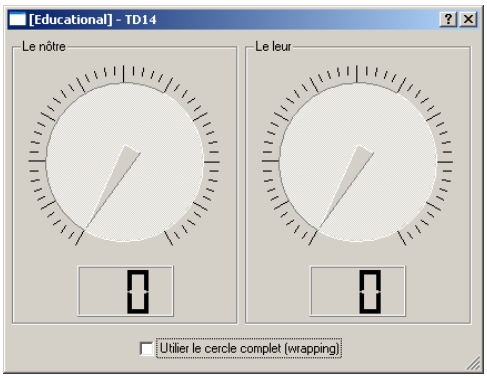
Centre Informatique pour les Lettres et les Sciences Humaines

TD 14 : Et si les widgets proposés par Qt ne me conviennent pas parfaitement ?

L'utilisation de `QDial` au cours des TD 1 et 12 nous a permis de remarquer une de leurs caractéristiques qui peut s'avérer gênante : lorsqu'un widget de ce type permet à l'utilisateur de spécifier une valeur, la "butée" qui arrête l'aiguille en fin de course peut être outrepassée si le pointeur de la souris va trop loin.

> La valeur spécifiée passe alors instantanément d'un extrême à l'autre, ce qui ne correspond sans doute pas à l'intention de l'utilisateur.

Le programme que nous allons écrire utilise un `QDial` modifié, dont la butée n'est pas "outrepassable". Le mécanisme de dérivation et la redéfinition d'une fonction virtuelle vont nous permettre de créer notre propre widget en n'écrivant que quelques lignes de code.



L'interface utilisateur du programme réalisé au cours du TD 14

1 - Création du projet et dessin de l'interface

Utilisez la procédure habituelle (cf. aide mémoire) pour créer un projet nommé TD 14 ☐.

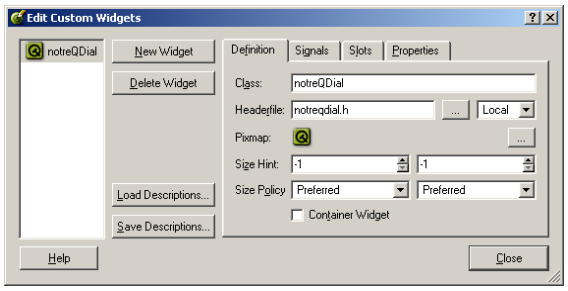
En vous inspirant de la copie d'écran ci-dessus, disposez dans votre dialogue :

- deux groupBox ayant respectivement pour titre "Le nôtre" et "Le leur" ☐.
- deux lcdNumber respectivement nommés `notreLCD` et `leurLCD` ☐.
- une checkBox nommée `checkWrapping` ☐.

Dans la groupBox de droite, insérez un QDial nommé `leLeur` ☐ et réglez sa propriété "notchesVisible" à la valeur `true` ☐.

Le contrôle qui va prendre place dans la groupeBox de gauche ne sera pas un contrôle standard. Il nous faut donc indiquer à Qt Designer comment s'appelle la classe qui le décrit et de quels signaux il dispose.

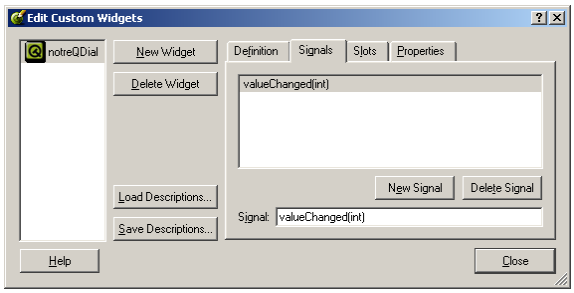
Dans le menu "Tools", catégorie "Custom", choisissez la commande "Edit custom widgets..." ☐.



Dans le dialogue qui apparaît alors, cliquez sur le bouton [New Widget] ☐.

Modifiez le contenu des champs "Class" et "Headerfile" pour leur donner respectivement les valeurs `notreQDial` et notreqdial.h ☐.

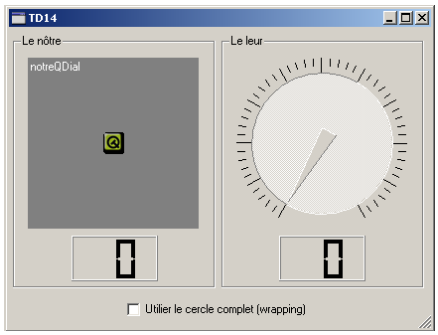
Dans l'onglet "Signals", cliquez sur le bouton [New Signal] et tapez `valueChanged(int)` dans le champ "Signal:" ☐.



Cliquez enfin sur le bouton [Close] ☐.

Dans le menu "Tools", catégorie "Custom", choisissez l'outil "notreQDial" et placez un widget dans la groupeBox de droite ☐.

> Votre maquette de dialogue devrait maintenant ressembler à celle présentée ci-contre.



Donnez à ce nouveau widget le nom `leNotre` ☐.

Etablissez les connexions suivantes ☐ :

	Sender	Signal	Receiver	Slot
✓1	leNotre	valueChanged(int)	notreLCD	display(int)
✓2	leLeur	valueChanged(int)	leurLCD	display(int)
✓3	checkWrapping	stateChanged(int)	TD14Dialog	f_changeWrapping(int)

La troisième connexion exige évidemment la création préalable du slot `f_changeWrapping(int)`.

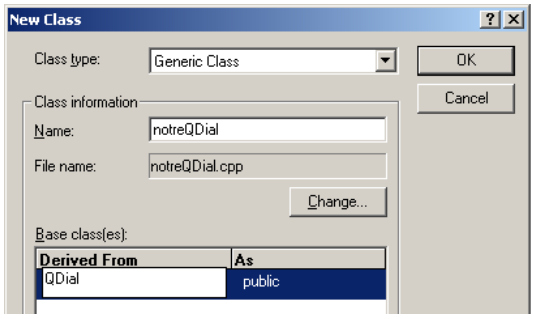
Enregistrez votre dialogue et retournez dans Visual C++ ☐.

2 - La classe `notreQDial`

Notre classe va présenter deux facettes : les instances de `notreQDial` sont des `QDial` (ce qui leur permet de fonctionner) qui ont une particularité (c'est ce qui fait l'intérêt de cette classe). La première facette est obtenue par héritage, la seconde par redéfinition d'une fonction virtuelle de la classe de base.

Dérivation et héritage

Suivez la procédure habituelle (cf. aide-mémoire) pour créer une classe dérivée de `QDial` ☐.



Le code généré par Qt Designer instancie les widgets en passant deux arguments à leur constructeur : l'adresse du widget qui les contient, et leur nom. Il convient donc que notre classe dispose d'un constructeur acceptant ces arguments.

Rappel : les constructeurs ne sont jamais hérités de la classe de base.

Le fichier notreqdial.h devient donc ☐ :

```
1  #include "qdial.h"
2  class notreQDial : public QDial
3  {
4  public:
5     notreQDial(QWidget *parent, QString nom);
6     virtual ~notreQDial();
7  };
```

Le constructeur de `notreQDial` se contente de transmettre ces arguments au constructeur de `QDial` chargé de créer la partie héritée. Les modifications à apporter au fichier notredial.cpp sont donc minimes ☐ :

```
1  notreQDial::notreQDial(QWidget *parent, QString nom) : QDial(parent, nom)
2  {
3  setNotchesVisible(true);
4  }
```

Le constructeur est aussi chargé d'ajuster *(3)* les propriétés du widget que nous n'avons pas pu spécifier avec Qt Designer.

Si vous compilez ☐ et exécutez ☐ votre programme, vous pouvez vérifier que `notreQDial` se comporte exactement comme le `QDial` ordinaire placé à sa droite.

La checkBox est pour l'instant sans effet, puisque nous ne nous en sommes pas encore occupés.

Redéfinition d'une fonction virtuelle

Lorsque l'utilisateur déplace l'aiguille d'un `QDial`, la mise à jour de l'affichage implique l'exécution d'une fonction membre de cette classe nommée `valueChange()`.

> A ne pas confondre avec le signal `valueChanged()` que nous avons connecté au slot `display()` d'un QLCDnumber...

Comme cette fonction est virtuelle, si nous la redéfinissons dans la classe `notreQDial`, c'est cette nouvelle version qui sera exécutée lorsque le code hérité appellera `valueChange()`.

> Les classes `QDial` et `notreQDial` sont exactement dans la situation évoquée par la Leçon 14 (page 4), situation qui peut être schématisée ainsi :

```
class QDial //la classe mère
{
public:
   void fonctionHeritee() {valueChange();}
protected:
   virtual void valueChange();     //fonction redéfinie
};

class notreQDial : public QDial   //la classe fille
{
protected:
   virtual void valueChange();     //redéfinition
};
```

> Lorsqu'elle est exécutée au titre d'une instance de `notreQDial`, la `fonctionHeritee()` ne peut exécuter la version redéfinie de `valueChange()` que dans la mesure où cette dernière est virtuelle. Pour qu'un "cocktail" de code hérité et de code propre fonctionne correctement, il est impératif que la redéfinition ne soit pratiquée que sur des fonctions virtuelles.

Pour interdire le passage direct d'une extrémité de la plage de variation à l'autre, nous allons simplement censurer les déplacements de grande amplitude. Pour calculer l'amplitude du mouvement considéré, il nous faut disposer de la valeur antérieure. Cette valeur sera stockée dans une variable membre.

Ajoutez à la classe `notreQDial` une variable membre de type `int` nommée `ancienneValeur` ☐.

Cette variable doit être initialisée pour que le calcul de l'amplitude du premier déplacement soit possible. Ajoutez donc l'instruction nécessaire *(4)* au constructeur de `notreQDial` ☐ :

```
1  notreQDial::notreQDial(QWidget *parent, QString titre) : QDial(parent, titre)
2  {
3  setNotchesVisible(true);
4  ancienneValeur = value();
5  }
```

Cette `ancienneValeur` étant disponible, la fonction `valueChange()` peut facilement comparer l'amplitude du mouvement proposé avec un seuil arbitraire (le quart de la plage de variation du widget, en l'occurrence). Si le mouvement est raisonnable, il suffit d'appeler la fonction `valueChange()` "ordinaire" *(7)* (qui traitera la situation comme si le widget était un simple `QDial`) et de mémoriser la nouvelle position *(8)*. Si le mouvement est excessif, notre fonction se contente de rétablir la valeur antérieure *(11)*.

```
1  void notreQDial::valueChange()
2  {
3  const int PLAGE = maxValue() - minValue();
4  const int SEUIL = (PLAGE > 3) ? PLAGE / 4 : 1;
5  if(abs(ancienneValeur - value()) <= SEUIL)
6     { //on accepte
7     QDial::valueChange();
8     ancienneValeur = value(); //pour pouvoir évaluer le prochain mouvement
9     }
10 else //on refuse
11    setValue(ancienneValeur);
12 }
```

Ajoutez à votre classe `notreQDial` la fonction décrite ci-dessus ☐.

Si vous compilez et exécutez votre programme, vous pouvez vérifier que `notreQDial` ne se comporte plus comme le `QDial` ordinaire placé à sa droite : arrivé en bout de course, l'aiguille se bloque et refuse de suivre la souris, même si celle-ci dépasse l'autre extrémité de la piste graduée.

3 - La fonction `f_changeWrapping()`

La différence entre le comportement d'un `QDial` et celui d'un `notreQDial` est encore plus manifeste lorsque les widgets sont privés de la "zone neutre" qui occupe normalement leur partie inférieure : le dépassement de la valeur extrême est alors difficile à éviter.

La classe `QDial` dispose d'une propriété nommée "wrapping" qui permet justement d'éliminer cette zone neutre. Comme nous avons lié le signal `stateChanged()` de la `checkBox` présente dans notre dialogue à un slot nommé `f_changeWrapping()`, il suffit de définir ainsi cette fonction pour pouvoir changer le mode de fonctionnement des `QDial`:

```
1  void TD14DialogImpl::f_changeWrapping(int v)
2  {
3  leLeur->setWrapping(v);
4  leNotre->setWrapping(v);
5  }
```

Remarquez, une fois encore, que nous pouvons utiliser `notreQDial` *(4)* exactement comme s'il s'agissait d'un `QDial` ordinaire : du point de vue du code qui l'utilise, la particularité que nous lui avons ajoutée ne l'a privé d'aucune de ses caractéristiques originelles. Cette idée est souvent exprimée en disant qu'une instance de la classe dérivée est *substituable* à une instance de la classe de base.

> En d'autres termes, si un fragment de code quelconque opère sur une instance de la classe de base, on doit pouvoir remplacer cette instance par une instance de la classe dérivée sans avoir à modifier le code en question.

La perte de cette substituabilité est en général le signe d'une erreur dans la conception de la hiérarchie de classes utilisée.

Un autre point concernant la fonction `f_changeWrapping()` mérite d'être souligné. Le lien signal/slot établi à l'aide Qt Designer implique une variable et une fonction qui sont membres de la classe `TD14Dialog`.

> Le partage des responsabilités que nous utilisons exige qu'il en soit ainsi : Qt Designer génère la classe `…Dialog` et ignore tout le reste du programme.

La fonction que nous avons définie ci-dessus est, pour sa part, membre de la classe `TD14DialogImpl`.

> C'est le second volet du partage des responsabilités : nous n'utilisons jamais Visual C++ pour modifier la classe `…Dialog`.

Comment se fait-il donc qu'un clic sur la checkBox provoque l'exécution de notre code ?

L'examen des définitions respectives de ces deux classes permet de comprendre ce qui se passe. Dans le fichier TD14DialogImpl.h, nous constatons que la classe `TD14DialogImp` dérive de la classe `TD14Dialog` :

```
1  #include "td14dialog.h"
2  class TD14DialogImpl : public TD14Dialog
3  {
4     Q_OBJECT
5  public:
6     void f_changeWrapping(int v);
7     TD14DialogImpl(QWidget* parent=0, const char* name=0, bool modal=FALSE, WFlags f=0 );
8  };
```

La classe `TD14DialogImpl` hérite donc de toutes les variables et fonctions définies par Qt Designer conformément aux spécifications que nous avons exprimées à l'aide de l'interface de ce logiciel.

Ceci explique que, bien que nous modifiions la fonction `main()` pour faire en sorte que la fenêtre de base de notre programme soit de type `TD14DialogImp`, l'affichage obtenu contient tous les widgets dont la disposition a été ordonnée dans la classe `TD14Dialog`. En d'autres termes, un `TD14DialogImpl` *est un* `TD14Dialog`.

Si vous consultez le fichier TD14Dialog.cpp, vous verrez que le constructeur de la classe `TD14Dialog` comporte la ligne suivante :

```
connect(checkWrapping, SIGNAL(stateChanged(int)), this, SLOT(f_changeWrapping(int)));
```

L'événement "clic sur la checkBox" déclenche donc l'exécution d'une fonction membre de `TD14Dialog`.

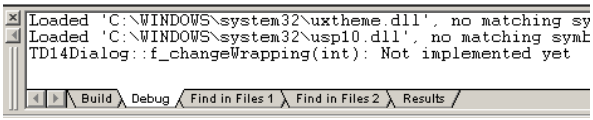
En tant que classe dérivée de `TD14Dialog`, `TD14DialogImpl` hérite (entre autres choses) de tout ce dispositif : la checkBox, la fonction et la connexion qui les lie. Nous savons toutefois que, concrètement, un clic sur la checkBox ne se traduit pas par l'exécution de la fonction héritée mais par celle de la fonction que nous avons définie nous même dans `TD14Dialog`. L'examen de la définition de la classe `TD14Dialog` (TD14Dialog.h) montre clairement pourquoi il en est ainsi :

```
1  class TD14Dialog : public QDialog
2  {
3      Q_OBJECT
4  public:
5      TD14Dialog(QWidget* parent=0, const char* name=0, bool modal=FALSE, WFlags fl=0 );
6      ~TD14Dialog();
7      QGroupBox* groupBox1;
8      QDial* leLeur;
9      QLCDNumber* leurLCD;
10     QGroupBox* groupBox2;
11     notreQDial* leNotre;
12     QLCDNumber* notreLCD;
13     QCheckBox* checkWrapping;
14 public slots:
15     virtual void f_changeWrapping(int);
16 protected slots:
17     virtual void languageChange();
18 private:
19     QPixmap image0;
20 };
```

En résumé :

- Quand vous créez un slot avec QT Designer, vous demandez en fait à celui-ci d'inclure une fonction virtuelle dans la classe …`Dialog` qu'il va écrire pour représenter votre dialogue.
- Quand vous modifiez la fonction `main()` pour lui faire instancier la classe …`DialogImpl`, vous décidez de ne pas travailler directement avec la classe créée par Qt Designer, mais avec une classe dérivée de celle-ci.
- Quand, avec Visual C++, vous ajoutez à votre classe …`DialogImpl` une fonction membre déclarée exactement comme le slot précédemment créé, vous êtes en train de redéfinir une fonction virtuelle héritée de la classe …`Dialog`.
- Lorsque le code généré par Qt Designer et dont a hérité votre classe …`DialogImpl` appelle une fonction slot (parce que l'événement correspondant vient de se produire), c'est la virtualité de cette fonction qui fait que c'est bien sa version redéfinie qui est exécutée.

Vous pouvez facilement vérifier ce point : effacez le mot `virtual` qui précède la déclaration de `f_changeWrapping()` dans la définition de la classe `TD14Dialog`, puis compilez (F7) et exécutez (F5) votre programme. Le clic sur la checkBox ne se traduit plus par une modification de l'apparence des `QDial`, mais par l'affichage d'un simple avertissement :



Si vous regardez comment est définie la fonction `TD14Dialog::f_changeWrapping()`, l'origine de ce message vous apparaîtra clairement.