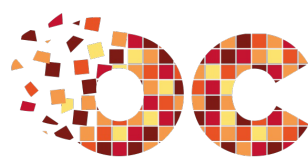


Stocker et sérialiser des objets avec Qt

Par Julien Rosset (Darkelfe)



OPENCCLASSROOMS

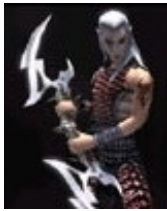
www.openclassrooms.com

Sommaire

Sommaire	2
Stocker et sérialiser des objets avec Qt	3
Pourquoi utiliser Qt ?	3
Sérialiser avec QVariant	4
QVariant et les objets sérialisables	4
Rendre un objet quelconque sérialisable	5
Le stockage avec QDataStream	6
Les opérateurs de flux vers QDataStream	6
TP-exemple : gérer un contact	7
La classe Contact	7
Rendre sérialisable notre objet	9
Définir les opérateurs de flux	9
La fonction main	12
Les fichiers finaux	14
Partager	17



Stocker et sérialiser des objets avec Qt



Par [Julien Rosset \(Darkelfe\)](#)

Mise à jour : 01/01/1970

Difficulté : Facile



Pour pouvoir comprendre parfaitement (et exploiter au maximum de ses capacités) ce tutoriel, il faut que vous connaissiez le cours de [M@teo21](#) sur le C++ jusqu'à la partie 2 (non incluse si vous connaissez déjà Qt).

Il arrive forcément, comme à tout programmeur qui se respecte, un jour où on a besoin de manipuler le contenu d'un objet, par exemple pour le placer dans un fichier (ce qui est souvent le cas, mais cela peut être pour une autre raison). En d'autres termes, vous souhaitez [sérialiser](#) votre objet.

Dans le cas d'un objet simple (avec seulement quatre entiers par exemple), on peut se débrouiller tout seul. Mais imaginez maintenant que vous ayez un objet contenant les listes des chaînes de caractères, ainsi que plein d'autres valeurs... Tout à coup, ça devient beaucoup plus compliqué.

C'est pourquoi je vous propose, par le biais de ce tutoriel, de découvrir les outils proposés par Qt, à travers la classe [QVariant](#), qui vont vous faciliter **grandement** les choses.

Sommaire du tutoriel :



- Pourquoi utiliser Qt ?
- Sérialiser avec QVariant
- Le stockage avec QDataStream
- TP-exemple : gérer un contact

Pourquoi utiliser Qt ?

Je pense qu'il est très important de savoir que la sérialisation des objets n'est pas proposée **que** par Qt. En effet, de nombreuses bibliothèques proposent des outils analogues à ceux de Qt.



[Pourquoi utiliser Qt, et pas une autre bibliothèque comme Boost ?](#)

Je pourrais être tenté de vous répondre simplement "pourquoi pas ?", mais le choix de Qt ne relève pas d'une simple décision arbitraire.

Le premier avantage de Qt est que lorsqu'on souhaite sérialiser un objet, on n'a pas besoin de modifier le code de l'objet concerné. Si on prend exemple sur la bibliothèque Boost qui propose aussi une manière de sérialiser vos objets, Boost vous impose d'ajouter une fonction *serialize*.

Dans le cas où vous utilisez Qt, toutes les opérations de sérialisation se déroulent hors de la classe. Cet avantage peut aussi permettre la sérialisation d'objets auxquels on n'a pas accès au code source, comme c'est souvent le cas pour des objets situés dans des bibliothèques.

La seconde raison de mon choix vient du fait que, contrairement à Boost, Qt utilise une seule et unique classe pour vous permettre de sérialiser vos objets. Donc pas besoin de plonger à chaque fois dans la documentation d'une demi-douzaine de classes pour réussir votre sérialisation.

Et enfin, la dernière raison est qu'avec Qt, il y a très peu de code à ajouter pour sérialiser un objet, et que celui-ci peut être placé à peu près n'importe où.

Sérialiser avec QVariant

Pour débiter, je vous propose d'apprendre à sérialiser vos objets.



Désolé de déjà t'interrompre, mais tu entends quoi par "sérialiser" ?

Pour tout savoir, allez sur [wikipédia](#). Même si ce n'est pas très clair, continuez à lire ce tutoriel, vous verrez vite à quoi ça peut servir.

Afin de simplifier la sérialisation d'objet, Qt a créé une classe dédiée à la sérialisation : [QVariant](#). D'une manière extrêmement générale, on peut considérer que QVariant est la représentation d'un objet quelconque sérialisé. Le fonctionnement de QVariant est très simple : à partir du moment où un objet est sérialisable (on verra un peu plus bas comment faire pour rendre un objet sérialisable), on peut sérialiser l'objet et le placer dans un QVariant. Ensuite vous en faites ce que vous voulez.

Pour éviter l'apparition de multiples sérialisations des objets "de base" de Qt, ceux-ci ont déjà été déclarés comme sérialisables. Donc QVariant fonctionne déjà avec la plus grande partie des objets du module [QtCore](#), ainsi qu'une partie de ceux du module [QtGui](#). Une liste (malheureusement incomplète) des objets sérialisables avec QVariant est disponible [ici](#).

On va donc procéder en deux parties : dans un premier temps, on va voir comment faire pour utiliser QVariant avec des objets sérialisables. Puis ensuite, on va voir comment faire pour rendre un objet quelconque sérialisable.

QVariant et les objets sérialisables

On dispose de trois façons pour créer un QVariant à partir d'un objet sérialisable (ça permet une manipulation plus aisée) :

- Par le [constructeur](#) : c'est la méthode par défaut, mais elle n'est possible qu'avec les objets sérialisables de Qt (donc pas avec vos objets sérialisables) ;
- Par la méthode [setValue](#), qui affecte une copie de l'objet au QVariant concerné ;
- Par la méthode (statique) [fromValue](#), qui crée un nouveau QVariant à partir d'une copie de l'objet donné.



Les deux dernières méthodes citées ne fonctionnent pas avec le compilateur MSVC 6. Si c'est celui dont vous vous servez, vous pouvez utiliser respectivement les fonctions [qVariantSetValue](#) et [qVariantFromValue](#).

On distingue encore trois manières de récupérer la classe voulue à partir d'un QVariant (dans ce cas là, on parle de *désérialisation*) :

- Par les fonctions de la forme "toClass", où *Class* représente le type à obtenir ([exemple](#) : [toInt](#) pour obtenir un entier, ou encore [toString](#) pour renvoyer une [QString](#)). Ces fonctions ne sont disponibles que pour les objets sérialisables de Qt ;
- Par la fonction [value](#), qui renvoie la classe demandée ([exemple](#) : `var.value<QString>()` qui renvoie un [QString](#)) ;
- Par la fonction [qvariant_cast](#), qui s'emploie comme la précédente : `qvariant_cast<QString>(var)` ; .



Comme précédemment, la fonction [value](#) ne marche pas avec MSVC 6. Dans ce cas, vous pouvez employer la fonction [qVariantValue](#).

Dans la suite de ce tutoriel, je vais utiliser les fonctions [setValue](#) et [value](#), mais vous pouvez vous servir de celles que vous voulez.

Voici un petit exemple qui montre la sérialisation puis la désérialisation d'une chaîne de caractère ([QString](#)) et puis qui compare le résultat :

Code : C++

```
#include <QCoreApplication>
#include <QString>
#include <QVariant>

#include <iostream>
```

```
int main (int argc, char ** argv)
{
    QApplication app(argc, argv);
    QString initial = "Ceci est le texte à sérialiser", final = "";

    QVariant chaine serialise;
    chaine serialise.setValue(initial);
    final = chaine_serialise.value<QString>();

    if(initial == final)
    {
        std::cout << "Les deux chaines sont identiques" << std::endl;
    }
    else
    {
        std::cout << "Les deux chaines sont differentes" << std::endl;
    }
    getchar();

    app.quit();
    return 0;
}
```



N'oubliez pas de rajouter "CONFIG += console" à votre .pro, sinon la console n'affichera rien.

Juste avec la sérialisation des objets de Qt, on dispose déjà de quelque chose d'extrêmement puissant. Mais Qt fourni aussi la possibilité de rendre sérialisable n'importe quel objet, même appartenant à une autre librairie dont le code source ne serait pas disponible.

Rendre un objet quelconque sérialisable

Bon, malgré le coté parfait de QVariant, on ne peut pas tout faire non plus :o. Afin de pouvoir rendre sérialisable un objet, celui-ci doit respecter les trois règles suivantes :

- L'objet doit posséder un constructeur public par défaut (ou alors tous ses arguments doivent avoir une valeur par défaut) ;
- L'objet doit posséder un constructeur de copie public ;
- L'objet doit posséder un destructeur public.



Il n'est pas obligatoire de définir soi-même ces fonctions puisque celles fournies par défaut conviennent très bien (c'est notamment utile pour les structures).

À partir du moment où ces trois conditions sont respectées, l'objet est prêt à être sérialisé. Pour faire cela, il suffit d'appeler la macro `Q_DECLARE_METATYPE` de la manière qui suit (dans le cas où on cherche à rendre sérialisable un objet *MaClasse*) :

Code : C++

```
Q_DECLARE_METATYPE (MaClasse)
```

Ce morceau de code est à placer hors de tout objet et de toute fonction. On le met, la plupart du temps, à la suite de la déclaration de l'objet concernée.



Tout à l'heure, tu as dit qu'il était possible de rendre sérialisable un objet qui appartient à une bibliothèque (dont le code source est inaccessible). Je le met où, ce code, alors ? 🤔

L'endroit n'a pas une grande importance, tant que ce code précède **toutes** les fonctions qui vont manipuler l'objet. En général, on le place dans le ou les fichier(s) qui inclue(nt) la librairie en question, juste après l'inclusion :

Inclusion.h

Code : C++

```
#ifndef INCLUDE_H
#define INCLUDE_H

// Voici un exemple avec la structure SDL_Event de la bibliothèque
SDL

#include <QVariant>
#include <SDL.h>
Q_DECLARE_METATYPE(SDL_Event);

#endif
```

Notez que j'ai appelé la macro **après** avoir inclus `QVariant`, ce qui est parfaitement logique. Pour ceux qui ont du mal à saisir, jetez un coup d'oeil au TP-exemple à la fin de ce tutoriel.

Le stockage avec `QDataStream`



Pourquoi parles-tu de stocker avec `QDataStream` ? `QVariant` ne suffit-il pas ?

Non. `QVariant` est une classe très puissante, mais elle ne peut pas tout faire non plus. Cette partie est là pour résoudre un problème qui apparaît lorsqu'on tente d'enregistrer un objet sérialisé dans un fichier (par exemple). En effet, jusqu'à maintenant, on s'est contenté de dire à Qt que tel ou tel objet pouvait être sérialisé. Ces déclarations suffisaient totalement à `QVariant` pour gérer les objets concernés. Mais dans le cadre de l'enregistrement (en général dans un fichier) de l'objet, `QVariant` a besoin de savoir quelles données et surtout dans quel ordre ces données de l'objet doivent être stockées.

Contrairement à Boost qui impose de créer une fonction *serialize* dans l'objet (ce qui peut se révéler réducteur), Qt profite du fait qu'on peut sur-définir les opérateurs. Dans le cas de la sérialisation, ce sont les opérateurs de flux ("`<<`" et "`>>`") vers `QDataStream` qui ont été choisis.

Les opérateurs de flux vers `QDataStream`

La classe `QDataStream` est là pour fournir un moyen de stocker des suites d'octets, généralement dans un fichier. Avant de passer à la sur-définition des opérateurs de flux manquants, je vais vous expliquer pourquoi est-ce qu'il sera très simple de les mettre en œuvre.

D'origine, `QDataStream` possède de nombreux opérateurs de flux déjà définis pour les classes de Qt utiles dans ce contexte (la liste complète est visible ici : [Format des opérateurs de QDataStream](#)). Puisque la plupart des objets que l'on crée ou rencontre sont majoritairement constitués de ces types, il suffit de réemployer ces opérateurs.

On va donc réaliser les opérateurs "`<<`" et "`>>`" vers `QDataStream`. Voici leur prototype (à placer en-dehors de la déclaration de l'objet concerné) :

Code : C++

```
QDataStream & operator << (QDataStream & out, const MaClasse &
Valeur);
QDataStream & operator >> (QDataStream & in, MaClasse & Valeur);
```

Comme la plupart des opérateurs de flux, ils renvoient le `QDataStream` donné en paramètre, afin de permettre l'enchaînement des opérateurs.

La technique à employer est simple : pour chaque variable membre (que vous voulez stocker) de l'objet, envoyez celle-ci vers le

QDataStream (ou bien sortez-l'en).

Ainsi que vous l'avez peut-être vu dans la [liste des opérateurs de flux](#) de QDataStream, il est possible d'envoyer directement des [QList](#), [QMap](#) et autres conteneurs au QDataStream. La seule condition est que le type du conteneur ait des opérateurs de flux vers QDataStream.



Les valeurs à stocker doivent être envoyées et sorties d'un QDataStream dans le même ordre, ou bien elles ne correspondront plus.

Voici un exemple, où *nom* est une QString et *numero_rue* un entier :

Code : C++

```
QDataStream & operator << (QDataStream & out, const MaClasse &
Valeur)
{
    out << Valeur.nom << static_cast<quint16>(Valeur.numero_rue);

    return out;
}
```

Extrêmement difficile, n'est-ce pas ? 🤔



Pour ceux qui veulent être véritablement portables et durables dans le temps, vous devriez consulter les informations relatives à la version du QDataStream. Tout y est très bien expliqué.

Pour terminer, il reste deux fonctions de la classe [QMetaType](#) à appeler pour indiquer à QVariant que les opérateurs de flux ont été définis (il ne le devine pas tout seul o_O). Pour des raisons pratiques, je les place généralement dans une fonction *initMaClasseSystem*, que j'appelle dans le "main".

Les voici :

- [qRegisterMetaTypeStreamOperators](#) : c'est la fonction qui finit de déclarer pour QVariant les opérateurs de flux (facile à deviner vu le nom). On l'utilise comme suit :
`qRegisterMetaTypeStreamOperators<MaClasse> ("MaClasse") ;`
- [qMetaTypeId](#) : elle n'est pas obligatoire, mais très utile : en effet, dans le cas où le [Q_DECLARE_METATYPE](#) échoue (conditions non respectées, par exemple), la compilation échoue elle aussi (ne vous étonnez pas si le compilateur vous envoie des erreurs concernant le code source de Qt). Vous pouvez l'utiliser ainsi : `qMetaTypeId<MaClasse> () ;`

Comme je le disais ci-dessus, il suffit de les mettre dans une fonction d'initialisation par exemple, et d'appeler celle-ci dans le main (un seul appel suffit).

TP-exemple : gérer un contact

Que ce soit pour tester vos connaissances fraîchement acquises ou pour comprendre un peu mieux ce qui vient d'être dit, voici un TP expliqué pas à pas pour vous guider dans la sérialisation complète d'une classe représentant un contact.

La classe *Contact*

La classe *Contact* comporte cinq propriétés : le numéro de la rue, l'adresse, le nom, la date d'anniversaire et la liste des sites web du contact. Chacune des propriétés a son propre type, afin de bien mettre en évidence la puissance de ce que je vous explique. En outre, les sites webs sont représentés par une structure incluant le nom et l'adresse du site web.

Le but n'est pas de travailler sur la réalisation de la classe *Contact* elle-même, aussi je vous donne directement son code :

Contact.h

Code : C++

```
#ifndef CONTACT_H
#define CONTACT_H
```

```

#include <QUrl>
#include <QString>
#include <QList>
#include <QDate>

class Contact // Représentation d'un Contact
{
public:
    struct SiteWeb
    {
        SiteWeb (const QString Nom = "", const QUrl Adr = "");

        QString name;
        QUrl adresse;
    };
    typedef QList<SiteWeb> SiteWebList;

    Contact (const quint16 NumeroMaison = 0,
             const QString Adresse = "",
             const QString Nom = "",
             const QDate Aniv = QDate(),
             const SiteWebList Sites = SiteWebList()); // Constructeur par
défaut public
    Contact (const Contact & Copie); // Constructeur de copie
public
    ~Contact (); // Destructeur public

    void afficher () const; // Affiche les informations du
contact dans la console

private:
    quint16 m_numeroMaison; // Le n° de la maison
    QString m_adresse; // L'adresse (sans le n°)
    QString m_nom; // Le nom
    QDate m_aniv; // La date d'anniversaire

    SiteWebList m_sites; // La liste des sites webs
};

#endif

```

Contact.cpp

Code : C++

```

#include "Contact.h"

#include <iostream>

Contact::SiteWeb::SiteWeb (const QString Nom, const QUrl Adr)
{
    name = Nom;
    adresse = Adr;
}

Contact::Contact (const quint16 NumeroMaison,
                 const QString Adresse,
                 const QString Nom,
                 const QDate Aniv,
                 const SiteWebList Sites)
: m_numeroMaison(NumeroMaison),
  m_adresse(Adresse),
  m_nom(Nom),
  m_aniv(Aniv),
  m_sites(Sites)

```



```

{}
Contact::Contact (const Contact & Copie)
{
    m_numeroMaison = Copie.m_numeroMaison;
    m_adresse = Copie.m_adresse;
    m_nom = Copie.m_nom;
    m_aniv = Copie.m_aniv;
    m_sites = Copie.m_sites;
}
Contact::~~Contact()
{}

void Contact::afficher () const
{
    std::cout << "Le contact "
        << m_nom.toStdString()
        << " nee le "
        << m_aniv.day() << "/" << m_aniv.month() << "/" << m_aniv.year()
        << " et qui loge au "
        << static_cast<int>(m_numeroMaison) << " " <<
        m_adresse.toStdString()
        << " possede les sites webs suivant : ";

    SiteWeb curr;
    foreach(curr, m_sites)
        std::cout << curr.name.toStdString() << ":" <<
        curr.adresse.toString().toStdString() << " - ";

    std::cout << std::endl;
}

```

J'ai simplement ajouté une fonction *afficher*, qui résume les informations sur le contact.

Rendre sérialisable notre objet

A priori, cela ne devrait pas poser de problème, mais voici les détails.

La première chose, c'est d'ajouter `QVariant` à notre fichier (sans quoi le compilateur ne trouvera pas la macro `Q_DECLARE_METATYPE`), au début :

Code : C++

```
#include <QVariant>
```

Ensuite, il faut étendre la classe.

Code : C++

```
Q_DECLARE_METATYPE(Contact::SiteWeb)
Q_DECLARE_METATYPE(Contact)
```

Comme la structure *SiteWeb* n'est pas accessible directement en-dehors de la classe *Contact*, il faut se servir de l'opérateur de résolution de portée `::`. Le code est à mettre après la déclaration de la classe *Contact*.

Définir les opérateurs de flux

Déclaration de Contact (Contact.h)

Bon, à partir de là c'est beaucoup plus dur, alors on va y aller doucement. On commence d'abord par mettre les prototypes, sans oublier ceux de *Position3D* si on veut que ça fonctionne :

Code : C++

```
QDataStream & operator << (QDataStream & out, const Contact::SiteWeb
& Valeur);
QDataStream & operator >> (QDataStream & in, Contact::SiteWeb &
Valeur);

QDataStream & operator << (QDataStream & out, const Contact &
Valeur);
QDataStream & operator >> (QDataStream & in, Contact & Valeur);
```

Comme précédemment, on utilise l'opérateur "::" pour accéder à *SiteWeb*.

Il serait bien utile de pouvoir accéder aux variables membres privées de la classe *Contact*, mais les prototypes sont hors de la classe alors on ne peut pas. Afin de résoudre ce problème, nous allons employer les fonctions amies. Il faut donc ajouter le code suivant, dans la classe *Contact* :

Code : C++

```
/* Les opérateurs de flux sont des fonctions amies */
friend QDataStream & operator << (QDataStream &, const
Contact::SiteWeb &);
friend QDataStream & operator >> (QDataStream &, Contact::SiteWeb
&);

friend QDataStream & operator << (QDataStream &, const Contact &);
friend QDataStream & operator >> (QDataStream &, Contact &);
```

Avec ça, on obtient le fichier *Contact.h* suivant :

Contact.h

Code : C++

```
#ifndef CONTACT_H
#define CONTACT_H

#include <QVariant>

#include <QUrl>
#include <QString>
#include <QList>
#include <QDate>

class Contact // Représentation d'un Contact
{
public:
    struct SiteWeb
    {
        SiteWeb (const QString Nom = "", const QUrl Adr = "");

        QString name;
        QUrl adresse;
    };
    typedef QList<SiteWeb> SiteWebList;

    Contact (const quint16 NumeroMaison = 0,
             const QString Adresse = "",
             const QString Nom = "",
             const QDate Aniv = QDate(),
             const SiteWebList Sites = SiteWebList()); // Constructeur par
défaut public
    Contact (const Contact & Copie); // Constructeur de copie
```

```

public
    ~Contact ();                // Destructeur public

    void afficher () const;      // Affiche les informations du
    contact dans la console

private:
    quint16 m_numeroMaison;      // Le n° de la maison
    QString m_adresse;           // L'adresse (sans le n°)
    QString m_nom;               // Le nom
    QDate m_aniv;                // La date d'anniversaire

    SiteWebList m_sites;         // La liste des sites webs

    /* Les opérateurs de flux sont des fonctions amies */
    friend QDataStream & operator << (QDataStream &, const
Contact::SiteWeb &);
    friend QDataStream & operator >> (QDataStream &, Contact::SiteWeb
&);

    friend QDataStream & operator << (QDataStream &, const Contact
&);
    friend QDataStream & operator >> (QDataStream &, Contact &);
};

Q_DECLARE_METATYPE(Contact::SiteWeb)
QDataStream & operator << (QDataStream & out, const
Contact::SiteWeb & Valeur);
QDataStream & operator >> (QDataStream & in, Contact::SiteWeb &
Valeur);

Q_DECLARE_METATYPE(Contact)
QDataStream & operator << (QDataStream & out, const Contact &
Valeur);
QDataStream & operator >> (QDataStream & in, Contact & Valeur);

#endif

```

Définition de Contact (Contact.cpp)

Maintenant que l'on a déclaré les prototypes, il ne reste plus qu'à définir les opérateurs. Je commence par l'opérateur "<<" entre *SiteWeb* et *QDataStream*:

Code : C++

```

QDataStream & operator << (QDataStream & out, const Contact::SiteWeb
& Valeur)
{
    out << Valeur.name
    << Valeur.adresse;

    return out;
}

```

Pas grand chose de nouveau sous le soleil, je passe mes valeurs à la suite puis je renvoie le flux. Au tour de l'opérateur inverse (">>"):

Code : C++

```

QDataStream & operator >> (QDataStream & in, Contact::SiteWeb &
Valeur)
{
    in >> Valeur.name;
    in >> Valeur.adresse;
}

```

```
return in;
}
```

Là, je fais donc exactement le contraire : je récupère mes valeurs une à une, puis je renvoie à nouveau le flux. Vous pouvez noter que j'ai respecté l'ordre d'entrée pour faire l'ordre de sortie (on ne tient pas à mélanger des valeurs).

Pour continuer, on peut ajouter la définition des deux opérateurs mais pour la classe *Contact*. Puisque on a déjà défini les opérateurs pour *SiteWeb*, on peut directement le passer ou le recevoir de *QDataStream* :

Code : C++

```
QDataStream & operator << (QDataStream & out, const Contact &
Valeur)
{
    out << Valeur.m_numeroMaison
    << Valeur.m_adresse
    << Valeur.m_nom
    << Valeur.m_aniv
    << Valeur.m_sites;

    return out;
}
QDataStream & operator >> (QDataStream & in, Contact & Valeur)
{
    in >> Valeur.m_numeroMaison;
    in >> Valeur.m_adresse;
    in >> Valeur.m_nom;
    in >> Valeur.m_aniv;
    in >> Valeur.m_sites;

    return in;
}
```

Si vous n'y arrivez pas, ne sombrez pas dans un profond désespoir :lol:. Il faut toujours un petit moment, le temps de s'habituer. Après deux, trois tentatives, vous vous débrouillerez comme des pros.

Comme je l'ai indiqué à la fin de la sous-partie précédente, il manque encore deux fonctions. Je les ai rassemblé dans une même fonction : *initContactSystem*, que je déclare comme fonction statique de *Contact* :

Code : C++


```
static void initContactSystem ();
```

Rien de transcendant, un simple prototype. Enfin, il ne nous reste plus qu'à la définir (j'ai également ajouté les fonctions de test) :

Code : C++

```
void Contact::initContactSystem ()
{
    qRegisterMetaTypeStreamOperators<Contact::SiteWeb>("Contact::SiteWeb");
    ;
    qRegisterMetaTypeStreamOperators<Contact>("Contact");

    qMetaTypeId<Contact::SiteWeb>(); // Teste la validité de la structure
    SiteWeb
    qMetaTypeId<Contact>(); // Teste la validité de la classe Contact
}
```

Et voilà, c'est fini. Toujours en vie ? 

La fonction main

Bon, ça été long, parfois difficile, mais si vous lisez ça c'est sûrement que vous avez survécu 😊. Alors, pour vous remercier de m'avoir suivi jusqu'ici, je vous donne une fonction *main* toute faite pour tester notre classe *Contact*. Je teste la transformation en *QVariant* avec la fonction `qVariantFromValue`, ainsi que l'opérateur inverse avec la fonction `value` (ligne 21).

Pour mettre à l'épreuve notre stockage dans un fichier, j'emploie *QSettings* (lisez la [doc](#) et [ce tuto](#) de [granarc](#) pour en savoir plus sur son utilité et comment l'utiliser). En fait, j'écris notre classe *Contact* sous la forme d'un *QVariant* (si vous ouvrez le fichier *Test.ini*, qui apparaît après l'exécution, vous pourrez le voir) ; puis je lis le fichier que je viens d'écrire, et j'affiche notre contact afin de voir si c'est le même.

Le code pour faire tout ça :

main.cpp

Code : C++

```
#include "Contact.h"

#include <QFile>
#include <QSettings>

int main()
{
    Contact::initContactSystem();
    QFile::remove("Test.ini");

    Contact::SiteWeb site1(QString("Site du Zero"),
        QUrl("www.siteduzero.com")), site2(QString("Google"),
        QUrl("www.google.fr"));
    Contact::SiteWebList sites;
    sites << site1 << site2;

    Contact smith(12, "boulevard des sports", "Smith", QDate(1972, 7,
        14), sites);
    smith.afficher();

    QSettings fichier_ecrire("Test.ini", QSettings::IniFormat);
    fichier_ecrire.setValue("Contact", qVariantFromValue(smith));
    fichier_ecrire.sync();

    Contact contact_lu;
    QSettings fichier_lire("Test.ini", QSettings::IniFormat);
    contact_lu = fichier_lire.value("Contact",
        qVariantFromValue(Contact()).value<Contact>());
    contact_lu.afficher();

    system("PAUSE");
    return 0;
}
```

Il ne reste plus qu'à compiler... (toujours avec "CONFIG+= console" dans votre .pro) et exécuter votre super fichier ".exe", et tadam !

Code : Console

```
Le contact Smith nee le 14/7/1972 et qui loge au 12 boulevard des sports possede le
Google:www.google.fr -
Le contact Smith nee le 14/7/1972 et qui loge au 12 boulevard des sports possede le
Google:www.google.fr -
```

Ce sont les mêmes ! Vous avez réussi, bravo :-°.

Les fichiers finaux

Bon, un dernier cadeau pour la route : tous les fichiers, complets cette fois ci :

Contact.h

Code : C++

```
#ifndef CONTACT_H
#define CONTACT_H

#include <QVariant>

#include <QUrl>
#include <QString>
#include <QList>
#include <QDate>

class Contact // Représentation d'un Contact
{
public:
    struct SiteWeb
    {
        SiteWeb (const QString Nom = "", const QUrl Adr = "");

        QString name;
        QUrl adresse;
    };
    typedef QList<SiteWeb> SiteWebList;

    Contact (const quint16 NumeroMaison = 0,
             const QString Adresse = "",
             const QString Nom = "",
             const QDate Aniv = QDate(),
             const SiteWebList Sites = SiteWebList()); // Constructeur par
défaut public
    Contact (const Contact & Copie); // Constructeur de copie
public
    ~Contact (); // Destructeur public

    void afficher () const; // Affiche les informations du
contact dans la console

    static void initContactSystem ();

private:
    quint16 m_numeroMaison; // Le n° de la maison
    QString m_adresse; // L'adresse (sans le n°)
    QString m_nom; // Le nom
    QDate m_aniv; // La date d'anniversaire

    SiteWebList m_sites; // La liste des sites webs

    /* Les opérateurs de flux sont des fonctions amies */
    friend QDataStream & operator << (QDataStream &, const
Contact::SiteWeb &);
    friend QDataStream & operator >> (QDataStream &, Contact::SiteWeb
&);

    friend QDataStream & operator << (QDataStream &, const Contact
&);
    friend QDataStream & operator >> (QDataStream &, Contact &);
};

Q_DECLARE_METATYPE(Contact::SiteWeb)
QDataStream & operator << (QDataStream & out, const
Contact::SiteWeb & Valeur);
```

```

    QDataStream & operator >> (QDataStream & in, Contact::SiteWeb &
    Valeur);

    Q_DECLARE_METATYPE(Contact)
    QDataStream & operator << (QDataStream & out, const Contact &
    Valeur);
    QDataStream & operator >> (QDataStream & in, Contact & Valeur);

#endif

```

Contact.cpp

Code : C++

```

#include "Contact.h"

#include <iostream>

Contact::SiteWeb::SiteWeb (const QString Nom, const QUrl Adr)
{
    name = Nom;
    adresse = Adr;
}

Contact::Contact (const quint16 NumeroMaison,
                  const QString Adresse,
                  const QString Nom,
                  const QDate Aniv,
                  const SiteWebList Sites)
: m_numeroMaison(NumeroMaison),
  m_adresse(Adresse),
  m_nom(Nom),
  m_aniv(Aniv),
  m_sites(Sites)
{}

Contact::Contact (const Contact & Copie)
{
    m_numeroMaison = Copie.m_numeroMaison;
    m_adresse = Copie.m_adresse;
    m_nom = Copie.m_nom;
    m_aniv = Copie.m_aniv;
    m_sites = Copie.m_sites;
}

Contact::~Contact()
{}

void Contact::afficher () const
{
    std::cout << "Le contact "
              << m_nom.toStdString()
              << " nee le "
              << m_aniv.day() << "/" << m_aniv.month() << "/" << m_aniv.year()
              << " et qui loge au "
              << static_cast<int>(m_numeroMaison) << " " <<
    m_adresse.toStdString()
              << " possede les sites webs suivant : ";

    SiteWeb curr;
    foreach(curr, m_sites)
        std::cout << curr.name.toStdString() << ":" <<
        curr.adresse.toString().toStdString() << " - ";

    std::cout << std::endl;
}

```

```

void Contact::initContactSystem ()
{
    qRegisterMetaTypeStreamOperators<Contact::SiteWeb>("Contact::SiteWeb")
;
    qRegisterMetaTypeStreamOperators<Contact>("Contact");

    qMetaTypeId<Contact::SiteWeb>(); // Teste la validité de la structure
SiteWeb
    qMetaTypeId<Contact>();        // Teste la validité de la classe Contact
}

QDataStream & operator << (QDataStream & out, const Contact::SiteWeb &
Valeur)
{
    out << Valeur.name
        << Valeur.adresse;

    return out;
}

QDataStream & operator >> (QDataStream & in, Contact::SiteWeb & Valeur)
{
    in >> Valeur.name;
    in >> Valeur.adresse;

    return in;
}

QDataStream & operator << (QDataStream & out, const Contact & Valeur)
{
    out << Valeur.m_numeroMaison
        << Valeur.m_adresse
        << Valeur.m_nom
        << Valeur.m_aniv
        << Valeur.m_sites;

    return out;
}

QDataStream & operator >> (QDataStream & in, Contact & Valeur)
{
    in >> Valeur.m_numeroMaison;
    in >> Valeur.m_adresse;
    in >> Valeur.m_nom;
    in >> Valeur.m_aniv;
    in >> Valeur.m_sites;

    return in;
}

```

main.cpp

Code : C++

```

#include "Contact.h"

#include <QFile>
#include <QSettings>

int main()
{
    Contact::initContactSystem();
    QFile::remove("Test.ini");

    Contact::SiteWeb site1(QString("Site du Zero"),
        QUrl("www.siteduzero.com")), site2(QString("Google"),
        QUrl("www.google.fr"));
    Contact::SiteWebList sites;

```



```
sites << site1 << site2;

Contact smith(12, "boulevard des sports", "Smith", QDate(1972, 7,
14), sites);
smith.afficher();

QSettings fichier_ecrire("Test.ini", QSettings::IniFormat);
fichier_ecrire.setValue("Contact", qVariantFromValue(smith));
fichier_ecrire.sync();

Contact contact_lu;
QSettings fichier_lire("Test.ini", QSettings::IniFormat);
contact_lu = fichier_lire.value("Contact",
qVariantFromValue(Contact()))<Contact>();
contact_lu.afficher();

    system("PAUSE");
return 0;
}
```

Et voilà, c'est fini (hé oui, déjà).

N'hésitez surtout pas à vous entraîner (ne vous en faites pas, le coup de main vient rapidement) et à faire plein d'essais, avec toutes les combinaisons possibles (une classe, dans une classe, dans une ... :lol:). Et si vous avez des questions, il y a le forum.

Partager

