

Table des matières

1	Présentation	3
2	Les prérequis	4
2.1	Le compilateur C++	4
2.2	Qt	4
2.2.1	Windows	4
2.2.2	Linux	5
2.2.3	Compilation de Qt	5
2.3	Exuberant Ctags	6
2.4	Gdb	6
3	Installer QDevelop	7
3.1	Depuis les binaires	7
3.2	Depuis les sources	7
4	Modèle objet de Qt	7
5	Signaux et Slots	8
6	Démarrage	10
6.1	Contrôle des outils externes	10
6.2	Présentation de l'interface	11
6.2.1	L'explorateur de fichiers	12
6.2.2	L'explorateur de classes	12
6.2.3	L'éditeur de fichiers sources	13
6.2.3.1	Complétion de code	15
6.2.4	Utiliser l'aide	15
6.2.5	Bases de données	16
7	Projets Qt	17
7.1	Création	17
7.2	Propriétés	19
7.3	Éditions des fichiers	21
7.3.1	Ressources	21
7.3.2	Dialogues	21
7.3.3	Traductions	22
7.3.4	Ajouter un nouvel élément	24
7.4	Ajouter une classe	24
7.5	Ajouter une méthode à une classe	26
7.6	Ajouter une variable à une classe	27
7.7	Ajouter une portée	27
7.8	Sous-classage	28
7.9	Génération	31
7.10	Exécution	32
7.11	Paramètres du programme	32
7.12	Débogage	32

1 Présentation

Beaucoup de développeurs, particulièrement sous Linux programment de manière « artisanale ». C'est-à-dire qu'ils modifient le code dans un éditeur de texte comme vi, emacs, kate ou gedit puis compilent les programmes dans un terminal. Cette méthode, adaptée au développeur expérimenté, présente l'avantage d'un contrôle total sur le processus de fabrication des programmes même si une certaine perte de temps se produit lors de la bascule entre l'éditeur et le terminal. Les développeurs débutants, en revanche, peuvent être complètement perdus face à cette manière de procéder. En effet, comment créer facilement un projet, ajouter des fichiers, compiler, déboguer. Autant de problèmes qui souvent s'ajoutent à des difficultés liées à l'apprentissage du langage lui-même. QDevelop est né du besoin d'un outil unifié permettant d'effectuer toutes les opérations nécessaires à la création d'une application Qt. C'est un environnement de développement intégré plus connu en Anglais sous le nom d'IDE (Integrated Development Environment) et permet de regrouper dans un seul outil, toutes les fonctionnalités nécessaires à la production de programmes en Qt.

QDevelop n'est pas le seul IDE dédié à Qt et lorsque son développement à commencé, d'autres IDE étaient déjà en cours de développement. Il s'agit d'Eduyk, Monkey Studio et Cobras. Ces projets offrent des fonctionnalités similaires à QDevelop. Mais à l'époque (juin 2006), ils ne me convenaient pas totalement et j'ai donc décidé de démarrer mon propre projet. Le but était de disposer dans mes deux environnements habituels, Linux et Windows¹, d'un outil permettant de développer en Qt de la même manière avec un IDE unique. Car chacun des deux systèmes possèdent des environnements de développement performants, bien plus performants que les « petits » IDE dédiés. Mais le gros inconvénient est qu'ils ne fonctionnent que dans leur système d'exploitation respectif. Citons bien-sûr Visual Studio sous Windows et Kdevelop sous Linux. Très performants mais aussi limités à un seul système. Kdevelop propose de gérer des projets Qt mais créé une multitude de fichiers autour et reste quand même assez compliqué à utiliser. Visual Studio quant à lui n'est officiellement compatible qu'avec la version commerciale de Qt. L'autre impératif concernait la possibilité de copier le répertoire d'un projet puis d'ouvrir ce projet dans l'autre environnement sans avoir à modifier ou adapter quoi que ce soit. En créant un IDE capable de lire et d'écrire les fichiers projets créés par qmake, cela donne également la possibilité de compiler en ligne de commande un projet créé dans QDevelop (par exemple sur une machine où il n'est pas installé) et à l'inverse d'ouvrir dans QDevelop un projet précédemment généré en ligne de commande. Mais rester compatible avec qmake apporte quelques inconvénients. En effet les fichiers projet Qt peuvent contenir quelques éléments très difficiles à interpréter dans un IDE. L'exemple le plus significatif est la fonction *for* qui permet d'exécuter une action dans une boucle:

```
LIST = 1 2 3
for(a, LIST):exists(file.$${a}):message(I see a file.$${a}!)
```

Dans l'exemple ci-dessus, tiré de la documentation de *qmake*, pour chaque occurrence de la liste LIST va être affiché un message. Ce genre de fonction ne peut pas être traité par QDevelop. Donc vous l'avez compris, une compatibilité totale avec les fichiers projet de Qt n'est pas effective. Néanmoins, et même si c'est encore perfectible, la grande majorité des fichiers projets Qt peut être

¹QDevelop fonctionne sous Linux, Windows, Mac OS X, certains Unix, BSD, en 32 et 64 bits. De manière générale, si Qt existe pour un environnement, QDevelop également.

ouvert dans QDevelop. Il n'y aura bien entendu aucun problème et c'est la moindre des choses, si le fichier projet a préalablement été créé dans l'IDE.

2 Les prérequis

Afin de créer et compiler des programmes, QDevelop appelle un certain nombre de programmes externes qui lui sont nécessaires.

2.1 Le compilateur C++

Afin de compiler QDevelop lui-même puis ensuite des projets Qt, un compilateur C++ est requis. Sous Linux, il suffit d'installer les paquets de développement C++ qui fourniront le compilateur *g++* ainsi que les outils *make* et *gdb*. Sous Windows, c'est l'ensemble MinGW (qui signifie Minimal GNU for Windows), un portage du compilateur GNU sur cette plateforme qui fournit les mêmes outils cités précédemment. A noter néanmoins que *make* porte sous Windows le nom de *mingw32-make* afin de le différencier d'autres outils du même nom qui peuvent coexister sous Windows.

2.2 Qt

QDevelop est écrit en Qt et est destiné à construire des programmes basés sur cette même bibliothèque. Le minimum requis est donc que Qt soit installé dans votre environnement. QDevelop nécessite au minimum la version 4.2 tout en s'accommodant très bien de la version 4.3. Il existe plusieurs méthodes pour installer Qt en fonction de votre système.

2.2.1 Windows

Trolltech propose sur son site en téléchargement la dernière version de Qt sous deux formes, avec ou sans le compilateur MinGW. A noter que ce dernier est indispensable à la construction des programmes (à commencer par QDevelop lui-même) et que vous devez choisir la version l'intégrant s'il n'est pas déjà installé sur votre système. Cette version fournit la version Release de la bibliothèque Qt et ne permet pas le débogage. Afin de permettre le débogage des programmes, il sera nécessaire de recompiler Qt en version Debug. Une entrée dans le menu démarrer est normalement présente pour effectuer cette compilation.

2.2.2 Linux

Les distributions récentes proposent souvent les paquets de développement Qt4. Le mieux dans ce cas est de les installer grâce au gestionnaire de paquets habituels. La version disponible sera sans doute antérieure à celle qui est proposée sur le site de Trolltech mais suffira si elle est supérieure ou égale à 4.2. Lorsque les paquets sont proposés dans une distribution, certaines modifications peuvent-être apportées concernant le nom des programmes. Par exemple, sur *Ubuntu Feisty*, afin de les différencier de leur homologues Qt3, les programmes pour Qt4 sont nommés *qmake-qt4*, *designer-qt4* etc. C'est pour cette raison qu'il faut configurer dans le dialogue « Outils » le nom des programmes Qt à appeler. Ce paramétrage sera effectué lors du premier démarrage de QDevelop.

2.2.3 Compilation de Qt

Si votre distribution Linux ne propose pas les paquets de développement de Qt ou si vous désirez affiner le paramétrage de la bibliothèque, une compilation à partir des sources est nécessaire. Après avoir téléchargé² puis décompressé l'archive, déplacez-vous dans le nouveau répertoire créé. Il faut en premier configurer votre future version de Qt. Cela s'effectue dans un terminal (un cmd.exe sous Windows) en lançant la commande `./configure`. QDevelop requière le support de base de données SQLITE afin de stocker des informations sur les projets. Afin d'activer ce support, il faut donc saisir :

```
./configure -qt-sql-sqlite
```

Si vous désirez permettre le débogage de vos applications, il est nécessaire de construire Qt en version Debug. Dans ce cas, il faudra employer:

```
./configure -qt-sql-sqlite -debug
```

L'idéal étant de le faire en première fois en version Release (qui est la valeur par défaut) puis en version Debug afin de disposer des deux modes.

A noter que l'ensemble des options possibles peut être obtenu en entrant `./configure -help`. Une fois configuré, il faut construire la bibliothèque. Cela s'effectue en lançant la commande:

```
make (ou mingw32-make.exe sous Windows)
```

puis l'installer par:

```
make install (ou mingw32-make.exe install)
```

Une fois Qt installé, il reste à configurer les variables d'environnement afin de faire connaître au système cette nouvelle bibliothèque. Si vous avez installé Qt à partir de paquets fournis dans votre distribution, vous n'avez rien à faire de plus. Dans le cas contraire, il est nécessaire de configurer trois variables:

- **PATH:** Cette variable existe déjà sur votre système et il suffit d'ajouter le chemin vers le répertoire `bin` de votre installation de Qt. Exemple: `/usr/local/Trolltech/Qt-4.3.0/bin`
- **QTDIR:** Cette variable doit être créée et doit contenir le chemin vers votre installation de Qt. Exemple: `QTDIR=/usr/local/Trolltech/Qt-4.3.0/`
- **QMAKESPEC:** Cette variable est utilisée par Qt pour connaître l'environnement de développement dans lequel il se trouve. Il s'agit d'une combinaison indiquant la plateforme et le compilateur que vous employez sur votre système. Sous Linux cette variable contiendra probablement `linux-g++` alors que sous Windows ce sera `win32-g++`. Exemple:
`QMAKESPEC=linux-g++`

Sous Windows ces trois variables d'environnement peuvent être ajoutées dans le panneau de configuration. Sous Linux, le plus simple est sans doute d'ajouter ces paramètres à un script de connexion. Par exemple dans ma distribution Ubuntu, j'ai ajouté dans le fichier `~/.bashrc` le contenu suivant:

²Il n'est question ici comme dans l'ensemble de ce document que de la version Open sources de Qt. Donc lorsqu'il est indiqué de télécharger Qt, cela sous entend la version Open Source quelque soit le système utilisé.

```
export PATH=/usr/local/Trolltech/Qt-4.3.0/bin:$PATH
export QTDIR=/usr/local/Trolltech/Qt-4.3.0/
export QMAKESPEC=linux-g++
```



Les trois variables d'environnement ci-dessus ne sont pas indispensables si vous utilisez uniquement QDevelop puisque tous les chemins y sont définis. En revanche, elles sont utiles pour compiler en ligne de commande.

2.3 Exuberant Ctags

Ctags est un outil qui extrait des marquants dans des fichiers sources (C++ dans notre cas). Ces marquants, écrits dans un fichier formaté, permettent de trouver rapidement toutes les classes et les variables contenues dans les fichiers scannés. QDevelop peut fonctionner sans ctags mais il faut reconnaître que ce dernier est devenu indispensable au fil des versions. En effet, il est utilisé pour la complétion de code et pour renseigner l'explorateur de classes. Deux fonctionnalités très importantes qui apportent un confort d'utilisation dont il est difficile de se passer. Sous Windows, ctags peut être téléchargé à partir de la page du projet : <http://ctags.sourceforge.net/>

Sous Linux, ctags est présent dans les distributions. Attention néanmoins car il existe souvent sous Linux deux versions de ctags. Une première qui ne convient pas est destinée à l'éditeur de texte Emacs. La deuxième nommée *ctags-exuberant* est en revanche la version qu'il convient d'installer.

2.4 Gdb

Le débogueur de GNU est un programme qui donne la possibilité de déboguer des programmes exécutables afin de traquer les erreurs de conception. Cet outil n'est indispensable que si vous désirez utiliser des outils de débogage. Dans ce cas pensez également à reconstruire la bibliothèque Qt afin d'y inclure les symboles de débogage.



Lors du lancement de QDevelop, la présence de ces outils est contrôlée et si l'un d'entre-eux est manquant, une boîte de dialogue de configuration est affichée.

3 Installer QDevelop

QDevelop est disponible en téléchargement sur le site <http://qdevelop.org/> sous plusieurs formats:

3.1 Depuis les binaires

Ces paquets contiennent la dernière version stable compilée et sont disponibles pour plusieurs environnements: Ubuntu, Mandriva, Fedora Core etc. Une version setup pour Windows est disponible ainsi qu'une version pour Linux indépendante de la distribution permettant d'utiliser l'application si aucun paquet n'est disponible pour votre environnement favori.


3.2 Depuis les sources

Il est également possible de télécharger sur le site qdevelop.org le fichier zip contenant les sources de la version stable du projet afin de le compiler. Une autre possibilité est de récupérer la dernière version de développement accessible à l'adresse <http://code.google.com/p/qdevelop>.

QDevelop peut-être construit en ligne de commande comme toute application Qt. Déplacez-vous dans le répertoire du projet puis entrez les commandes:

```
qmake  
make
```

Après la compilation, l'exécutable est disponible dans le sous-répertoire bin/. Le programme obtenu est autonome, c'est-à-dire que tout ce qui lui est nécessaire (images, fichiers de traductions etc.) est embarqué dans l'exécutable. Vous pouvez donc le placer dans n'importe quel répertoire de votre disque où vous pourrez le lancer.

 QDevelop est un projet Qt comme les autres. A ce titre il est ensuite possible d'ouvrir le projet dans l'application elle-même afin de le compiler ou de le modifier.

4 Modèle objet de Qt

La classe `QObject` est la classe de base de tous les objets de Qt utilisant des signaux et/ou des slots. Tous les objets Qt héritant de `QObject` possèdent un paramètre dans le constructeur appelé parent. Les `QObject` s'organisent dans des arbres d'objets. Quand vous créez un `QObject` avec un autre `QObject` comme parent, l'objet parent ajoute l'autre à sa liste d'enfants. Lorsque le parent sera détruit, il supprimera automatiquement ses enfants dans son destructeur. La gestion de la mémoire est, grâce à ce mécanisme grandement facilité et les risques de fuites-mémoire sont moindres. En résumé, les seuls objets à détruire avec *delete* sont ceux ayant été créés avec *new* et ne possédant pas de paramètre parent renseigné.

5 Signaux et Slots

Le dispositif central du modèle objet de Qt est un mécanisme très puissant pour la communication d'objets faiblement couplés composé des signaux et des slots. Faiblement couplé signifie que l'émetteur d'un signal ne sait pas quel objet va le prendre en compte (il sera peut être ignoré). De la même façon, un objet interceptant un signal ne sait pas quel autre objet a émis le signal. Cette technique permet de relier entre eux des objets de types différents et offre une grande souplesse de développement. Lorsqu'un objet veut signaler que quelque chose s'est passé le concernant, un bouton qui vient d'être cliqué par exemple, il émet un signal. Un signal est également émis par un objet lorsque son état a changé. L'objet émetteur ne sait pas comment ce signal sera interprété (il ne sera peut être pas pris en compte). L'action à mener, le code à exécuter est contenu dans un slot. Les slots peuvent être employés pour recevoir des signaux, mais ce sont également des fonctions normales membres de classes. Il suffit alors de connecter (associer) le signal au slot pour que le programme exécute le code de cette méthode définie comme slot lorsque le signal sera émis. Par exemple, pour un bouton, le signal le plus utilisé est sans doute *clicked()* qui est émis lorsque l'utilisateur clique sur un bouton. Connecter un signal à un slot consiste à indiquer à Qt quel slot déclencher lorsqu'un signal est émis. Voici en exemple la définition d'une classe contenant des signaux et des slots:

```
class Personne : public QObject  
{
```

Q_OBJECT

public:

```
    Personne(QObject *parent=0, char *name=0 );
```

```
    int agePersonne() const { return age; }
```

public slots:

```
    void setAge( int );
```

signals:

```
    void ageChange( int );
```

private:

```
    int age;
```

```
};
```

Le fichier d'implémentation pourrait ressembler à ceci:

```
#include <QObject>
```

```
#include "personne.h"
```

```
Personne::Personne( QObject *parent, char *name ) : QObject(parent, name )
```

```
{
```

```
}
```

```
Personne::setAge(int a)
```

```
{
```

```
    if( a != age )
```

```
    {
```

```
        age = a;
```

```
        emit ageChange(a);
```

```
    }
```

```
}
```

Un objet ne sait pas si quelque chose reçoit ses signaux et un slot ne sait pas s'il a des signaux qui lui sont reliés. Ceci permet de créer avec Qt des composants véritablement indépendants. Vous pouvez relier autant de signaux que vous voulez à un simple slot, et un signal peut être relié à autant de slot que vous le désirez. Il est même possible de relier un signal directement à un autre signal. (Celui-ci émettra le deuxième signal immédiatement toutes les fois que le premier sera émis.)

Ensemble, les signaux et les slots composent un mécanisme de programmation composant puissant.

- **Un signal peut être connecté à plusieurs slots:**

```
connect(bouton, SIGNAL(clicked()), this, SLOT(slotMultiplie()));
```

```
connect(bouton, SIGNAL(clicked()), this, SLOT(slotFerme()));
```

Lorsque le signal est émis, les slots sont appelés les uns après les autres dans un ordre qui peut-être différent de la déclaration.

- **Plusieurs signaux peuvent être connectés à un même slot:**

```
connect(bouton, SIGNAL(clicked()), this, SLOT(slotMAJ()));
```

```
connect( MaListBox, SIGNAL(selectionChanged()), this, SLOT(slotMAJ()));
```

Lorsque l'un des signaux est émis, le slot est exécuté. Si chaque objet émet son signal, le slot est exécuté deux fois.

- **Un signal peut être connecté à un autre signal:**

```
connect(bouton, SIGNAL(clicked()), autreBouton, SIGNAL(clicked()));
```

Lorsque le premier signal est émis, le deuxième l'est également.

- **Les connexions peuvent être supprimées:**

```
disconnect(bouton, SIGNAL(clicked()), this, SIGNAL(slotClic()));
```

C'est rarement nécessaire, car Qt détruit les connexions d'un objet lorsque celui-ci est supprimé.

Comme c'est expliqué dans la partie traitant du sous-classage, QDevelop utilise l'auto-connexion des objets afin d'éviter l'habituelle connexion par la fonction *connect()*.

6 Démarrage

QDevelop peut-être lancé depuis l'interface graphique à partir de l'explorateur de fichiers ou d'un menu. Il peut également être lancé en ligne de commande et recevoir un certain nombre d'arguments:

```
QDevelop [-l traduction] [fichier(s)] [projet]
```

Lorsque le programme démarre, il détecte automatiquement la langue du système installé et charge, s'il est disponible, le fichier de traduction correspondant. Ainsi, si votre système est configuré en Portugais, l'interface apparaîtra automatiquement dans cette langue. Dans certains cas, il peut être utile de demander l'utilisation d'une autre langue en utilisant *-l traduction*. Actuellement treize langues sont disponibles dont le Français.

Toujours en paramètre, il est possible d'indiquer des fichiers sources ou un fichier projet à ouvrir.

6.1 Contrôle des outils externes

Au lancement, une des premières tâches exécutées est de vérifier la présence des outils externes qui sont nécessaires au programme. Ce contrôle, réalisé de manière silencieuse affichera un dialogue si un ou plusieurs outils sont absents.

Guide d'utilisation de QDevelop

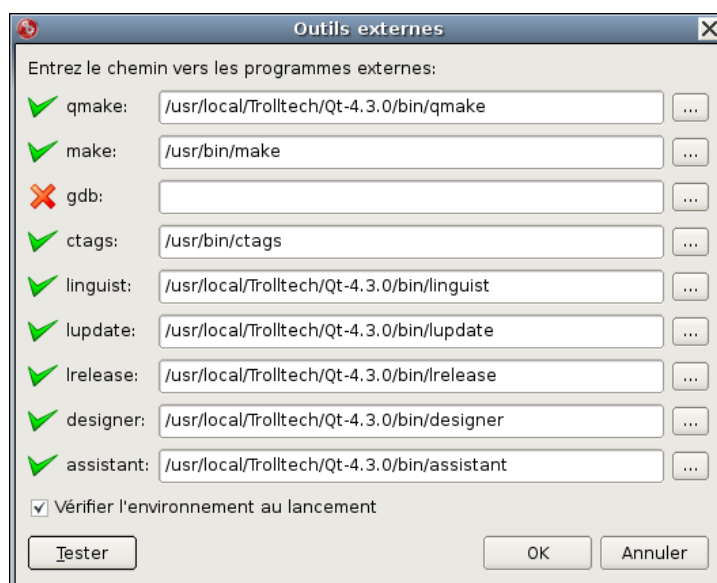


Illustration 1: Contrôle des outils externes

Une coche verte indique que le

programme est présent à l'emplacement renseigné, alors qu'une croix rouge signifie que le programme n'est pas trouvé à l'emplacement indiqué. Pour chacun des outils, il est possible en cliquant sur le bouton « ... » de rechercher le programme exécutable correspondant. Le bouton « Tester » permet de vérifier à nouveau la présence des programmes ce qui peut-être utile lorsqu'un nouveau chemin est ajouté. Lorsque toutes les lignes sont en « vert », QDevelop est correctement configuré pour construire des programmes Qt. Ce dialogue est également accessible par le menu « Options | Outils externes ».

i ctags et gdb sont optionnels et n'empêcheront pas la création de programmes. Comme indiqué plus haut, ctags est néanmoins fortement conseillé afin de bénéficier de toutes les fonctionnalités telles que la complétion de code et l'explorateur de classes.

6.2 Présentation de l'interface

L'interface utilisateur est conforme à ce que l'on peut attendre d'un IDE. Une barre de menu, une

barre d'outils

puis le reste de

la fenêtre

découpé en

trois parties.

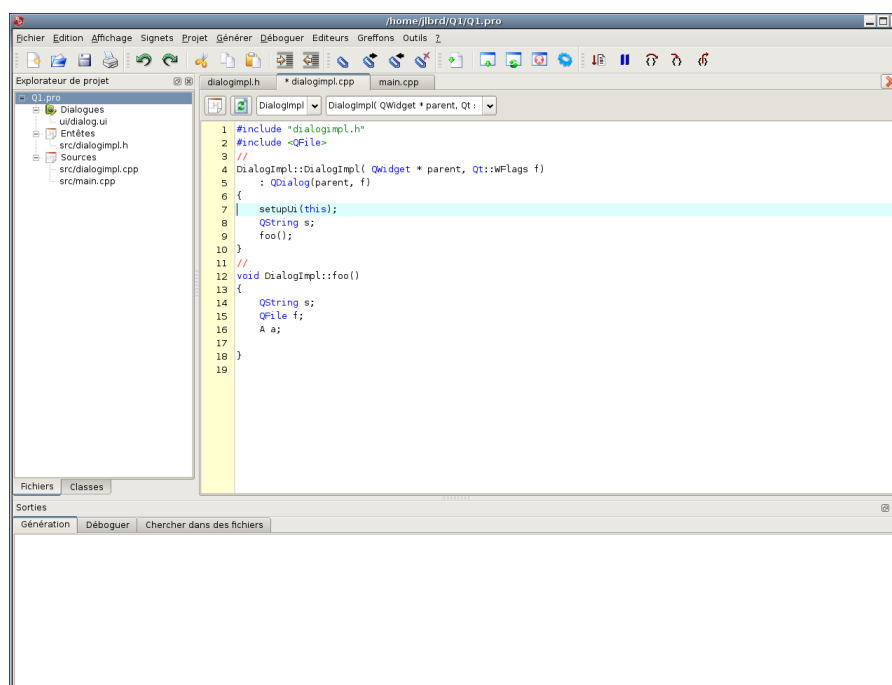


Illustration 2: L'interface de QDevelop

6.2.1 L'explorateur de fichiers.

L'explorateur de fichiers présente sous forme d'arborescence les projets ainsi que leurs fichiers. Dans le cas d'un projet unique, La première entrée contient le nom du projet puis les autres entrées sont constituées des fichiers groupés par genre: En-têtes, sources, traductions etc.

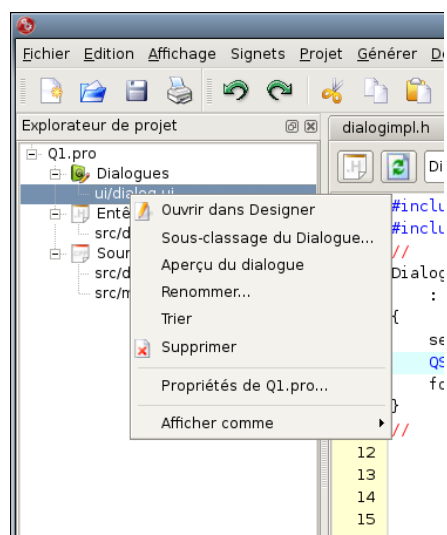


Illustration 3: Menu contextuel dans l'explorateur de fichiers

Il est également possible de créer des projets de type SUBDIRS. Il s'agit d'un projet dont le seul rôle est de contenir un ou plusieurs sous-projets et ne peut pas contenir lui-même de fichiers. La création d'un projet de type SUBDIRS est expliquée dans la partie « Création de projet ».

Il est possible de manipuler les fichiers par un clic droit affichant un menu contextuel adapté au type de fichier. Dans l'exemple ci-dessus, comme il s'agit d'un dialogue, il est possible de l'ouvrir dans Designer (l'éditeur d'interfaces utilisateur de Qt) ou encore de le sous-classer. Certaines entrées de ce menu sont communes à tous les types de fichiers comme « Renommer », « Supprimer » etc. D'autres sont disponibles dans le menu principal. Par exemple l'entrée « Propriétés du projet » est également accessible dans le menu déroulant « Projet | Propriétés... ».

6.2.2 L'explorateur de classes

Présent dans un deuxième onglet à côté de l'explorateur de projets, l'explorateur de classes affiche sous forme d'arbre les fonctions, les classes et les variables du projet. Il permet d'ouvrir rapidement le fichier d'en-tête ou d'implémentation et de se placer sur la variable, la méthode ou le nom de classe sélectionné. Un double-clic ouvre de préférence le fichier d'implémentation si l'objet y est présent et à défaut le fichier d'en-tête. Un clic droit affiche là aussi un menu contextuel dont les entrées sont adaptées au type de l'élément cliqué. Si l'élément est un nom de classe, il est possible d'ajouter une méthode ou une variable dans cette classe.

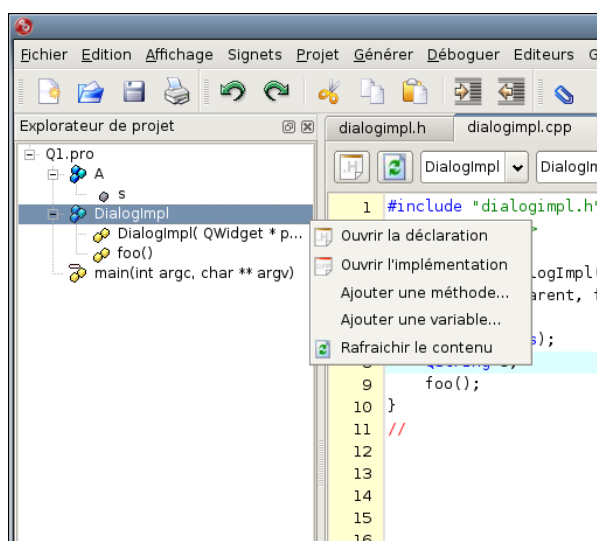



Illustration 4: L'explorateur de classes

Lorsqu'un fichier est ouvert dans l'éditeur et que son contenu est modifié, l'explorateur est mis à jour automatiquement afin d'ajouter ou supprimer les éléments qui pourraient être modifiés. Pour cela, le programme vérifie régulièrement si le contenu des fichiers ouverts a été modifié. L'intervalle de ce contrôle peut-être configuré dans le dialogue « Options » du menu « Outils ».

-  Son utilisation nécessite l'installation du programme « exuberant ctags ». Si après avoir ouvert un projet l'explorateur reste vide, c'est très certainement parce-que ctags n'est pas installé.

6.2.3 L'éditeur de fichiers sources

Les fichiers sources (.cpp) et d'en-têtes (.h) sont édités dans l'éditeur intégré. Cet éditeur fournit ce que l'on est en droit d'attendre d'un éditeur de code sources: coloration syntaxique, numérotation des lignes, indentation automatique, complétion de code etc.

```

1 #include "dialogimpl.h"
2 #include <QFile>
3 //
4 DialogImpl::DialogImpl( QWidget * parent, Qt::WFlags
5 : QDialog(parent, f)
6 {
7     setupUi(this);
8     QString s;
9     foo();
10 }
11 //
12 void DialogImpl::foo()
13 {

```

Illustration 5: L'éditeur de sources

La plupart de ces fonctionnalités sont optionnelles et peuvent être désactivées dans le dialogue « Options ». Pour ouvrir un fichier du projet, il suffit de double-cliquer sur l'élément dans l'explorateur de fichiers.

Les fichiers ouverts sont présents dans des onglets qui peuvent être déplacés par simple cliqué-déplacé et qui peuvent être fermés par un clic droit.

Au dessus du texte de l'éditeur sont présents:

- Un bouton permettant de basculer dans le fichier en-tête ou d'implémentation correspondant à celui ouvert.
- Un bouton de rafraîchissement qui met à jour les deux listes déroulantes.
- Une liste déroulante contenant le nom des classes présentes dans le fichier.
- Une liste contenant les méthodes pour la classe sélectionnée

Ces deux listes qui sont uniquement présentes pour les fichiers d'implémentation permettent de se positionner rapidement dans le corps d'une méthode à l'intérieur du fichier sources.

Une fonctionnalité intéressante est la possibilité de se déplacer sur la déclaration ou sur l'implémentation d'un membre d'une classe ou une fonction. Dans la capture d'écran ci-dessus, après avoir effectué un clic droit sur `foo()` de la ligne 9, il est possible de se déplacer à la ligne 12 si en choisissant « Aller à l'implémentation » ou se positionner dans le fichier d'en-tête en sélectionnant « Aller à la déclaration ». De la même façon, en cliquant sur le nom de méthode dans le fichier d'en-tête, il est possible de placer le curseur sur la méthode dans le fichier sources.

A gauche de l'éditeur se trouve la colonne de numérotation des lignes. En plus de sa fonction principale, cette élément permet de placer des signets par un clic sur le numéro de la ligne désiré. Il permet également de placer des points d'arrêt par un clic droit.

6.2.3.1 Complétion de code

La complétion de code affiche une liste des fonctions et les attributs disponibles pour le type de préfixe saisi. Cette liste est affichée dès que « . », « -> » ou « :: » est saisi derrière un nom. Il est également possible de demander explicitement la complétion en se plaçant à l'endroit désiré puis en l'appelant par le menu «Édition | Compléter le code » ou par le raccourci clavier qui est par défaut

« Ctrl+Espace ».

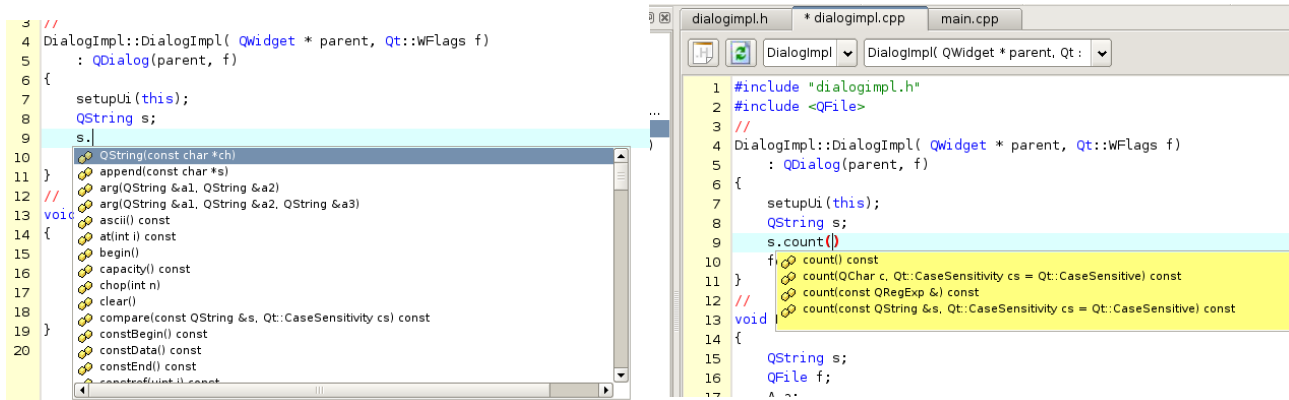



Illustration 6: La complétion de code

Ci-dessus la complétion est demandée pour la variable « s » qui est de type QString. La liste affiche donc toutes les fonctions qui appartiennent à QString. Lorsque l'utilisateur a choisi une fonction dans la liste, une bulle d'aide affiche toutes les syntaxes possibles. Dans l'exemple, les quatre façons d'employer la fonction *count()* sont affichées. Si la fonction ne peut pas recevoir de paramètre, le curseur est placé après les parenthèses et aucune bulle d'aide n'est affichée.

 Lorsque le nom de la fonction est déjà présent dans le texte, un « Ctrl+Espace » lorsque le curseur est placé entre les parenthèses permet d'afficher directement la bulle d'aide contenant l'ensemble des paramètres possibles.

6.2.4 Utiliser l'aide

La bibliothèque Qt est livrée avec Assistant, un outil permettant de consulter l'aide en ligne. Lorsque Qt Assistant est lancé, il est présenté dans une application basée sur une fenêtre principale standard, avec une barre de menu et une barre d'outils. En dessous, à gauche se trouve la fenêtre de navigation, et à droite, la fenêtre de documentation. Assistant fonctionne de la même façon qu'un navigateur Web. Si l'utilisateur clique sur un texte souligné, la fenêtre de documentation affiche la page référencée. Il est également possible de placer des signets afin de retrouver rapidement une page précédemment visitée.

Même si Assistant peut-être utilisé comme un navigateur pour parcourir la documentation Qt, il offre une puissante fonctionnalité de navigation que les navigateurs ne permettent pas.

Pour effectuer une recherche par index, cliquez sur l'onglet Index dans la barre de gauche. Entrez un mot dans le champs de saisie « Look For », par exemple « QString ». Après validation, une liste contenant tous les thèmes relatifs à ce mot est affichée. Il suffit alors de double-cliquer sur une ligne afin d'afficher la page correspondante.

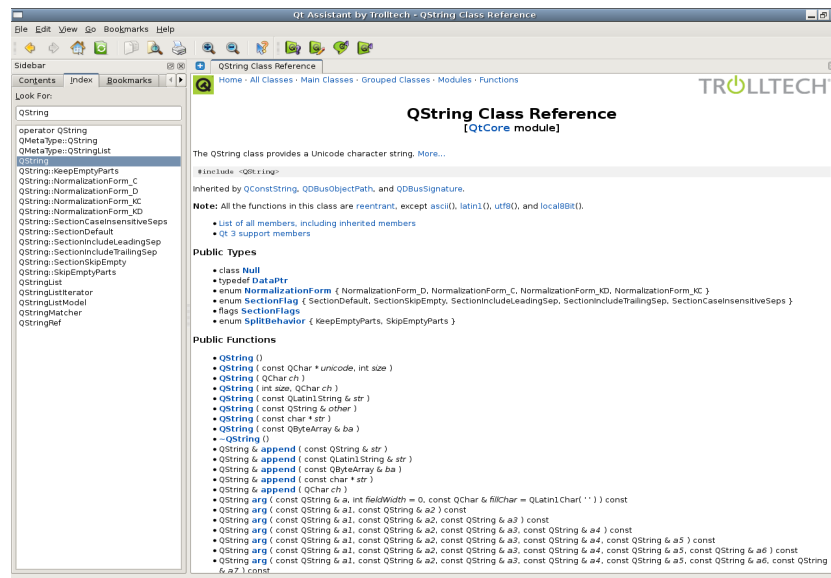


Illustration 7: L'aide dans Qt Assistant

Lorsque l'aide est appelée à partir

de QDevelop, Assistant est lancé comme programme externe puis est totalement contrôlé par l'application. En effet, si le curseur est placé sur un mot dans l'éditeur lorsque l'aide est appelée, QDevelop tente de déterminer la classe à laquelle appartient le mot puis présente dans Assistant la page correspondante. Dans le code:

```
QString s;
int i = s.count();
```

Si l'aide est appelée alors que le curseur est positionné sur *count*, l'aide affichée portera sur la fonction correspondante de *QString*.

6.2.5 Bases de données

Le fichier *qdevelop-settings.db* est une base de données SQLITE associée à chaque projet. Il est utilisé par l'IDE pour stocker des informations comme les fichiers ouverts, le contenu de l'explorateur de classes, les signets ainsi que les variables et méthodes du projet. Cette base de données permet une réouverture très rapide des projets.

Un autre fichier de base de données utilisé est *qdevelop.db* qui est créé dans le répertoire « Application Data\QDevelop » de l'utilisateur sous Windows et dans le répertoire *.qdevelop* également dans le répertoire de l'utilisateur sous les autres systèmes comme Linux. Ce fichier est utilisé pour stocker toutes les informations utiles à la complétion de code des classes Qt. Il est automatiquement renseigné lorsque ce fichier est manquant ou vide lors du lancement du programme. Il est également possible de demander sa reconstruction par le menu « Options | Reconstruire la base de données des classes Qt ». Cette option devra être utilisée lorsqu'une nouvelle version de Qt sera installée et permettra de prendre en compte les changements dans les classes Qt. Dans le dialogue de configuration du programme, il est possible d'indiquer le répertoire des fichiers d'entêtes de Qt à utiliser pour la construction de la base de données. Ce répertoire est configuré par défaut à la valeur renvoyée par la bibliothèque et conviendra dans la plupart des cas.

Ces deux fichiers peuvent sans problème être supprimés car ils sont automatiquement récréés par le programme en cas de besoin lorsque l'application est lancée et lorsqu'un projet est chargé.

7 Projets Qt

qmake fournit un système de projet orienté objet pour gérer la construction des applications, bibliothèques, et d'autres composants. Cette approche donne au développeur le contrôle sur les fichiers source utilisés, et permet de décrire toutes les étapes du processus de manière concise, généralement dans un fichier projet unique. *qmake* génère les informations de chaque fichier projet dans un Makefile qui exécute les commandes nécessaires pour compiler et lier. Les ressources utilisées par un projet sont généralement spécifiées par une série de déclarations. Cette manière simple de décrire les projets permet également de spécifier des constructions particulières pour différentes plate-formes ou environnements.

QDevelop gère les projets dans des fichiers ayant une structure strictement identique à celle utilisée par *qmake* et indépendante de la plate-forme utilisée.

7.1 Création

QDevelop possède un assistant de création de projet qui génère une application directement compilable et utilisable. Pour cela trois types de modèles d'application sont utilisés:

- **Dialogue:** Le fichier d'interface est basé sur un QDialog. Ce type de fenêtre simple sans menu ni barre d'outils est bien adapté aux « petites applications » qui se contentent d'un seul dialogue.
- **Fenêtre principale:** Si l'application est un peu plus importante, l'utilisation d'une interface basée sur QMainWindow permettra d'ajouter un menu, une barre d'outils. La fenêtre créée par l'assistant est vide et doit être enrichie dans Designer.
- **Vide:** Ce modèle ne crée aucun répertoire ou fichier et tout devra être ajouté ultérieurement. Ce type de projet doit être également choisi si des sous-projets doivent être ajoutés. Après la création du projet « maître », l'ajout de sous-projets s'effectuera par clic droit sur le nom du projet dans l'explorateur.

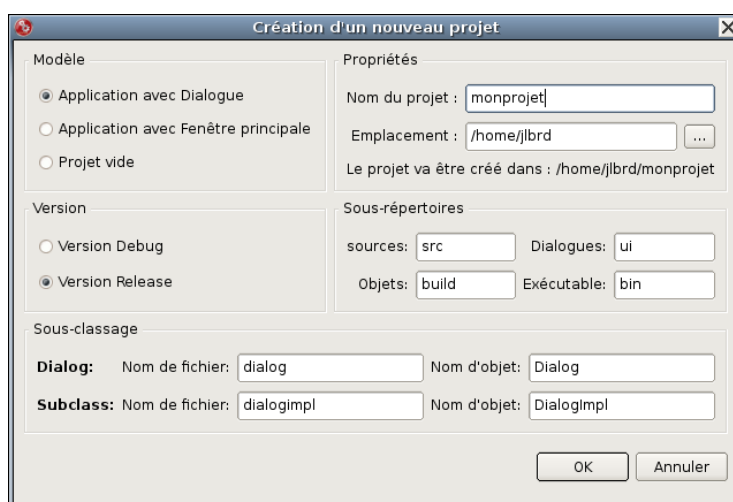


Illustration 8: Création d'un projet

Dans les deux


premiers modèles, les sous-répertoires sont prédéfinis afin de créer une arborescence de répertoires. En créant un projet simple nommé *monprojet* et en validant les options par défaut, les répertoires et les fichiers du projet sont après la construction :

```

monprojet/:
    Makefile monprojet.pro qdevelop-settings.db
monprojet/bin:
    monprojet
monprojet/build:
    dialogimpl.o main.o moc_dialogimpl.cpp moc_dialogimpl.o ui_dialog.h
monprojet/src:
    dialogimpl.cpp dialogimpl.h main.cpp
monprojet/ui:
    dialog.ui

```

On peut voir que chaque fichier est créé dans un répertoire différent adapté à son type ce qui permet d'avoir une plus grande clarté.

 Le répertoire *build*, contient des fichiers sources. Ils ne doivent pas être modifiés manuellement car ils sont générés automatiquement lors de la compilation. Les modifications que vous pourriez y faire seraient écrasées lors de la prochaine compilation.

La zone « Dialog: » permet de définir le nom du fichier d'interface, *dialog.ui* ou *mainwindow.ui* par défaut et le nom d'objet de cette fenêtre. Ce nom d'objet qui correspond à la propriété « *objectName* » dans *Designer* est utilisé pour le sous-classage du dialogue. Dans le fichier « *dialogimpl.h* » :

```
class DialogImpl : public QDialog, public Ui::Dialog
```

Dialog correspond au nom d'objet donné au dialogue. Si vous modifiez ultérieurement le nom dans *Designer*, pensez à le faire également dans la classe héritée du dialogue, « *dialogimpl.h* » par défaut. La zone « subclass » permet d'indiquer le nom du fichier et le nom de la classe qui héritent du dialogue. Cette classe générée par l'assistant possède uniquement le constructeur. La partie « Sous-classage » traite plus en détail la manière d'implémenter les dialogues.

7.2 Propriétés

Le dialogue de configuration permet de modifier les caractéristiques ainsi que les variables du projet. La plupart de ce qui peut-être trouvé dans un fichier projet est configurable dans ce dialogue. Dans cette partie, nous n'allons pas décrire l'intégralité des éléments présents dans ce dialogue. Pour plus de détail, le lecteur pourra étudier la documentation relative à *qmake* qui traite de l'ensemble des options possibles.

Ce dialogue peut être affiché dans deux modes. Le mode simple affiche peu d'options dans l'onglet « Configuration ». Ce mode convient aux applications simples sans configuration particulière. Le « Mode Avancé » affiche au contraire l'ensemble des options possibles pour un projet et permet de le configurer plus finement. Voici quelques éléments importants de ce dialogue :

- **Type de projet:** Les types disponibles sont *app*, *lib* et *subdirs*. Le type *subdirs* est un peu particulier car il désigne un projet destiné uniquement à contenir des sous-projets. Une contrainte est qu'un fichier projet de ce type ne peut contenir que la variable *SUBDIRS* qui désigne les sous-répertoires. Cette contrainte empêche de changer le type d'un projet *app* ou *lib* pour un projet *subdirs* et inversement. La seule méthode pour créer un projet *subdirs* est de créer un projet vide puis, par un clic droit sur le nom du projet dans l'explorateur, de créer un ou plusieurs sous-projets.
- **Répertoires par défaut:** Ces deux chemins ne sont pas enregistrés dans le fichier projet mais dans la base de données « *qdevelop-settings.db* » associée au projet. Ils permettent d'indiquer les chemins par défaut où doivent être créés les fichiers dialogues et les fichiers sources.
- **Onglet Configuration:** Les éléments de cet onglet permettent de renseigner dans les fichiers projets les variables *TEMPLATE*, *CONFIG* et *QT* qui suffisent à décrire toutes les options d'un projet. Le champs « Variables CONFIG supplémentaires » doit être utilisé afin d'ajouter une nouvelle valeur à la variable *CONFIG* non prévue dans les cases à cocher du dialogue.
- **Onglet Variable:** Permet d'ajouter au projet une variable prédéfinie par Qt ou de créer une variable utilisateur. Ces variables, notamment celles de Qt permettent d'indiquer à *qmake* des options particulières, comme les répertoires à utiliser pour la construction, les bibliothèques à lier à l'exécutable etc. Attention car *qmake* supporte très mal les espaces dans les noms de variables. Exemple:

```
LIBS += -IC:\mon chemin\malib.a
```

l'espace entre « mon » et « chemin » va créer une erreur lorsque *qmake* va analyser le fichier projet car il considère l'espace comme un délimiteur et pense que deux valeurs sont présentes, « *-IC:\mon* » et « *chemin\malib.a* ». Dans ce cas précis il faut entourer la chaîne de caractères par des guillemets. Un autre problème survient lorsque deux bibliothèques doivent être liées:

```
LIBS += -lmalib1 -lmalib2
```

va également provoquer un avertissement dans QDevelop causée par l'espace. La solution ici est de créer deux instances de *LIBS*, une pour chaque bibliothèque. QDevelop autorise bien sûr la création de deux variables identiques.

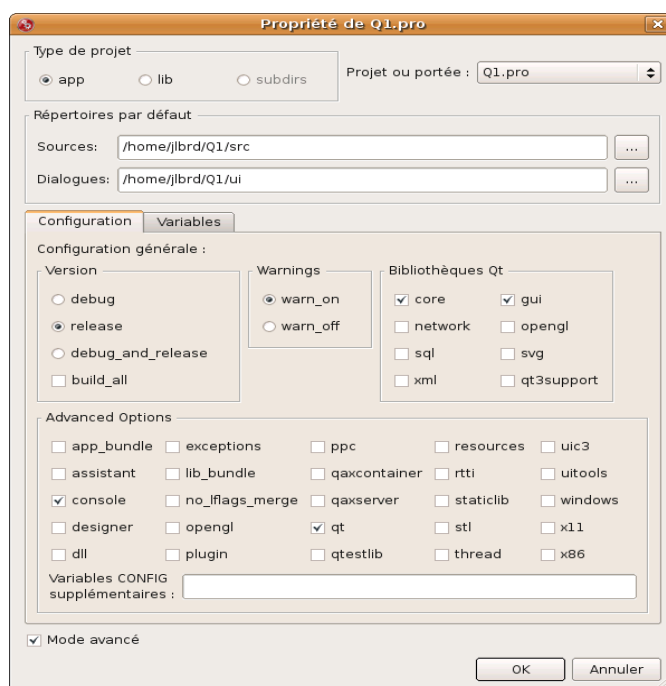


Illustration 9: Propriétés en mode avancé

L'ensemble des propriétés et des variables peuvent être modifiées soit pour le projet lui-même, c'est le cas le plus fréquent, soit pour une portée de ce projet. Dans ce cas, bien entendu un paramétrage ou une variable ne s'applique qu'à la portée.

i Peut-être que la manière dont QDevelop enregistre les fichiers projet (.pro) ne vous convient pas. Il est également possible que la complexité d'un fichier projet ne permette pas à l'application de l'enregistrer correctement. QDevelop ne sauvegarde sur disque un fichier projet que lorsqu'une modification y est apportée: ajout, suppression de fichiers ou validation du dialogue de propriétés. Si vous n'effectuez aucune de ces modifications, le fichier projet ne sera jamais enregistré.

7.3 Éditions des fichiers

Dans QDevelop, la facilité d'utilisation est le fil conducteur de toutes les évolutions du programme. C'est-à-dire que l'ajout de nouvelles fonctionnalités ne doit pas rendre son utilisation plus compliquée que précédemment. C'est pour cette raison que chaque fois que c'est possible, l'appel de programmes externes est effectuée. C'est particulièrement vrai pour les outils fournis avec Qt qui alourdiraient beaucoup l'interface s'ils étaient intégrés dans l'IDE. Comme nous l'avons déjà vu précédemment, l'édition de fichiers sources est effectuée dans l'application afin de bénéficier de

fonctionnalités avancées d'édition de code.

7.3.1 Ressources

Un fichier ressources (.qrc) est un fichier texte au format XML qui contient une liste de fichiers sur disque destinés à être embarqués dans l'exécutable. Cela permet de livrer un fichier programme unique qui peut contenir des images, des fichiers de traduction, des fichiers sons etc. Comme ces fichiers sont inclus dans le programme, le développeur n'a pas à craindre la suppression ou le déplacement de l'un de ces fichiers. Les fichiers ressources sont actuellement ouverts dans l'interface intégrée. Il est également possible de les éditer dans Designer afin d'ajouter ou de supprimer des fichiers.

7.3.2 Dialogues

Le terme « Dialogue » doit être pris au sens large car il désigne les fenêtres héritant de *QMainWindow*, de *QDialog* ou même de *QWidget*. L'édition d'un dialogue peut être demandée par un double-clic ou un clic droit sur son nom dans l'explorateur de fichiers. Qt Designer est alors lancé et ouvre le fichier dont le nom est reçu en paramètre.

Designer est lancé en externe ce qui permet de ne pas surcharger l'interface. L'inconvénient c'est qu'une nouvelle instance de Designer est lancée à chaque appel et qu'il faut éviter d'ouvrir le même dialogue plusieurs fois simultanément. A noter qu'à partir de la version 4.3 de Qt ce problème n'existe plus puisqu'une seule instance est lancée puis contrôlée par QDevelop.

Lorsqu'un dialogue est enregistré par Designer, il est automatiquement pris en compte lors de la construction suivante du programme.

Par un clic droit dans l'explorateur de fichiers, il est possible de demander un aperçu du dialogue. Cet aperçu permet d'afficher le dialogue sans sortir de QDevelop et donc sans lancer Designer. Chaque widget du dialogue affiche son nom d'objet dans une bulle d'aide lorsque la souris est positionnée dessus.

7.3.3 Traductions

L'internationalisation d'une application permet de la rendre utilisable dans toutes les langues. Il est assez simple de permettre la traduction d'une application. Chaque chaîne de caractères affichant des messages à l'utilisateur doit être insérée dans la macro *tr()*. Ainsi dans :

```
QMessageBox::warning(0, "monApp", tr("Enter a filename."), tr("Cancel"));
```

Les deux chaînes entourées par *tr()* seront pris en compte dans les fichiers de traductions.



Si un paramètre doit être utilisé dans la chaîne à traduire, il est mieux d'utiliser *arg()*:

```
QMessageBox::warning(0, "monApp", tr("Unable to copy %1").arg(filename),  
tr("Cancel"));
```

En effet si on utilisait *tr("Unable to copy")+filename*, on forcerait le positionnement de *filename* à

droite de la chaîne ce qui pourrait poser des problèmes de traduction dans certaines langues. Alors qu'en utilisant `%l` le traducteur, dans Qt Linguist pourra le positionner à l'endroit adéquat.

Qt manipule deux types de fichiers pour les traductions:

- Fichier (.ts) qui est un fichier XML contenant les chaînes à traduire ainsi que les chaînes traduites grâce à l'outil *Qt Linguist*.
- Fichier (.qm) au format binaire et qui contient les traductions dans un format très compact. Ce sont ces fichiers qui doivent être distribués avec les programmes.

Un fichier de traduction peut être ajouté au projet par le menu « Projet | Ajouter un nouvel élément ». Par exemple *French.ts*. Si vous utilisez un fichier ressources (.qrc), ce fichier de traduction (.qm) doit y être ajouté de cette manière:

```
<qresource prefix="/translations" >  
<file>translations/French.qm</file>  
</qresource>
```

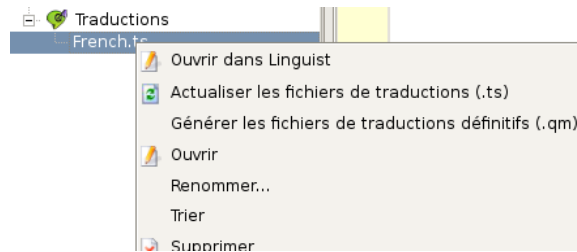


Illustration 10: Menu contextuel sur un fichier de traduction

Les actions sur les fichiers de traductions sont disponibles par un clic droit sur un des fichiers de traduction dans l'explorateur de fichiers.

- « Ouvrir dans Linguist » appelle l'outil de traduction de Qt qui ouvre le fichier (.ts) passé en paramètre.
- « Actualiser les traductions » met à jour l'ensemble des fichiers de traduction en analysant tous les fichiers sources et les dialogues. *lupdate* ajoute les nouvelles chaînes trouvées et supprime des fichiers de traductions les chaînes devenues obsolètes dans les sources.
- « Générer les fichiers définitifs » doit être appelé afin de mettre en correspondance les fichiers « release » destinés à être distribués avec les fichiers de travail (.ts). Dans ce cas c'est *lrelease* qui est appelé par QDevelop.



Si c'est le fichier (.ts) qui est présent dans le fichier projet, c'est bien le fichier définitif (.qm) qui doit être utilisé dans le programme exécutable et qui figure donc dans le fichier ressources.

Une méthode utilisée dans les sources de QDevelop lui-même permet d'automatiser le chargement de l'interface dans la langue trouvée sur la machine. Dans la fonction *main*, *Qlocale::system().language()* est utilisé afin de déterminer la langue système installée. Le format renvoyé correspond à *French*, *German*, *Dutch* etc. Les fichiers de traductions sont donc nommés en

utilisant ce nom en préfixe.

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QString locale = QLocale::system().name();
    translationFile = "./translations/" + locale + ".qm";
    QTranslator translator;
    translator.load(translationFile);
    app.installTranslator(&translator);
    ...
}
```

Dans l'exemple ci-dessus, un fichier ressources est utilisé afin de stocker le fichier de traduction. Sur un système Français, le programme chargera `./translations/French.qm` qui devra être présent dans le fichier ressource. Cette méthode qui permet de présenter à l'utilisateur une interface immédiatement dans sa langue présente l'avantage d'être simple à mettre en oeuvre.



Les fichiers dialogues (.ui) sont créés par Designer avec chaque chaîne entourée de la macro `tr()` et sont donc prêts pour la traduction.

7.3.4 Ajouter un nouvel élément

Cette partie permet d'ajouter un fichier sources ou une nouvelle ressource au projet parmi les cinq types disponibles dans le dialogue. En fonction du type de fichier, l'emplacement est prédéfini par la valeur entrée dans le dialogue de propriétés du projet.

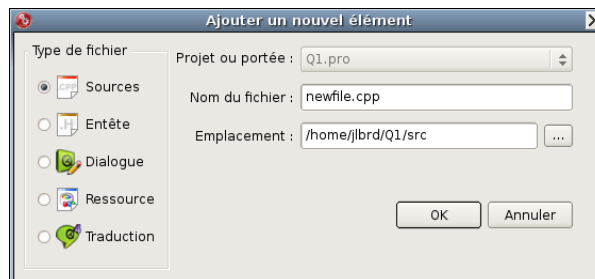


Illustration 11: Ajout d'un élément

Lorsqu'une portée est sélectionnée dans la liste, le nouveau fichier est ajouté dans cette portée uniquement. Dans le cas contraire, c'est le projet qui reçoit le fichier. Après validation, la nouvelle entrée est affichée dans l'explorateur de fichiers dans le container correspondant à son type.

7.4 Ajouter une classe

Le menu « Projet | Ajouter une nouvelle classe » accessible également par clic droit dans l'explorateur affiche le dialogue d'ajout de classe. N'importe quel type de classe, héritant ou non d'un objet Qt peut être ajouté dans le projet.

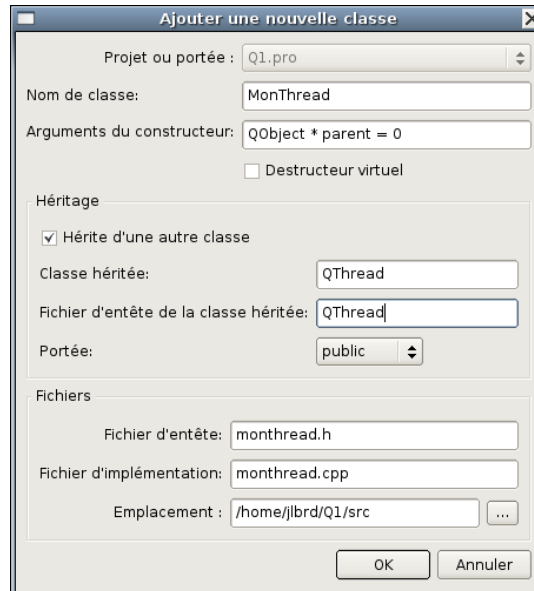


Illustration 12: Ajout d'une classe

Tous les renseignements utiles à cette création

peuvent être renseignés:

- **Nom de classe:** Il est préférable pour la clarté du programme de commencer le nom des classes par une majuscule. Il s'agit uniquement d'une convention qui n'est absolument pas obligatoire mais seulement conseillée.
- **Arguments du constructeur:** Lorsque la nouvelle classe hérite d'une classe Qt, le plus simple est de reprendre les arguments du constructeur de la classe Qt.
- **Héritage:** Ce bloc permet d'indiquer si la nouvelle classe hérite d'une classe existante. La classe héritée peut appartenir au projet ou être par exemple une classe Qt. Les classes Qt simplifient la recherche car le nom du fichier d'en-tête est souvent le même que le nom de la classe.
- **Fichiers:** C'est dans cette partie que le nom des fichiers à créer et l'emplacement sont entrés. Bien que le C++ autorise plusieurs façons de faire, QDevelop impose la création de deux fichiers séparés *.h* et *.cpp* pour y stocker l'en-tête et l'implémentation de la nouvelle classe. Pensez à ne pas saisir de majuscule dans le nom des fichiers car cette mauvaise habitude peut poser des problèmes lors de la bascule d'un projet d'un environnement à l'autre. En effet, la compilation sous Windows fonctionnera avec un « `#include "monfichier.h"` » même si le fichier s'appelle réellement « MonFichier.h ». En revanche, sous Linux, une erreur de compilation se produirait à cause du fichier non trouvé.

L'entête de la nouvelle classe est similaire à ceci:

```
#ifndef MONTHREAD_H
```

```
#define MONTHREAD_H
//
#include <QThread>
//
class MonThread : public QThread
{
    Q_OBJECT
public:
    MonThread(QObject *parent = 0);
};
#endif
```

Lorsque la nouvelle classe hérite d'une classe Qt, *Q_OBJECT* est inséré dans le corps de la classe. Cette macro indique à *qmake* que le fichier doit recevoir les traitements nécessaires aux classes Qt lors de la construction. En pratique *moc* qui va être appelé va créer un fichier *moc_monthread.cpp* (par défaut dans le sous-répertoire *build*) qui contient le code nécessaire pour implémenter les propriétés de méta-objet, c'est-à-dire la possibilité d'utiliser, entre autres, les signaux et les slots.

7.5 Ajouter une méthode à une classe

Un clic droit sur le nom de classe permet d'ajouter une méthode à cette classe. Une méthode de classe est une fonction dont la visibilité est déterminée par la liste déroulante « Portée ».

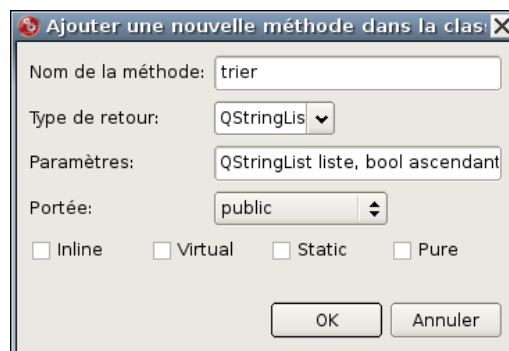


Illustration 13: Ajout d'une méthode

En plus des portées classiques du C++: *public*, *private* et *protected*, la méthode peut être déclarée comme un slot avec l'une des trois visibilités citées précédemment. Un slot est en fait une méthode classique qu'il est possible d'appeler normalement dans le programme. C'est d'ailleurs pour cette raison qu'elle peut recevoir la même visibilité qu'une méthode « normale » C++. Il est également possible de connecter ce slot à un signal. La plupart du temps un slot est connecté à un signal prédéfini d'un widget, par exemple le signal *clicked()* d'un *QPushButton* ou encore *textChanged()* d'un *QLineEdit*. Mais il est également possible de créer ses propres signaux. Dans ce cas, la classe contenant le signal doit hériter de *QObject*. Les classes Qt hérite généralement de *QObject*, en sous-classant une classe comme par exemple *QThread*, il est possible de créer et d'émettre des signaux personnalisés.

7.6 Ajouter une variable à une classe

Un clic droit sur le nom de classe permet d'ajouter une variable à cette classe.

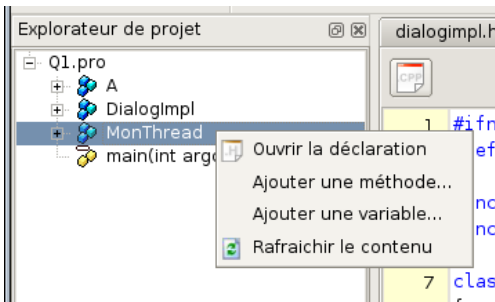


Illustration 14: Menu contextuel dans l'explorateur de classes

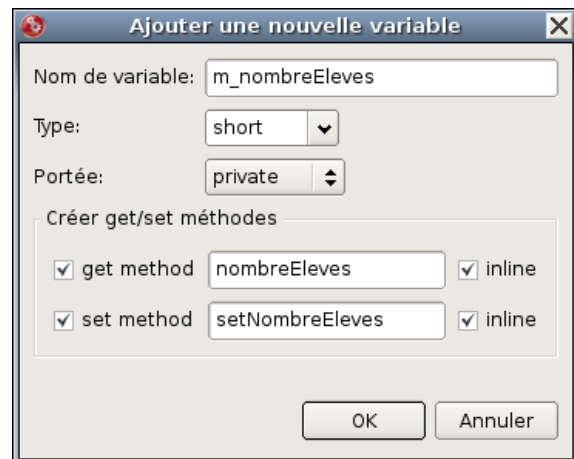


Illustration 15: Ajout d'une variable

Le dialogue permet de saisir tous les éléments requis à cette création:

```
class MonThread : public QThread
{
    Q_OBJECT
private:
    short m_nombreElevés;
public:
    void setNombreElevés(short value) { m_nombreElevés = value; }
    short nombreElevés() { return m_nombreElevés; }
    QStringList trier(QStringList liste, bool ascendant);
    MonThread(QObject * parent = 0);
};
#endif
```

Une bonne pratique du C++ est de déclarer une variable membre d'une classe comme privée et d'ajouter comme publique les méthodes nécessaires à sa lecture et son écriture. Si les deux fonctions sont déclarées *inline*, le code est inséré dans le fichier d'en-tête (.h). Dans le cas contraire il est ajouté dans le fichier d'implémentation (.cpp).

7.7 Ajouter une portée

Les portées (scope en Anglais) sont similaires à une condition *if* dans les langages de programmation procéduraux. Si une certaine condition est vraie, les déclarations à l'intérieur de la portée sont pris en compte. Une portée est particulièrement utile pour indiquer des fichiers ou des variables à intégrer dans une plate-forme particulière.

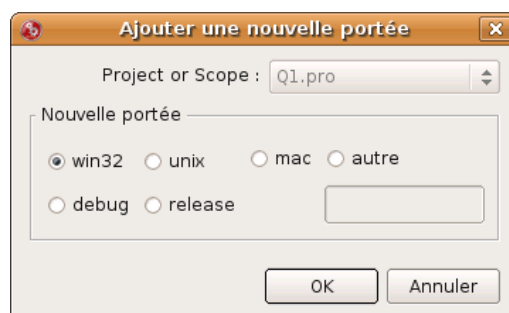


Illustration 16: Ajout d'une portée

Par exemple un fichier source à compiler uniquement sous Windows sera placé dans une portée *win32*. Une portée peut-être ajoutée à partir du menu « Projet » ou en effectuant un clic droit dans l'explorateur de fichiers sur le nom du projet. Cinq portées prédéfinies adaptées aux systèmes et configurations les plus courants sont disponibles mais il est possible de créer des portées ayant n'importe quel nom. Les portées prédéfinies sont vérifiées automatiquement par *qmake*. Il suffit de se trouver sous Linux pour que la portée *unix* soit vrai et que par exemple, les fichiers sources s'y trouvant soient compilés. Pour qu'une portée « autre » soit traitée, il est nécessaire d'ajouter son nom à la variable *CONFIG*. Le dialogue « Propriété du projet » permet d'ajouter ce genre de valeurs à la variable *CONFIG*.

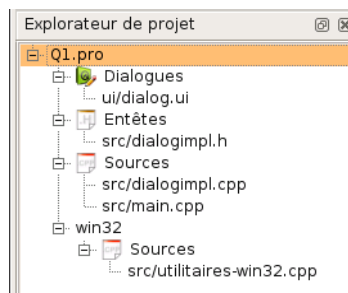
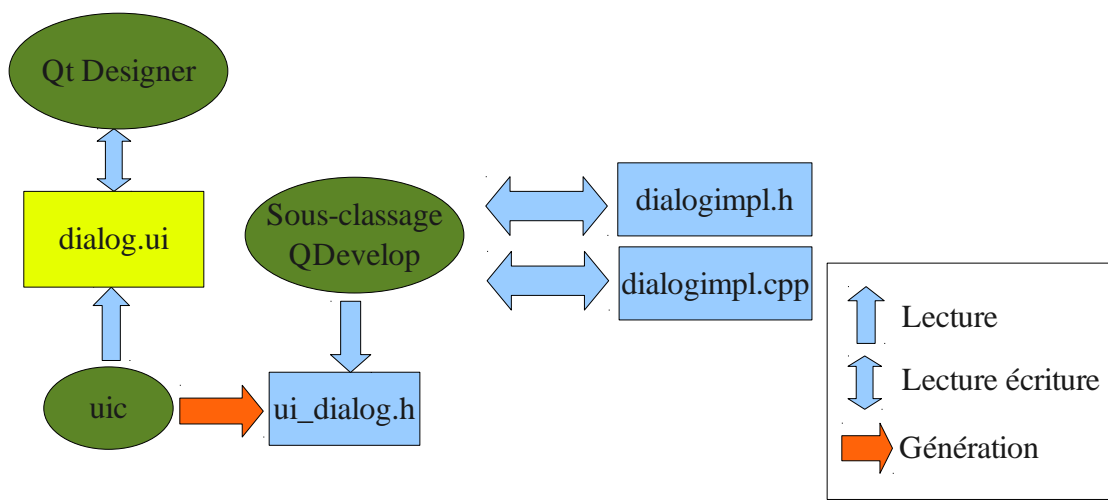


Illustration 17: Portée

Lorsqu'une portée est créée, son nom est ajouté dans l'explorateur de fichiers. Il est alors possible de lui ajouter des fichiers qui ne seront pris en compte que si la portée est vérifiée.

7.8 Sous-classage

Quant on crée des dialogues dans Qt Designer, on peut se demander comment faire exécuter au programme du code en réponse aux actions de l'utilisateur. Par exemple effectuer un traitement lorsqu'il clique sur un bouton. Un fichier dialogue (.ui) contient du XML qui définit son contenu: les widgets qu'on peut y trouver, leur nom et leur disposition les uns par rapport aux autres. Ce XML n'est pas compréhensible par un compilateur et nécessite d'être converti en fichier d'en-tête C++ pour être compilé avec le projet. Cette transformation est réalisée automatiquement pour les dialogues du projet lors de la compilation. Sachant qu'il n'est pas possible d'entrer du code dans Designer et qu'à chaque fois que le dialogue y est modifié, *uic* écrase le fichier d'en-tête, il est nécessaire de sous-classer les dialogues. C'est-à-dire créer une classe dérivée du dialogue qui va contenir le code à exécuter. Une fonctionnalité intéressante de QDevelop est de fournir une assistance à l'implémentation des dialogues.



En programmation Qt, tout objet, graphique ou non, est capable de communiquer avec un autre. Par exemple, si l'utilisateur clique sur le bouton « Fermer », nous voudrions probablement appeler la fonction `close()` de la fenêtre. Pour cela, Qt utilise les signaux et les slots. Un signal est émit lorsqu'un évènement particulier se produit. Les widgets Qt possèdent beaucoup de signaux prédéfinis, mais il est possible de sous-classer les widgets pour leur ajouter nos propres signaux. Un slot est une fonction qui est appelée en réponse à un signal particulier. Les widgets Qt possèdent des slots prédéfinis mais le sous-classage permet de créer nos slots à connecter aux signaux. Qt propose de connecter les signaux et les slots de plusieurs manières décrites précédemment. La méthode utilisée par QDevelop est l'auto-connexion des objets. Elle consiste à créer un *slot* dont le nom est composé de la manière suivante:

« on_ » + nomWidget + « _ » + nomSignal()

Par exemple `void on_okButton_clicked();` déclare un slot qui va s'exécuter lorsque le signal `clicked()` sera déclenché pour l'objet `okButton`. La manière dont est nommé le slot permet à Qt de faire l'association et d'auto-connector l'objet et son signal. Le sous-classage va créer ou modifier des fichiers d'en-tête et d'implémentation à l'intérieur desquels va être inséré le slot privé comme sur les captures ci-dessous.

```

10 //
11 class DialogImpl : public QDialog, public Ui::Dialog
12 {
13     Q_OBJECT
14 public:
15     DialogImpl( QWidget * parent = 0, Qt::WFlags f = 0 );
16     void foo();
17 private slots:
18     void on_okButton_clicked();
19 };
20 #endif
21
22
23

```

Illustration 18: dialogimpl.h

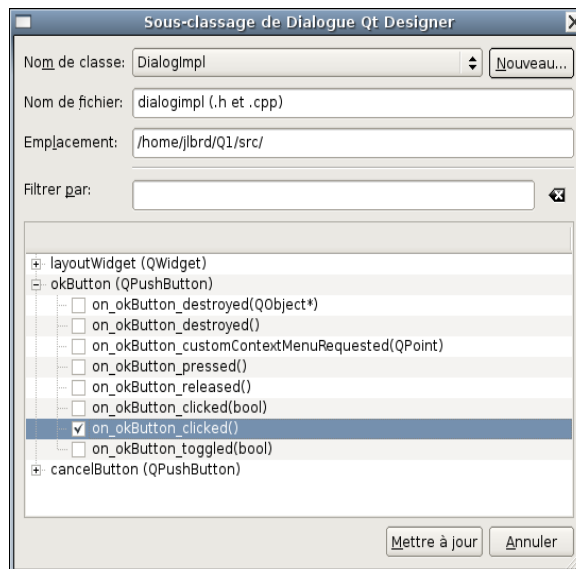
```

11 //
12 void DialogImpl::foo()
13 {
14     QString s;
15     QFile f;
16     A a;
17 }
18
19
20 void DialogImpl::on_okButton_clicked()
21 {
22     // TODO
23 }
24

```

Illustration 19: dialogimpl.cpp

Un clic-droit sur un fichier dialogue dans l'explorateur de fichiers puis le choix « Sous-Classage du Dialogue » affiche la fenêtre de sous-classage. Cette dernière liste tous les widgets présents dans le dialogue et permet, pour chacun d'entre eux de choisir les signaux à implémenter.



Le sous-classe est réalisé *Illustration 20: Sous-classe*
dans un fichier d'en-tête

(*dialogimpl.h* dans l'exemple) et un fichier d'implémentation (*dialogimpl.cpp*). Ces deux fichiers sont créés lors du premier sous-classe puis ensuite mis à jour. Le nom des fichiers utilise par défaut le nom du dialogue à la fin duquel est ajouté « impl ». Le nom de la classe dérivée du dialogue utilise la même convention. C'est-à-dire que le nom d'objet du dialogue (défini dans Designer) est utilisé en ajoutant « Impl ». Bien entendu chacun est libre de nommer les fichiers et la classe comme il le désire mais cette convention de nommage est plus claire.

Lorsque le slot est créé dans ces nouveaux fichiers, il suffit alors de saisir le code à exécuter lorsque le signal de l'objet sera déclenché, soit par l'utilisateur, soit par le programme lui-même. Ce mécanisme d'auto-connexion permet de s'affranchir des connexions habituellement réalisées par:

```
connect(okButton, SIGNAL(clicked()), this, SLOT(monSlot()));
```

Ce module est capable de créer les slots mais pas de les supprimer. Dans ce cas la suppression doit être effectuée manuellement dans les deux fichiers en effaçant le slot.

7.9 Génération

La génération consiste à construire une application exécutable, une bibliothèque ou un plugin à partir des fichiers sources d'un projet. On parle de génération car la compilation consiste uniquement en la production d'un fichier objet (.o) à partir d'un fichier sources (.cpp). La génération compile les fichiers sources, assemble les fichiers objets obtenus puis lie, si nécessaire les en-tête permettant de rendre le programme exécutable. QDevelop s'appuie sur Qt en appelant *qmake* chargé de lire le fichier projet avec lequel il produit le fichier *Makefile*. Il utilise également *make* qui se charge, en lisant le fichier *Makefile*, d'effectuer tout le processus de compilation et de liage.

Lorsque la génération est demandée, si le projet a été modifié, et seulement dans ce cas, il est sauvegardé sur disque et écrase l'ancien contenu du fichier projet (.pro). Ensuite, si nécessaire seulement (mais toujours lors de la première génération), *qmake* est appelé pour créer le *Makefile*. Puis c'est *make* qui est lancé. *Make* va compiler les uns après les autres tous les fichiers du projet et afficher, lorsqu'il y en a, les erreurs et les avertissements de compilation. Les messages de compilation produits par *make* sont affichés dans la console de sortie de l'application.



Il est possible dans le dialogue « Options » de spécifier des paramètres à passer au programme *make* lors de la construction des programmes. Ces options peuvent être par exemple « -j » afin d'effectuer plusieurs compilations simultanées, utile dans le cas d'un processeur multi-coeur. Egalement « -s » qui demande à *make* de n'afficher que les erreurs et le avertissements de compilation.

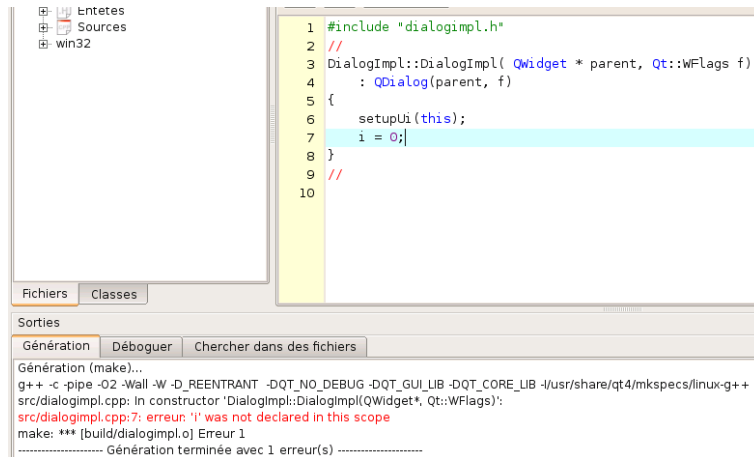


Illustration 21: La console de génération

Les erreurs sont affichées en rouge alors que les avertissements sont en bleu. Un double-clic sur les lignes dans la console de sortie permet de se positionner dans le fichier à l'emplacement de l'erreur ou de l'avertissement.



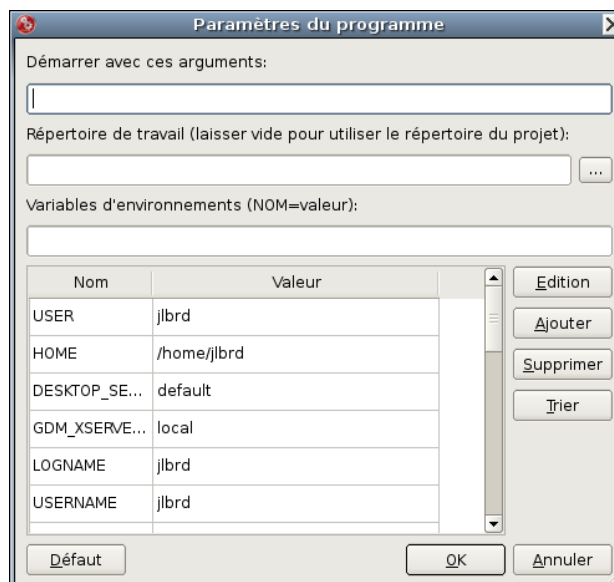
Lorsqu'un projet de type SUBDIRS est ouvert dans l'explorateur, le même processus de construction est appelé sur chacun des projets.

7.10 Exécution

L'exécution d'un programme permet de le tester en mode « Release ». C'est-à-dire que les symboles de débogage n'ont pas été inclus lors de la compilation. QDevelop intercepte les messages que peuvent produire le programme par des fonctions similaires à *qDebug()* et le affiche dans la console de sortie. Il est également possible de stopper prématurément l'exécution par le menu « Interrompre ». Lorsque plusieurs projets sont présents, l'IDE demandera quel programme l'utilisateur désire exécuter.

7.11 Paramètres du programme

Il peut être utile de transmettre au programme exécuté ou débogué des paramètres qui seront analysés par l'application lancée.



Ce dialogue permet de définir des arguments à transmettre à l'application, de changer le répertoire de travail et même de modifier les variables d'environnement qui seront transmises.

7.12 Débogage

Le débogage d'un programme est une aide à la traque des erreurs de conception. Il permet d'exécuter pas à pas le programme afin d'en vérifier tous les mécanismes. Il donne également le possibilité de visualiser le contenu des variables. QDevelop pour le débogage s'interface avec *gdb* qui doit être présent dans l'environnement de développement (voir la partie « Prérequis »). Il est également nécessaire que Qt soit construit avec les symboles de débogage. A noter que la version Windows qui intègre MinGW est livrée en mode Release uniquement. Pour construire Qt en version Debug, reportez vous à la partie « Prérequis ».

La configuration d'un projet en mode Debug doit être réalisée dans la fenêtre de configuration du projet. De plus, pour déboguer, des points d'arrêt doivent être placés dans le corps des fonctions du projet. Dans le cas contraire, le programme lancé en mode débogage ne serait jamais stoppé et ne permettrait pas d'être examiner. Un point d'arrêt peut être placé par un clic droit dans la colonne de numérotation ou par le menu « Éditeurs | Basculer le point d'arrêt ». Lorsque l'exécution du programme rencontre un point d'arrêt, il est interrompu et une flèche bleu indique la position exact du traitement.

Il est alors possible:

- De continuer l'exécution en « Pas à pas principal » qui avance le programme d'un pas sans rentrer dans le corps des fonctions
- De poursuivre en « Pas à pas détaillé » qui permet d'entrer dans le corps des fonctions
- De « Reprendre » l'exécution jusqu'au prochain point d'arrêt ou jusqu'à la fin du programme
- D'afficher la pile d'appel du programme disponible dans le menu « Déboguer | Pile d'appels »

Dans la console de débogage sont présents trois onglets:

- « Sorties » dans lequel sont affichés les messages du programme. Par exemple ceux écrits par la fonction *qDebug()*. On y trouve également certains messages du débogueur *gdb*. A noter que les messages du programme y sont également affichés en mode Release mais que

la version Windows nécessite d'activer « console » dans le dialogue de configuration du projet.

- « Variables locales » qui affiche toutes les variables locales d'une fonction ainsi que leurs valeurs. Le contenu est actualisé à chaque fois que le programme est interrompu. Il l'est également après un pas à pas détaillé ou principal.
- « Autres variables » permet d'afficher par exemple le contenu de variables globales au projet.



Un avertissement sera affiché si vous tentez de déboguer un programme construit en mode Release ou de lancer en mode « normal » un programme fabriqué en mode Debug. A l'issue de ce message, le programme sera lancé par l'IDE dans le bon mode.

Name	Type	Address	Value
app	QApplication *	0x22ff50	{<QCoreApplication> = (<QObject> = (_vptr\$QObject = 0xc2fac, static stabiM...
win	MainWindowImpl	(MainWindowImpl *) 0x22ff20	{<MainWindow> = (<QWidget> = (<QObject> = (_vptr\$QObject = 0x40f788, ...
s	QString	(QString *) 0x22ff10	my string content
i	int	(int *) 0x22ff0c	12
foo	float	(float *) 0x22ff08	4.23000002
ptr	QString *	(QString *) 0x22ff10	my string content

Illustration 22: Affichage des variables locales en débogage



Une particularité de gdb sous Windows ne permet pas la prise en compte des points d'arrêt dans le constructeur des classes alors que cela fonctionne très bien sous Linux. Une astuce consiste à placer un point d'arrêt dans le source à l'emplacement où est instancié l'objet.

```

1 #include <QApplication>
2 #include "dialogimpl.h"
3 //
4 int main(int argc, char ** argv)
5 {
6     QApplication app( argc, argv );
7     DialogImpl win;
8     win.show();
9     app.connect( &app, SIGNAL( lastWindowClosed() ), &app,
10    return app.exec();

```

Illustration 23: Point d'arrêt sur une instantiation

Lorsque le programme stoppe sur le point d'arrêt, un « Pas à pas détaillé » permet de « rentrer » dans le corps du constructeur.