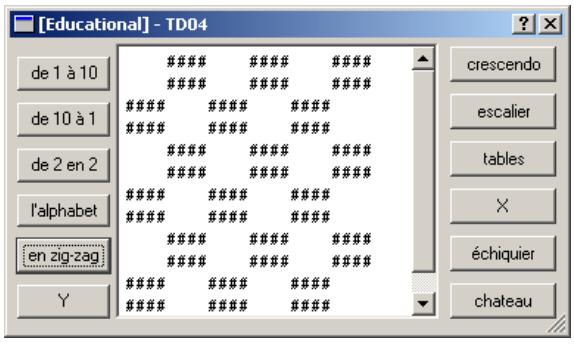
Une seconde régularité est que la position de l'autre X peut toujours être obtenue en soustrayant de 12 le numéro de la ligne.

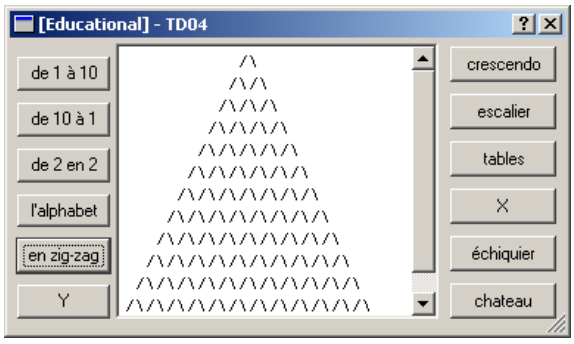
> La ligne 6, qui semblait a priori faire exception (puisqu'elle ne comporte qu'un seul X) rentre en fait dans le cas général : 6 est égal à 12-6, et on peut considérer que cette ligne comporte deux X superposés.

Si nous choisissons d'utiliser une fonction "auxiliaire", nous écrirons donc :

```
//Cette fonction est appelée lorsque l'utilisateur clique sur [X]
void TD04DialogImpl::f_x()
{
afficheur.clear();
for(numeroLigne = 1 ; numeroLigne < 12 ; numeroLigne = numeroLigne + 1)
   ligneX();
}
void TD04DialogImpl::ligneX() //cette fonction est appellée par f_x()
{
int posH;
for(posH = 1; posH < 12 ; posH = posH + 1)
   if (posH == numeroLigne || posH == 12 - numeroLigne)
      afficheur << "X";
   else
      afficheur << " ";
afficheur << "\n";
}
```

7 - A vous de jouer...





Créez les fonctions nécessaires pour que les fonctions associées aux boutons "Echiquier" et "Chateau" produisent les affichages représentés ci-dessus.