Le C++0x dans Qt

Qt by Nokia

Par Olivier Goffart - Traducteur : Guillaume Belz - Qt Labs

Date de publication : 27 mai 2011

Dernière mise à jour : 5 juillet 2011

Alors que beaucoup sont enthousiastes au sujet des technologies **QML** et JavaScript, quelques-uns d'entre nous continuent de coder en C++. Le C++ est sur le point d'être mis à jour : le C++11 (anciennement connu sous le nom C++0x). Le projet final a été approuvé en mars dernier par le comité de normalisation du C++ (lire **l'annonce** sur Developpez) et la spécification finale devrait être publiée cet été. Si vous ne le savez pas encore, je vous invite à lire les pages spécialisées, telles que **Wikipédia** ou la **FAQ C++0x**.

N'hésitez pas à commenter cet article !
Commentez

I - L'article original

Le site **Qt Labs** permet aux développeurs de Qt de présenter les projets, plus ou moins avancés, sur lesquels ils travaillent.

Cet article est la traduction de l'article **C++0x in Qt** d'**Olivier Goffart** paru dans Qt Labs.

Cet article est une traduction d'un des tutoriels écrits par **Nokia Corporation and/or its subsidiary(-ies)** inclus dans la documentation de Qt, en anglais. Les éventuels problèmes résultant d'une mauvaise traduction ne sont pas imputables à Nokia.

II - Introduction

L'un des objectifs de C++0x est de rendre le langage plus accessible, ce qui est relativement possible, puisque cette nouvelle norme ne fait qu'ajouter plus de choses à apprendre, mais avec l'espoir que la partie « la plus utilisée » du C++ soit plus facile et plus intuitive.

La bonne nouvelle est que vous n'avez pas besoin d'attendre pour l'utiliser. Vous pouvez commencer dès aujourd'hui. Le compilateur que vous utilisez supporte probablement déjà une partie de C++0x : **GCC** ou **MSVC 2010**.

Bien que vous puissiez déjà utiliser C++0x avec les versions de Qt plus anciennes telles que Qt 4.7, Qt 4.8 viendra avec plus de support pour certaines nouvelles fonctionnalités du C++0x.

III - Les nouvelles macros

Certaines macros sont définies si le compilateur supporte les nouvelles fonctionnalités :

```
Q_COMPILER_RVALUE_REFS
Q_COMPILER_DECLTYPE
Q_COMPILER_VARIADIC_TEMPLATES
Q_COMPILER_AUTO_TYPE
Q_COMPILER_EXTERN_TEMPLATES
Q_COMPILER_DEFAULT_DELETE_MEMBERS
Q_COMPILER_CLASS_ENUM
Q_COMPILER_INITIALIZER_LISTS
Q_COMPILER_LAMBDA
Q_COMPILER_UNICODE_STRINGS
```

IV - Les listes d'initialisation

Qt 4.8 ajoute de nouveaux constructeurs à **QVector**, **QList** et **QStringList**, qui vous permettent de les initialiser en utilisant des crochets d'initialisation. Vous pouvez maintenant faire :

```
QVector<int> testsData { 1, 2, 10, 42, 50, 123  };
QStringList options = { QLatin1String("foo"), QLatin1String("bar")  };
```

qui initialise les conteneurs avec ces éléments.

V - Les références rvalue et la sémantique move

La plupart des classes Qt sont **implicitement partagées**, ce qui signifie qu'il est efficace de les copier si vous ne les modifiez pas (Copy-On-Write). Ce n'est pas le cas pour les **std::vector**, où chaque copie implique de copier toutes les données.

Donc, si vous avez un code comme ça :

```
std::vector<int> m_foo;
...
m_foo = getVector();
```

le getVector peut construire un nouveau std::vector et le retourner comme un objet temporaire. Ensuite, l'opérateur std::vector::operator= va supprimer l'ancien contenu de m_foo puis copier toutes les données à partir du vecteur temporaire dans m_foo. À la fin de l'instruction, le vecteur temporaire sera détruit et son destructeur va supprimer ses données. Ce serait plus efficace si, à la place, l'opérateur = permettait d'échanger des données de m_foo avec les données du vecteur temporaire. De cette façon, les anciennes données de m_foo seraient supprimées lorsque le vecteur temporaire est détruit et nous n'aurions pas besoin de créer une copie inutile. C'est ce qu'apporte la sémantique *move* dans C++11 et c'est obtenu grâce à des références rvalue.

Même si la copie d'une classe implicitement partagée est peu coûteuse, ça n'est pas totalement gratuit, nous devons encore augmenter et diminuer le compteur de références et les appels à l'opérateur = ne peuvent pas être inlinés, car cela accède aux données privées (nous ne pouvons pas inliner cela pour faciliter la compatibilité binaire). Alors maintenant, regardons l'opérateur = de **QImage** utilisant la sémantique *move* dans Qt 4.8 :

```
#ifdef Q_COMPILER_RVALUE_REFS
    inline QImage &operator=(QImage &&other)
    { qSwap(d, other.d); return *this; }
#endif
```

Nous venons d'échanger les données internes des deux images, c'est une opération très bon marché par rapport à l'opération normale qui nécessite un appel de fonction. Nous avons ajouté cette fonctionnalité à la plupart des classes implicitement partagées dans Qt. Comme l'opérateur = de nos conteneurs est beaucoup utilisé avec des temporaires, ça peut apporter de petites améliorations de performances de Qt, ce qui pourrait être une raison pour compiler Qt avec le support du C++0x (voir ci-dessous).

VI - Les boucles for basées sur un range

Qt avait déjà un **foreach** très pratique, que vous pouvez aussi trouver dans d'autres bibliothèques, comme **Boost**. Le foreach de Qt fonctionne avec une macro compliquée, mais le C++0x va plus loin et intègre cette fonction dans le langage. Donc, au lieu d'écrire :

```
foreach(const QString &option, optionList) { ... }
```

vous pouvez écrire :

```
for (const QString &option : optionList) { ... }
```

Il y a une légère différence entre les deux : Qt fait une copie du conteneur avant de réaliser l'itération. C'est peu coûteux pour les conteneurs de Qt, puisqu'ils utilisent le partage implicite, mais elle l'est plus avec des conteneurs standards, qui réalisent une copie complète de tout le contenu. La version C++11 basée sur le *range* ne fait pas de copie, ce qui signifie que le comportement est indéfini si vous ajoutez ou supprimez des éléments dans le conteneur que vous parcourez. La nouvelle boucle for appellera les fonctions **begin()** et **end()**, qui feront une copie du conteneur Qt s'il est partagé et non constant. Par conséquent, il est préférable de passer les conteneurs en constant.

```
template<class T> const T &const_(const T &t) { return t; }
```

```
for(auto it : const_(vector)) { ... }
```

VII - Les lambdas

Nous avons testé les lambdas avec quelques-unes des fonctions de **QtConcurrent**. Nous les avons testées avec **QtConcurrent::run** et **QtConcurrent::map**, mais cela ne fonctionne pas encore pour **QtConcurrent::mapped**, nous essaierons de résoudre ce problème à l'avenir. Ainsi, l'exemple de mise à l'échelle présenté dans la documentation de QtConcurrent::map pourrait être réécrit comme ceci :

```
QList<QImage> images = ...
QFuture<void> future = QtConcurrent::map(images, [](QImage &image) {
          image = image.scaled(100,100);
      });
```

Je travaille également sur l'utilisation des lambdas dans les connexions signaux/slots, mais ce sera pour Qt5.

VIII - Les chaînes de caractères Unicode

Nous n'avons pas encore ajouté le support des nouveaux types de chaînes. Mais cela pourrait venir plus tard.

IX - Allez-y et testez-le !

Si vous utilisez **MSVC 2010**, il n'y a rien à faire, vous pouvez déjà utiliser certaines fonctionnalités telles que les lambdas ou les références rvalue. Si vous utilisez **gcc**, vous devez passer l'option -std=c++0x. Vous pouvez faire cela dans votre fichier projet qmake comme ceci :

```
QMAKE_CXXFLAGS += -std=c++0x
```

Si vous voulez compiler Qt en utilisant C++0x, vous pouvez faire :

```
CXXFLAGS="-std=c++0x" ./configure
```

Qt compilé avec le C++0x respecte la compatibilité binaire avec le bon vieux C++. Et vous n'avez pas besoin de compiler Qt avec C++0x pour écrire votre propre application en utilisant C++0x.

X - Conclusion

Au nom de toute l'équipe Qt, j'aimerais adresser le plus grand remerciement à Nokia pour nous avoir autorisés à traduire cet article !

Un grand merci à **Thibaut Cuvelier**, à **ClaudeLELOUP**, à **jacques_jean** et à **_Max_** pour leur relecture très attentive et leurs conseils.