Les signaux et slots dans Qt 5.2

Introduction

L'idée de base de la programmation objet est de créer des objets. Quand on fait cela, c'est déjà un bon début. Cependant, ce n'est pas suffisant généralement. Il faut également que les objets communiquent entre eux.

Lorsque qu'un objet-enfant est créé dans un objet-parent, il n'y pas de problème pour que le parent puisse communiquer avec l'enfant. Comme le parent a créé l'enfant, il le connait bien. En particulier, il connait les fonctions publiques de l'objet-enfant et peut donc les appeler directement.

```
class Enfant {
public:
    void une_fonction() const {}
};

class Parent {
    Enfant mon_enfant;
public:
    void autre_fonction() const {
        mon_enfant.une_fonction(); // Ok
    }
};
```

Lorsque deux classes communiquent entre elles, on dit qu'elles sont couplées. Ici, le couplage est dit fort, puisque la classe Parent doit absolument connaitre la classe Enfant. Si ces deux classes sont déclarées dans des fichiers différentes (parent.h et enfant.h par exemple), il sera donc nécessaire d'inclure le fichie enfant.h dans le fichier parent.h.

```
// dans parent.h
#include "enfant.h"
```

Ce couplage n'est pas symétrique. En effet, la classe Enfant ne connait pas (et n'a pas besoin de connaitre) la classe Parent. En revanche, si on souhaite appeler une fonction de Parent dans Enfant, on est face à deux difficultés.

Premièrement, il faut appeler la fonction sur un objet en particulier. Comme l'objet Enfant est déjà un membre de la classe Parent, il ne serait pas possible de créer un nouvel objet Parent dans Enfant, puisque dans cette situation, l'objet Parent contiendrait un objet Enfant, qui contient à son tour un autre objet Parent, lui même contenant un objet Enfant et ainsi de suite. On se retrouve dans une boucle infinie, cela n'a pas de sens.

```
class Enfant {
    Parent mon_parent;
};
```

Cette situation est résolue en ne créant pas un objet Parent dans la classe Enfant, mais créant simplement un “lien” vers un objet Parent existant (par exemple, l'objet Parent qui a créé l'objet Enfant). Ce type de “lien” est réalisé en C++ en utilisant une référence ou un pointeur. Il sera alors possible d'appeler une fonction de l'objet Parent dans l'objet Enfant.

```
// dans enfant.h
class Enfant {
    Parent const& mon_parent; // référence
public:
    void une_fonction() const {
        mon_parent.encore_une_fonction();
    }
};

// dans parent.h
#include "enfant.h"
class Parent {
    Enfant mon_enfant;
public:
    Parent() : mon_enfant(*this) { // beurk
    }
```

```
    void autre_fonction() const {
        mon_enfant.une_fonction(); // Ok
    }

    void encore_une_fonction() const {
    }
};
```

Si vous essayez de compiler ce code, vous obtiendrez une erreur dans la classe Endant : “le type Parent n'est pas connu”. La raison est que le compilateur ne connait pas encore la classe Parent, il ne peut pas savoir à quoi correspond le terme “Parent” dans votre code. La solution est alors d'indiquer au compilateur que ce terme existe et qu'il correspond à une classe. Dans cette situation, on parle de “déclaration anticipée” de la classe Parent.

```
// dans enfant.h
class Parent; // on indique au compilateur que le terme
"Parent" existe et que c'est une classe
class Enfant {
    Parent const& mon_parent; // référence
public:
    Enfant(Parent const& p) : mon_parent(p) {
    }

    void une_fonction() const {
        mon_parent.encore_une_fonction();
    }
};

// dans parent.h
#include "enfant.h"
class Parent {
    Enfant mon_enfant;
public:
    Parent() : mon_enfant(*this) { // beurk
    }

    void autre_fonction() const {
        mon_enfant.une_fonction(); // Ok
```

```
    }

    void encore_une_fonction() const {
    }
};
```

Cependant, cela ne fonctionne toujours pas. En effet, trois lignes en dessous, on essaie d'appeler une fonction “encore_une_fonction” de la classe Parent. Or, pour le moment, le compilateur sait simplement qu'il existe une classe Parent, il ne sait pas encore ce qu'elle contient. Pour résoudre ce second problème, on va donc séparer la déclaration des classes de leur implémentation en les mettant dans des fichiers séparées (par exemple “enfant.cpp” et “parent.cpp”) :

```
// dans enfant.h
class Parent; // déclaration anticipée
class Enfant {
    Parent const& mon_parent; // référence
public:
    Enfant(Parent const& p);
    void une_fonction() const;
};

// dans enfant.cpp
#include "enfant.h"
#include "parent.h"
Enfant::Enfant(Parent const& p) : mon_parent(p) {
}
void Enfant::une_fonction() const {
    mon_parent.encore_une_fonction();
}

// dans parent.h
#include "enfant.h"
class Parent {
    Enfant mon_enfant;
public:
    Parent() : mon_enfant(*this);
    void autre_fonction() const;
    void encore_une_fonction() const;
};
```

```
// dans parent.cpp
#include "parent.h"
#include "enfant.h"
Parent::Parent() : mon_enfant(*this) { // beurk
}
void Parent::autre_fonction() const {
    mon_enfant.une_fonction(); // Ok
}
void Parent::encore_une_fonction() const {
}
```

Ce type de code est très classique en C++, on le retrouve partout. Mais cela pose un problème majeur : ce code n'est pas évolutif. Les classes et les fonctions appelées sont figées, si on souhaite appeler une autre fonction ou une autre classe, il faudra modifier le code de ces classes et ajouter toutes les déclarations de classes et de fonctions (imaginez le résultat sur un projet contenant plusieurs centaines de classes...) Il est possible d'améliorer la situation en mettant les fonctions et les objets comme paramètres des classes et fonctions (utilisation des templates, fonctions callback, etc), mais cela donne un code relativement lourd à maintenir et à faire évoluer.

Le but de la techniques des signaux et slots (qui n'est pas propre à Qt) est de créer un couplage faible entre les classes, ce qui ne nécessite plus que les classes se connaissent entre elles.

Les signaux-slots dans Qt 4

L'idée dans cette approche est de créer un “lien” particulier entre deux fonctions de deux classes indépendantes, de façon à ce que lorsque l'on appelle la fonction du premier objet, la fonction du second objet est automatiquement appelée. La première fonction est appelée “signal”, la seconde “slot”, le lien entre les deux s'appelle une connexion.

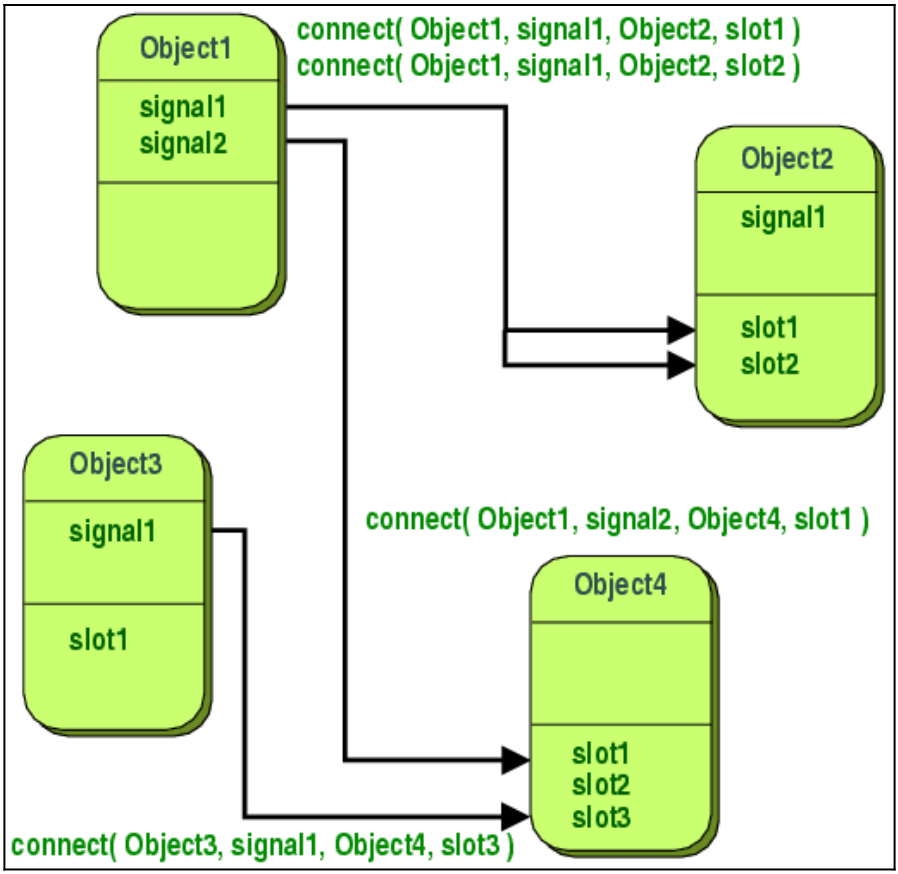
```
class A {
public:
    void fonction_de_A() const {}
};
```

```
class B {
public:
    void fonction_de_B() const {}
};

int main() {
    A un_objet_A;
    B un_objet_B;

    // création d'une connexion entre les deux fonction
    creer_connexion(un_objet_A, fonction_de_A, un_objet_B,
fonction_de_B);

    // lorsque l'on appelle la fonction de A, la fonction de
B est automatiquement appelée
    un_objet_A.fonction_de_A(); // appelle
un_objet_B.fonction_de_B()
```



Avec ce type d'approche, vous pouvez utiliser autant de classes et de fonctions que vous souhaitez, celle-ci seront indépendantes. A n'a pas besoin de connaitre B et B n'a pas besoin de connaitre A. On parle dans cette situation de “couplage faible”. Nous allons voir maintenant comme cela s'utilise dans Qt, mais sachez qu'il est également possible d'utiliser cette technique avec d'autres bibliothèques (par exemple, en utilisant Boost.Signals) ou en utilisant des fonctions comme argument (quelques mots-clés : std::function, functor object, lambda function, etc).

Le système de signaux et slots de Qt est relativement simple : lorsqu'un signal est émis avec le mot clé emit, tous les slots qui sont connectés à ce signal sont exécutés. Une connexion est créée en utilisant la fonction QObject::connect, en donnant les arguments suivants :

- un pointeur vers l'objet émetteur ;
- le nom du signal et la liste des types d'argument de la fonction signal (en utilisant la macro SIGNAL) ;
- un pointeur vers l'objet récepteur ;
- le nom du slot et la liste des types d'arguments de la fonction slot (en utilisant la macro SLOT).

Il est possible de connecter plusieurs signaux à un même slot, un signal à plusieurs slots ou un signal avec un signal. La correspondance entre les arguments des signaux et slots est vérifié comme une chaîne de caractère, lors de l'exécution. Ainsi, si vous connecter un signal(float) avec une slot(double), la connexion sera refusée puisque les arguments ne correspondent pas, malgré le fait qu'un float peut être converti en double sans problème.

```
QAction a;
QWidget w;
QObject::connect(
    &a, SIGNAL(triggered()), // connecte le signal
triggered() de QAction
    &w, SLOT(show())); // au slot show() de QWidget
// ce code permet donc d'afficher le QWidget lorsque
l'utilisateur active la QAction
```

Les classes de Qt fournissent de nombreux signaux et slots par défaut (et sont signalé comme tel dans la documentation de Qt). Vous pouvez également créer vos propres signaux et slots dans vos classes, en respectant les règles suivantes :

- la classe doit dériver de QObject (directement ou indirectement) ;
- il faut appeler la macro Q_OBJECT au début de la classe ;
- les slots sont des fonctions classiques déclarées avec le mot-clé “slots” ;
- les signaux sont des fonctions non implémentées déclarées avec le mot-clé “signals”.

Comme le principe est de pouvoir connecter n'importe quelle classe et

signaux-slots, vous pouvez sans problème connecter les classes Qt entre elles, avec vos propres classes ou vos propres classes entre elles.

Le code suivant est une exemple de propriété avec Qt. Une propriété est une variable membre d'une classe dérivée de QObject qui peut être appelée via les méta-objets et donc dans un script ou en QML. Le code suivant est relativement idiomatique, je vous conseille de respecter cette forme pour chaque propriété que vous déclarez (respectez également les noms des fonctions).

```
#include <QObject>

class Counter : public QObject // on hérite de QObject pour
bénéficier des méta-informations de Qt
{
    Q_OBJECT // cette macro permet de générer les signaux et
slots lors de la compilation
    Q_PROPERTY(int value READ value WRITE setValue
NOTIFICATION valueChanged)

public :
    int value() const {
        return m_value;
    }
public slots:
    void setValue(int value) {
        if (value != m_value) { // lorsque la valeur est
changée
            m_value = value;
            emit valueChanged(value); // on émet un signal
valueChanged
        }
    }
signals:
    void valueChanged(int value); // signal émis lorsque la
valeur est changée

private:
    int m_value;
};
```

```
Counter a, b;

// on connecte valueChanged de a à setValue de b
QObject::connect(&a, SIGNAL(valueChanged(int)), &b, SLOT(
setValue(int));

a.setValue(12);
// a émet un signal valueChanged qui active le slot setValue
de b
// a.value() == 12, b.value() == 12

b.setValue(48);
// b émet un signal valueChanged mais ce signal n'est pas
connecté à un slot
// a.value() == 12, b.value() == 48
```

Remarque importante : il faut obligatoirement déclarer les classes QObject dans un fichier .h et non dans un fichier .cpp, sinon le compilateur moc de Qt ne pourra pas fonctionner.

Créer une connexion dans Qt 5

Dans Qt 4, il est possible de connecter uniquement les fonctions déclarées comme signaux et slots dans la classes, comme indiqué dans le code d'exemple précédant. Dans Qt 5, il est maintenant possible de connecter directement des pointeurs de fonctions ou d'utiliser des fonctions lambdas.

La connexion de pointeurs de fonctions est similaire à une connexion classique, en donnant un pointeur sur les objets et sur les fonctions. Les classes émettrices et réceptrices doivent dériver de QObject mais il n'est pas nécessaire de déclarer les fonctions slots avec le mot clé « slots ».

```
class Sender : public QObject {
    Q_OBJECT

signals:
    void send(int i = 0);
};
```

```
class Receiver : public QObject {

public:
    void receive(int i = 0) { std::cout << "receive:" << i <
send(123);
];

connect(sender, &QObject::destroyed, this, &MyObject::
objectDestroyed);
```

L'avantage de cette écriture est que la compatibilité des paramètres est effectuée lors de la compilation et non lors de l'exécution. Ainsi, il est possible de connecter un signal mon_signal(float) vers un slot mon_slot(float).

Il est également possible d'utiliser une fonction lambda comme slot dans une connexion. Par exemple, lorsque l'on souhaite connecter une signal QPushButton::clicked avec un slot QLabel::setText(QString), il fallait passer par un slot intermédiaire dans Qt 4 :

```
QPushButton button;
QLabel label;
QObject::connect(&button, SIGNAL(clicked()), &label, SIGNAL(
setText(QString)); // erreur
```

Il faut donc écrire un intermédiaire :

```
class MyLabel : public QLabel { // ou une classe dérivée de
QPushButton
    Q_OBJECT
private:
    QPushButton button;
    QLabel label;
public:
    MyLabel(QObject* parent = 0) : QObject(parent) {
        connect(&button, SIGNAL(clicked()), SIGNAL(new_slot());
// Ok
    }
public slots:
    void new_slot() {
```

```
        emit label->setText("Mon texte");
    }
};
```

En utilisant les lambda, il est possible d'appeler directement :

```
QPushButton button;
QLabel label;
QObject::connect(&button, &QPushButton::clicked(), [&label]()
{ label->setText("Mon texte"); });
```

Dans ce code, on capture le pointeur vers l'objet récepteur et on récupère le paramètre passé par la fonction send() puis on appelle dans le corps de la lambda le fonction receive(). Le résultat obtenu est identique au code précédant, mais il est possible de faire beaucoup d'autres choses dans la lambda (par exemple déconnecter tous le signaux ou parcourir tous les enfants de l'objet récepteur).

Si le compilateur utilisé ne support pas les variadic template, les signaux et slots doivent avoir moins de 6 paramètres.

Dans Qt 5.2, une nouvelle syntaxe a été ajouter, pour éviter d'avoir à récupérer un objet dans la liste de capture du lambda :

```
QPushButton button;
QLabel label;
QObject::connect(&button, &QPushButton::clicked(), label, []
(){ label->setText("Mon texte"); });
```

Pense-bête

```
// Qt 4
QMetaObject::Connection QObject::connect(
    const QObject * sender, const char * signal,
    const QObject * receiver, const char * method,
    Qt::ConnectionType type = Qt::AutoConnection) [static]

QMetaObject::Connection QObject::connect(
    const QObject * sender, const char * signal,
```

```
    const char * method,
    Qt::ConnectionType type = Qt::AutoConnection) const

// Qt 4.8
QMetaObject::Connection QObject::connect(
    const QObject * sender, const QMetaMethod & signal,
    const QObject * receiver, const QMetaMethod & method,
    Qt::ConnectionType type = Qt::AutoConnection) [static]

// Qt 5.0
QMetaObject::Connection QObject::connect(
    const QObject * sender, PointerToMemberFunction signal,
    const QObject * receiver, PointerToMemberFunction method,
    Qt::ConnectionType type = Qt::AutoConnection) [static]

QMetaObject::Connection QObject::connect(
    const QObject * sender, PointerToMemberFunction signal,
    Functor functor) [static]

// Qt 5.2
QMetaObject::Connection QObject::connect(
    const QObject * sender, PointerToMemberFunction signal,
    const QObject * context, Functor functor,
    Qt::ConnectionType type = Qt::AutoConnection) [static]
```

Remarque : toutes ces fonctions sont `static`, sauf la seconde.

Conclusion

Vous pouvez télécharger un projet d'exemple montrant ces nouvelles fonctionnalités en action : la page de téléchargement.

Les images et codes d'exemple sont issus de la documentation de Qt5 disponible à cette page : Qt 5.0: Signals & Slots.

Qt