Centre Informatique pour les Lettres et les Sciences Humaines

TD 13 : Et si Qt Designer n'existait pas ?

Lors du TD 1, nous avons réalisé un premier programme sans avoir à écrire la moindre ligne de code : Qt Designer nous a suffit pour mettre en place quelques éléments d'interface, pour lesquels il a ensuite généré seul le code nécessaire.

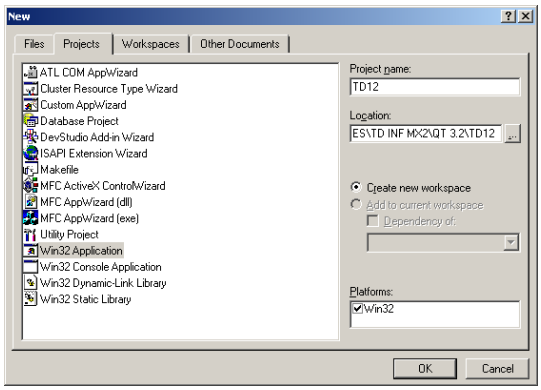
Au cours des TD suivants, nous nous sommes parfois intéressé à l'organisation générale de ce code généré automatiquement, mais notre maîtrise du langage restait insuffisante pour nous permettre de nous affranchir de l'aide apportée par Qt Designer.

L'interface utilisateur des programmes réalisés au cours des TD 1 et 13

L'objectif du TD 13 est de réaliser un programme identique à celui du TD 1, mais sans recourir à Qt Designer. Outre une meilleure compréhension du fonctionnement des programmes que nous réalisons habituellement, cette entreprise nous offrira une illustration de la simplicité d'utilisation et de la puissance du mécanisme de dérivation de classes.

1 - Création du projet

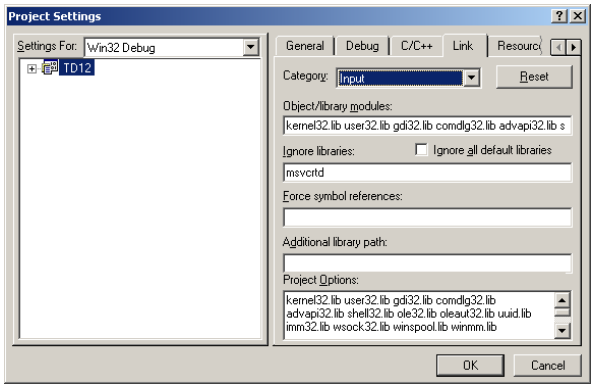
Comme dans le cas du TD 2, le projet que nous allons créer n'est pas un projet Qt, mais un simple projet Windows tel que Visual C++ sait les faire.



Dans Visual C++, choisissez l'option "New..." du menu "File".

Dans l'onglet "Projects" du dialogue qui s'ouvre alors (cf. ci-contre), choisissez "Win 32 Application" et indiquez le nom et l'emplacement souhaité pour votre projet.

Dans le dialogue suivant, choisissez l'option "A simple Win32 Application".



Une fois le projet créé, cliquez sur la commande "Settings" du menu Project".

Dans l'onglet "Link" du dialogue qui s'ouvre alors, sélectionnez la "category" "Input".

Dans le champ "Ignore libraries", tapez msvcrtd.

> Cette manipulation ne sert qu'à éviter l'émission par le linker d'un message d'avertissement sans intérêt.

Le projet créé par Visual C++ est aussi simple que promis : il ne comporte qu'une fonction globale nommée `WinMain()`, qui est exécutée lorsque le programme est lancé. Dans son état actuel, le projet n'utilise pas la librairie Qt : ouvrez le fichier TD13.cpp et insérez-y les lignes suivantes :

```
1  // TD13.cpp : Defines the entry point for the application.
2  //
3  #include "stdafx.h"
4  #include "qstring.h"
5  int APIENTRY WinMain(HINSTANCE hInstance,
                        HINSTANCE hPrevInstance,
                        LPSTR     lpCmdLine,
                        int       nCmdShow)
6  {
7  QString s = QString::number(36); //teste la présence de Qt
8  return 0;
9  }
```

Très logiquement, une tentative de compilation se soldera par un message d'erreur :

```
--------------------Configuration: TD12 - Win32 Debug--------------------
Compiling...
StdAfx.cpp
Compiling...
TD12.cpp
D:\TD12\TD12.cpp(5) : fatal error C1083: Cannot open include file: 'qstring.h': No such file or directory
Error executing cl.exe.

TD12.exe - 1 error(s), 0 warning(s)
```

Pour rendre Qt disponible, il faut cliquer sur le bouton prévu à cet effet, le sixième et avant-dernier de ceux proposés dans la barre d'outils Qt : 

Une fois cette formalité remplie ☐, une nouvelle compilation devrait, cette fois, réussir ☐.

> Si Visual C++ persiste à vous objecter qu'il ne trouve pas le fichier qstring.h, fermez votre projet (commande "Close Workspace" du menu "Fichier") et rouvrez-le (première proposition de la commande "Recent Workspace" du menu "Fichier") : tout devrait rentrer dans l'ordre...

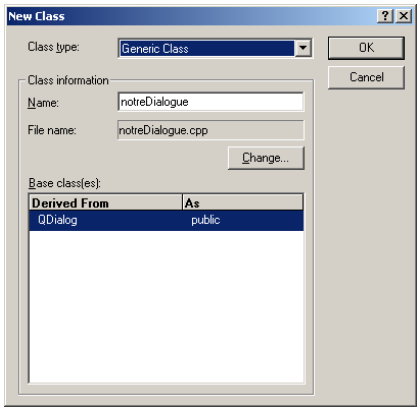
Cette réussite prouve non seulement que Visual C++ sait maintenant où trouver les fichiers .h définissant les classes de la librairie Qt, mais également que celle-ci est incluse (liée) dans le projet : la ligne 7 exige l'exécution d'une fonction de la classe `QString` qui n'est pas définie (mais seulement déclarée) dans qstring.h. Si le linker ne proteste pas, c'est qu'il a trouvé quelque part le code définissant cette fonction, ce qui signifie bien que la librairie Qt est désormais disponible dans le cadre de ce projet.

2 - Création de `notreClasseDialogue`

Il faut maintenant ajouter au projet une classe qui va nous permettre de décrire l'interface graphique dont nous avons besoin.

> Au besoin, consultez l'aide mémoire "Créer une classe et ajouter des membres".

Dans le dialogue de création de classe, indiquez dans le champ "Name" que la classe sera nommée `notreClasseDialogue` ☐.



Dans la colonne "Derived From" du tableau "Base class(es)", indiquez que cette classe est dérivée de `QDialog` ☐.

> Nous souhaitons utiliser un héritage public, c'est donc ce mot (proposé par défaut) qui doit figurer dans la colonne "As" du tableau "Base class(es)".

Lorsque vous cliquez sur [OK] ☐, Visual C++ vous prévient honnêtement qu'il ignore tout de la classe `QDialog` et qu'il vous faudra ajouter vous-même la directive `#include` correspondante dans le fichier notreClasseDialogue.h.

Ouvrez le fichier notreClasseDialogue.h et ajoutez-lui la directive nécessaire ☐ :

```
1  #include "qdialog.h"
2  class notreClasseDialogue : public QDialog
3  {
4  public:
5     notreClasseDialogue();
6     virtual ~notreClasseDialogue();
7  };
```

> Le lien entre la classe `QDialog` et le fichier qdialog.h est évident pour nous grâce aux conventions de dénomination adoptées par Trolltech. Il est légitime (et même très souhaitable) que Microsoft Visual C++ ne cherche pas à prendre en compte une "évidence" de ce genre.

Bien que `notreClasseDialogue` donne l'impression d'être vide (les corps du constructeur et du destructeur créés par Visual C++ ne comportent pas la moindre instruction, comme vous pouvez le constater en ouvrant le fichier notreClasseDialogue.cpp), elle permet déjà beaucoup de choses : du fait du lien de dérivation publique, les instances de `notreClasseDialogue` sont aussi des `QDialog`, ce dont nous allons immédiatement tirer partie.

3 - Utilisation de notreClasseDialogue

Pour que l'exécution WinMain() se traduise par l'apparition d'un programme comportant une fenêtre graphique, il faut que cette fonction procède à quatre opérations :

- Création d'une instance de la classe QApplication *(9-10)*;
- Création d'une instance de la classe notreClasseDialogue *(11-12)*;
- Adoption de cette instance comme interface graphique de la QApplication *(13)*;
- Mise en route de la QApplication *(14)*.

```
1  #include "stdafx.h"
2  #include "qapplication.h"
3  #include "notreClasseDialogue.h"
4  int APIENTRY WinMain(HINSTANCE hInstance,
5                       HINSTANCE hPrevInstance,
6                       LPSTR     lpCmdLine,
7                       int       nCmdShow)
8  {
9  int bidon = 0; //nécessaire au constructeur de QApplication
10 QApplication leProgramme(bidon, &lpCmdLine);
11 notreClasseDialogue * adresseDeNotreDialogue = new notreClasseDialogue;
12 adresseDeNotreDialogue->show();
13 leProgramme.setMainWidget(adresseDeNotreDialogue);
14 return leProgramme.exec();
15 }
```

> Le constructeur de la classe QApplication a deux paramètres, destinés à permettre l'analyse des éventuels arguments passés lors du lancement du programme au moyen d'une ligne de commande. Bien que ce dispositif soit sans grand intérêt ici, il nous faut respecter la syntaxe, ce qui exige la création d'une variable capable de satisfaire le premier paramètre, qui est de type "référence à un int".

L'association du dialogue à la QApplication se fait *(13)* à l'aide de la fonction setMainWidget(). Cette fonction réalise un véritable "transfert de propriété" qui rend la QApplication responsable de la destruction de l'objet qui lui est confié. Comme nous le savons (cf. Leçon et TD 12), cette façon de procéder exige que l'objet en question soit créé *(11)* par allocation dynamique.

> Les QDialog naissent invisibles, ce qui permet, selon les besoins, de les modifier en toute discrétion avant de les offrir en spectacle à l'utilisateur. La création de notre instance *(11)* est donc suivie de l'appel d'une fonction qui la rend visible *(12)*.
>
> La mise en route de la QApplication se fait simplement en appelant la fonction exec(), dont la valeur de retour constituera aussi celle de la fonction WinMain().

Compilez ☐ et exécutez votre programme ☐.

Le dialogue qui apparaît est évidemment vide, mais :
- il apparaît ;
- il peut être déplacé et redimensionné à l'écran, comme n'importe quelle autre fenêtre ;
- il dispose d'une barre de titre et d'une case de fermeture dont l'usage permet non seulement de refermer la fenêtre, mais aussi de mettre fin à l'exécution du programme.

Ces caractéristiques, fondamentales pour tout programme Windows, sont obtenues à peu de frais : une classe dérivée et une demi-douzaine de lignes de code. La relation de dérivation fait que les instances de notreClasseDialogue sont des QDialog, ce qui permet à la fonction QApplication::setMainWidget() d'accepter l'adresseDeNotreDialogue comme valeur servant à initialiser son paramètre (qui est de type "pointeur sur QDialog").

> Il est clair que setMainWidget() ne peut en aucun cas disposer d'un paramètre de type "pointeur sur notreClasseDialogue" : les ingénieurs de Trolltech pouvaient difficilement deviner que c'était cette classe que nous aurions envie d'inventer...

Comme leProgramme accède à notre dialogue en utilisant un pointeur sur QDialog, il n'a accès qu'aux fonctions membre que notre dialogue a hérité de cette classe. Ce sont ces fonctions qui permettent d'obtenir l'aspect visuel et les fonctionnalités de base que nous avons observés.

Bien entendu, ces fonctions héritées sont pour l'instant les seules dont dispose notre dialogue, et nous aurions obtenu exactement le même résultat en nous épargnant la création de `notreClasseDialogue` et en instanciant simplement `QDialog` à la ligne *11*. L'utilisation de `notreClasseDialogue` va se justifier dès que nous doterons cette classe de fonctions propres (c'est à dire non héritées de `QDialog`).

> Les instances de `notreClasseDialogue` vont dont avoir deux aspects : en tant que `QDialog`, elles disposent de caractéristiques standard qui permettent aux programmes de les utiliser comme fenêtre de base en ignorant tout de leurs spécificités. En tant qu'instance d'une classe dérivée, elles peuvent en outre disposer d'autres caractéristiques, ce qui va nous permettre de leur faire décrire une interface utilisateur particulière, propre à notre projet.

4 - Spécialisation de `notreClasseDialogue`

Qu'elle soit effectuée avec Qt Designer ou spécifiée directement en C++, la création d'un dialogue implique les mêmes opérations : création des widgets, "mise en page" du dialogue et établissement des connexions entre signaux et slots.

Création des widgets

Pour que notre interface ressemble à celle du TD 1, il faut ajouter quatre widgets à notre dialogue : un `QSlider`, un `QLCDNumber`, un `QDial` et un `QLabel`. Ces widgets devront être "ajustés" (contenu du `QLabel`, plage de variation du `QSlider` et du `QDial`, taille et position des quatre widgets, etc.) avant que le dialogue ne soit affiché. Le constructeur de `notreClasseDialogue` est un candidat idéal pour assurer cette mission : il sera automatiquement exécuté lors de toute instanciation de la classe, ce qui garantit la naissance de dialogues conformes à nos spécifications. Soulignons une fois encore que

Les widgets doivent être créés par allocation dynamique.

Comme ces widgets doivent évidemment continuer à être utilisables après la fin de l'exécution du constructeur, leurs adresses doivent être stockées dans des variables membre et non dans des variables locales. Si l'on décide que ces variables s'appelleront respectivement `curseur`, `lcd`, `quadrant` et `message`, le corps du constructeur débutera donc ainsi :

```
1  notreClasseDialogue::notreClasseDialogue()
2  {
3  curseur = new QSlider(this);
4  curseur->setRange(0,99);
5  curseur->setMinimumSize(40,50);
6  lcd = new QLCDNumber(this);
7  lcd->setSegmentStyle(QLCDNumber::Flat);
8  lcd->setFixedSize(100,100);
9  lcd->setNumDigits(2);
10 quadrant = new QDial(this);
11 quadrant->setMinimumSize(200,200);
12 message = new QLabel(this);
13 message->setText("Programmer, c'est trop facile !");
14 message->setAlignment(AlignHCenter);
15 QFont police("Helvetica", 18, QFont::Bold);
16 message->setFont(police);
```

> Remarquez que la création des widgets s'accompagne de la transmission de l'adresse de l'instance de `notreClasseDialogue` en cours de création au constructeur de widget concerné. Cette adresse permet au widget de s'inscrire dans la liste des objets qui devront être détruits lors de la destruction du dialogue.
>
> Chaque widget fait en outre l'objet de quelques réglages (*4-5*, *7-9*, *11* et *13-16*) effectués à l'aide de fonctions membre qui fonctionnent un peu comme le "Property Editor" de Qt Designer.

Mise en page

La librairie Qt offre un dispositif qui simplifie considérablement le positionnement des widgets dans un dialogue lorsqu'on ne souhaite pas obtenir des effets très particuliers.

> Dans le projet du TD 8 (la patience des pharaons), par exemple, l'effet visuel recherché imposait un calcul fastidieux pour positionner les cartes. Ce genre de calcul pourrait aussi être utilisé ici, mais il y a une meilleure solution.

Les classes `QHBoxLayout` et `QVBoxLayout` permettent de créer des widgets analogues à des `groupBox`, mais destinés à un tout autre usage.

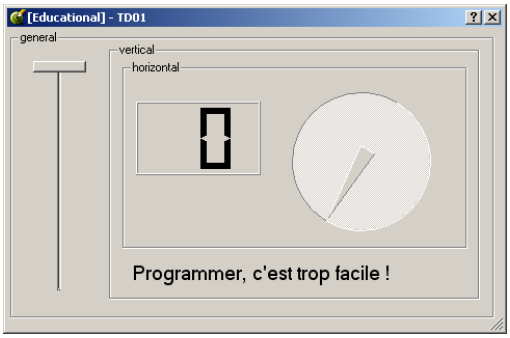
Les `groupBox`, comme leur nom l'indiquent, regroupent plusieurs widgets pour suggérer à l'utilisateur que ceux-ci ont quelque chose en commun.

Les `QHBoxLayout` et `QVBoxLayout` permettent également de créer des groupes de widgets, mais ils restent invisibles pour l'utilisateur et servent à spécifier les positions relatives des widgets qu'ils contiennent.

Si l'on dispose d'un `QHBoxLayout`, les widgets qu'on y insérera seront placés les uns *à côté* des autres. Dans le cas d'un `QVBoxLayout`, les widgets insérés prendront au contraire place les uns *en dessous* des autres.

Si l'on sait qu'il est possible d'insérer un layout (et les widgets qu'il contient) dans un autre layout, on peut concevoir la mise en page de notre dialogue de la façon suivante :

- le `QLCDNumber` et le `QDial` sont dans un `QHBoxLayout` nommé `horizontal` ;
- `horizontal` et le `QLabel` sont dans un `QVBoxLayout` nommé `vertical` ;
- le `QSlider` et `vertical` sont dans un `QHBoxLayout` nommé `general`.



> Dans l'image ci-contre, les layouts sont figurés par des `groupBox`, de façon à rendre plus intelligible l'organisation décrite ci-dessus.

Il devient alors très facile de réaliser notre mise en page :

```
17  QHBoxLayout * horizontal = new QHBoxLayout;
18  horizontal->addWidget(lcd);
19  horizontal->addWidget(quadrant);
20  QVBoxLayout * vertical = new QVBoxLayout;
21  vertical->addLayout(horizontal);
22  vertical->addWidget(message);
23  QHBoxLayout * general = new QHBoxLayout(this);
24  general->setMargin(20);
25  general->setSpacing(20);
26  general->addWidget(curseur);
27  general->addLayout(vertical);
```

> L'usage des fonctions `addWidget()` et `addLayout()` se passe de commentaires. Les fonctions `setMargin()` et `setSpacing()` permettent d'indiquer qu'on ne veut pas que les widgets soient trop près des bords du dialogue ou trop près les uns des autres.

Remarquez que le layout `general` est construit *(23)* en lui transmettant l'adresse de l'instance de `notreClasseDialogue` en cours de création. Ce layout occupera donc l'ensemble de l'espace disponible dans le dialogue.

La création des layout `horizontal` *(17)* et `vertical` *(20)*, en revanche, ne s'accompagne d'aucun passage de paramètre : l'espace qu'ils occuperont est déterminé par leur insertion (*21* et *27*) dans d'autres layouts.

Connexions

Les connections entre signaux et slots sont établies à l'aide de la fonction `connect()`:

```
28 connect(curseur,  SIGNAL(valueChanged(int)), lcd,      SLOT(display(int)));
29 connect(curseur,  SIGNAL(valueChanged(int)), quadrant, SLOT(setValue(int)));
30 connect(quadrant, SIGNAL(valueChanged(int)), curseur,  SLOT(setValue(int)));
31 }
```

Donnez au constructeur de `notreClasseDialogue` le contenu suggéré ci-dessus ☐.

Le fichier notreClasseDialogue.h

Pour que le programme soit terminé, il faut encore ajouter la déclaration des variables membre et les directives d'inclusions nécessaires ☐ :

```
1  #include "qdialog.h"
2  #include "qlabel.h"
3  #include "qslider.h"
4  #include "qlcdnumber.h"
5  #include "qdial.h"
6  #include "qlayout.h"
7  #include "qfont.h"
8  class notreClasseDialogue : public QDialog
9  {public:
10   notreClasseDialogue();
11   virtual ~notreClasseDialogue();
12 protected:
13   QSlider * curseur;
14   QLCDNumber * lcd;
15   QDial * quadrant;
16   QLabel * message;
   };
```

Vous pouvez maintenant compiler ☐ et exécuter votre programme ☐.

5 - Prolongement : un bouton qui appelle une fonction

Le TD 1 devant être effectué par des étudiants qui n'ont pas encore commencé à apprendre le langage, il ne comporte aucune création de fonction. Si vous souhaitez créer une application complète sans utiliser Qt Designer, il vous faudra sans doute lier des signaux émis par des widgets à des fonctions de votre cru.

Pour illustrer la marche à suivre, nous allons ajouter à notre programme un bouton permettant simplement d'afficher un message confirmant l'exécution de la fonction qui lui est associée.

Pour que son exécution puisse être déclenchée par un bouton, la fonction en question doit être un slot. La notion de slot ne faisant pas partie du langage C++, Trolltech fournit une "extension" du langage qui repose sur quelques mots clés spécifiques et sur l'invocation d'un programme de pré-traitement qui traduit les fichiers utilisant ces mots clés en textes C++ standard, qui peuvent ensuite être confiés au compilateur.

Concrètement, la définition de la classe prend l'aspect suivant :

```
1  class notreClasseDialogue : public QDialog
2  {
3  Q_OBJECT
4  public:
5    notreClasseDialogue();
6    virtual ~notreClasseDialogue();
7  public slots:
8    void f_bouton();       //la nouvelle fonction
9  protected:
10   QLabel * message;
11   QSlider * curseur;
12   QLCDNumber * lcd;
13   QDial * quadrant;
14   QPushButton * bouton;  //pour stocker l'adresse du nouveau widget
15 };
```

Ajoutez une directive d'inclusion de qpushbutton.h et les lignes 3, 7, 8 et 14 à la définition de votre classe ☐ et, pour que ce fichier soit traduit en "pur C++" avant d'être compilé, cliquez ☐ sur le 7° et dernier bouton de la barre d'outils Qt (Moc).

Vérifiez que la compilation de votre projet ne donne lieu à aucune erreur ☐.

> Si vous n'avez pas cliqué sur le bouton MOC (l'outil de traduction de Trolltech s'appelle le **M**eta **O**bject **C**ompiler), le linker va signaler des "unresolved external symbol" concernant, notamment, une fonction nommée `qt_property()`.

Il reste encore à modifier le constructeur pour créer et positionner le bouton et à le lier au slot `f_bouton()` :

```
1  notreClasseDialogue::notreClasseDialogue()
2  {
3  curseur = new QSlider(this);
4  curseur->setRange(0,99);
5  curseur->setMinimumSize(40,50);
6  lcd = new QLCDNumber(this);
7  lcd->setSegmentStyle(QLCDNumber::Flat);
8  lcd->setFixedSize(100,100);
9  lcd->setNumDigits(2);
10 quadrant = new QDial(this);
11 quadrant->setMinimumSize(200,200);
12 message = new QLabel(this);
13 message->setText("Programmer, c'est trop facile !");
14 message->setAlignment(AlignHCenter);
15 QFont f("Helvetica", 18, QFont::Bold);
16 message->setFont(f);
17 bouton = new QPushButton(this);
18 bouton->setText("Appeler f_bouton()");
19 QHBoxLayout *horizontal = new QHBoxLayout;
20 horizontal->addWidget(lcd);
21 horizontal->addWidget(quadrant);
22 QVBoxLayout *vertical = new QVBoxLayout;
23 vertical->addLayout(horizontal);
24 vertical->addWidget(message);
25 vertical->addWidget(bouton);
26 QHBoxLayout * general = new QHBoxLayout(this);
27 general->setMargin(20);
28 general->setSpacing(20);
29 general->addWidget(curseur);
30 general->addLayout(vertical);
31 connect(curseur, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)));
32 connect(curseur, SIGNAL(valueChanged(int)), quadrant, SLOT(setValue(int)));
33 connect(quadrant, SIGNAL(valueChanged(int)), curseur, SLOT(setValue(int)));
34 connect(bouton, SIGNAL(clicked()), this, SLOT(f_bouton()));
35 }
```

Il faut finalement définir la fonction `f_bouton()` :

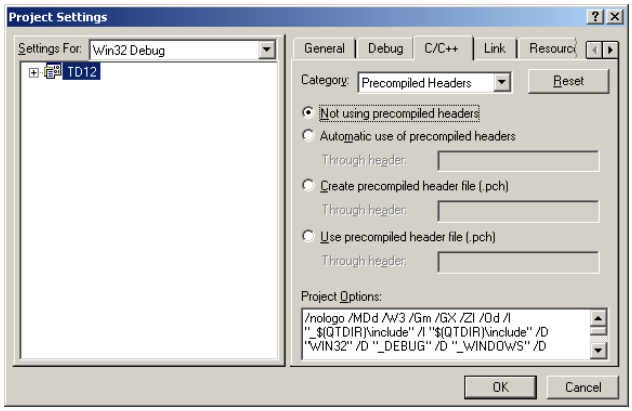
```
1  void notreClasseDialogue::f_bouton()
2  {
3  QMessageBox::information(0, "TD 13", "Exécution de f_bouton()");
4  }
```

Une fois ces ajustements faits (sans oublier d'inclure qmessagebox.h) ☐, vous pouvez compiler ☐ et essayer cette nouvelle version du programme ☐.

Si vous vous heurtez à une erreur de compilation du type "unexpected end of file while looking for pre-compiled header directive":

Utilisez la commande "Settings" du menu "Project" pour ouvrir le dialogue représenté ci-contre .

Dans l'onglet C/C++, category "Precompiled Headers", cliquez sur l'option "Not using Precompiled Headers" .



6 - Conclusion

Créer une classe dérivée d'une classe existant déjà est très simple.

L'intégration du code généré par Qt designer à nos projets habituels s'appuie automatiquement sur ce mécanisme, qui peut aussi être mis en place pour créer "à la main" des programmes utilisant Qt pour gérer leur interface graphique. Dans ce cas :

Les classes dont certains membres propres sont des slots ou des signaux doivent être signalées comme ayant besoin d'être MOCées avant d'être compilées.

Un seul clic sur le bouton suffit pour que la classe soit prétraitée avant chaque compilation.

L'utilisation de Qt dans un projet Windows "classique" donne parfois lieu à des erreurs lors de la compilation ou de l'édition des liens. Les trois cas les plus fréquents sont faciles à résoudre :

Symptôme	Résolution
Message d'erreur du compilateur : "Cannot open include file : "q????.h : No such file"	Rendre Qt disponible dans le projet en cliquant sur le bouton
Message d'erreur du linker : "unresolved external symbol ... qt_property()"	Faire traduire en C++ standard les idiomes Trolltech utilisés dans le fichier .h de votre classe de dialogue en cliquant sur
Message d'erreur du compilateur : "unexpected end of file while looking for pre-compiled header directive"	Désactivez l'utilisation des entêtes précompilés (Menu "Project", commande "Settings", onglet C/C++, catégorie "Precompiled Headers")