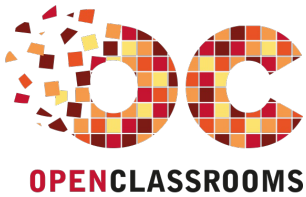
Qt : Quelques fonctionnalités non-GUI

Par tobast



www.openclassrooms.com

Dernière mise à jour le 12/04/2011

Sommaire

Qt : Quelques fonctionnalités non-GUI

Par 

tobast

Mise à jour : 12/04/2011
Difficulté : Facile

Bonjour à tous !

Je suis Tobast, et je vais vous présenter dans ce cours quelques fonctionnalités de Qt qui ne concernent pas la GUI !

En effet, jusqu'à maintenant, sur le SDZ, l'aspect GUI de Qt a été beaucoup présenté, mais Qt permet de faire beaucoup de choses, en dehors de la création de fenêtres. Dans ce tuto, je vais essayer de vous présenter du mieux que je peux certaines utilisations de Qt sortant justement de cette GUI.

J'espère que ce cours vous plaira !

Partie 1 : Fonctionnalités non-GUI

Utiliser les ressources Qt

Dans ce chapitre, nous allons voir comment utiliser les fichiers de ressource dont parlait M@teo21 dans son cours sur Qt designer :

Citation : M@teo21

Resource Browser : un petit utilitaire qui vous permet de naviguer à travers les fichiers de ressources de votre application. Ces fichiers de ressources rappellent un peu ceux de Windows (on en a brièvement parlé dans l'annexe du cours de C, à propos de l'ajout d'icône à un programme sous Windows).
Ici, les fichiers de ressouces portent l'extension .qrc et ont l'avantage d'être compatibles avec tous les OS.

Les fichiers de ressources servent empaqueter des fichiers (images, sons, texte...) au sein même de votre exécutable. Cela permet d'éviter d'avoir à placer ces fichiers dans le même dossier que votre programme, et cela évite donc le risque de les perdre (puisqu'ils se trouveront toujours dans votre exécutable).

C'est un peu hors-sujet, donc je n'en parlerai pas plus ici. Consultez la doc à propos des ressources si vous voulez en savoir plus.

Eh bien, je vais vous expliquer ici comment s'en servir ! C'est très simple, vous allez voir !

Présentation

C'est quoi, concrètement, ces *ressources* dont tu nous parles depuis le début ?

Pour faire simple, les ressources sont des fichiers de tout genre inclus directement *dans* votre exécutable. Le fichier .rc dont M@teo21 parlait en annexe de son cours sur le C pour mettre une icône à son programme **est** un fichier de ressources.
Seulement, ces ressources ne sont compatibles qu'avec Windows et j'ai jamais réussi à les utiliser.
Le gros avantage de Qt, c'est que son fichier de ressources est *multi-plateformes*, il peut donc marcher sur tous les OS gérés par Qt (dont Windows, Linux et Mac, les principaux utilisés).

Ok, donc les ressources c'est des fichiers intégrés à l'exécutable. Mais... Concrètement, ça va me servir à quoi, tout ça ?

À plusieurs choses ! Premièrement, l'utilisateur ne pourra pas enlever de fichiers ressources de votre exécutable (il pourra éventuellement les récupérer avec un programme adapté). Ensuite, et surtout, quel que soit l'emplacement de votre exécutable, les fichiers ressources seront toujours accessibles depuis votre programme ! Utile si on veut par exemple le déplacer dans une clé USB.

Mais... Pour l'instant, il n'y a rien que je ne peux pas faire sans, non ?

Bien que ça soit pratique, en effet, vous pouvez tout faire sans. Mais il y a une dernière chose très pratique dont je ne vous ai pas parlé. Imaginez que vous faites une image que vous mettez dans votre programme. Dessus, vous écrivez "MonProgramme - créé par [pseudo]". Si vous voulez la traduire, il vous faudra changer le code source, et faire un exécutable par pays !
C'est là que les ressources entrent en jeu. Vous pouvez définir plusieurs fois le même "nom", en indiquant le pays pour chacun ! Immaginez que vous mettez en ressource un fichier "MonProgramme - créé par [pseudo]" avec comme langue "Français", et un autre "MySoftware - created by [nick]" sans langue. Ainsi, si votre programme est exécuté sur un PC français, l'image française s'affichera. Mais si vous l'exécutez partout ailleurs, ça sera en anglais !

Mais c'est super bien tout ça ! Elle est où l'arnaque ?

En effet, il y a un défaut à cette méthode. Quand on y réfléchit 2 secondes, ça parait logique, mais **la taille de votre exécutable augmentera** ! Mais puisque les ressources incluses n'ont pas besoin d'être aussi dans le dossier de votre exécutable, la taille de votre application + ses dossiers sera la même.

Créer le fichier de ressources
Structure d'un fichier de ressources Qt

Les fichiers de ressources Qt sont organisés en *préfixes* et *fichiers*.

C'est quoi ces *préfixes* dont tu nous parles ?
Il y a un rapport avec les préfixes en Français ?

D'une certaine manière, oui, il y a un rapport. Pour rappel, un préfixe en Français est un "bout de mot" qui a une signification propre, et qui est placé au début d'un autre mot pour en changer le sens. Par exemple, le préfixe para- signifie "qui protège de". Ainsi, un **para**pluie protège de la pluie, un **para**foudre protège de la foudre, une **para**bole protège de... Ah non, là ça marche pas en fait  .

Un préfixe, dans Qt, est un "dossier" qui sert à regrouper les fichiers de même type. Par exemple, si votre fichier de ressource contient des icônes et des fichiers sons, vous créerez sûrement un préfixe "icones" et un "sons".

Et... Les fichiers, eux, sont tout simplement des fichiers de votre disque dur dont vous indiquez le chemin, et qui seront inclus à la compilation.

On pourrait donc symboliser un fichier de ressources comme ça :

Code : Autre

```
ressources
|
préfixe 1
|-- fichier 1
|-- fichier 2
préfixe 2
|-- fichier 3
```

Comme vous le voyez, un préfixe peut contenir autant de fichiers que l'on veut.

Créer le fichier de ressources

Un fichier de ressources Qt est un fichier **.qrc** (facile à retenir, Qt ResourCe)

Pour créer votre fichier ressources, il y a plusieurs possibilités :

- Vous utilisez comme IDE Qt Creator : très simple, un outil de création de ressources intégré vous permet de créer vos fichiers ressource.
- Vous n'utilisez pas Qt Creator, dans ce cas ~~vous êtes dans la...~~ vous allez devoir créer votre fichier de ressources "à la main" (ou plutôt au clavier).

Nous allons commencer par la méthode "à la main". Ça fera pas de mal aux utilisateurs de Qt Creator de la lire aussi, ça permet de voir comment c'est fait exactement  .

"À la main"

Un fichier ressource n'est rien d'autre qu'un fichier texte un peu spécial. Commencez donc par ouvrir votre éditeur de texte (bloc-notes ou notepad++ sous Windows, Gedit/Vim ou autres sous Ubuntu), et attaquons-nous de suite à la syntaxe d'un fichier qrc !

Pour ceux qui connaissent, un fichier qrc ressemble un peu à du HTML (c'est du XML pour être précis, un *langage de balisage*

dont dérive le HTML). Il commence donc par un *doctype* (plus court qu'un doctype HTML, je vous rassure) :

Code : XML - qrc

```
<!DOCTYPE RCC>
```

Ensuite, tout comme la balise <html> entoure une page html, la balise <RCC> entoure le qrc :

Code : XML - qrc

```
<!DOCTYPE RCC>
<RCC>

</RCC>
```

Pour créer un préfixe, le code est

Code : XML - qrc

```
<!DOCTYPE RCC>
<RCC>
    <qresource prefix="/monPrefixe">

    </qresource>
</RCC>
```

Ainsi, on met la balise **qresource** (attention, un seul "s" en Anglais !), avec **prefix="/nomDuPrefixe"** comme attribut.

Le nom d'un préfixe commence toujours par un slash ("/") !

Pour insérer des fichiers dans le qrc, encore une fois c'est très simple :

Code : XML - qrc

```
<!DOCTYPE RCC>
<RCC>
    <qresource prefix="/prefix">
        <file>dossier/nomDuFichier.png</file>
    </qresource>
</RCC>
```

La balise **<file>** se place forcément dans un préfixe ! On met ensuite entre **<file>** et **</file>** le chemin du fichier.

Si le nom du fichier est compliqué, il peut être utile de le simplifier avec un *alias* :

Code : XML - qrc

```
<!DOCTYPE RCC>
<RCC>
    <qresource prefix="/prefix">
        <file
alias="icone">dossier/petiteIconePourLaFenetreEtLaBarreDesTaches.png</file>
    </qresource>
```

```
</RCC>
```

Ainsi, avec l'attribut **alias**, dans notre programme la ressource ne sera pas appelée "petiteIconePourLaFenetreEtLaBarreDesTaches.png" mais "icone". Avouez que c'est plus court !

Exercice time !
Vous devez inclure dans votre programme les fichiers suivants :

- fichiers/icones/prog.png
- fichiers/icones/ok.png
- fichiers/icones/annuler.png
- fichiers/sons/connexion.wav
- fichiers/sons/emis_en_cas_d_echec.wav

Organisez-les en préfixes, et organisez-vous bien !

Solution

Secret (cliquez pour afficher)

Code : XML - qrc

```
<!DOCTYPE RCC>
<RCC>
    <qresource prefix="/icones">
        <file>fichiers/icones/prog.png</file>
        <file>fichiers/icones/ok.png</file>
        <file>fichiers/icones/annuler.png</file>
    </qresource>
    <qresource prefix="/sons">
        <file>fichiers/sons/connexion.wav</file>
        <file
alias="echec.wav">fichiers/sons/emis_en_cas_d_echec.wav</file>
    </qresource>
</RCC>
```

J'ai mis un alias au 2nd fichier son, il était un peu long

Et... Pour les langues ?

Il "suffit" de créer un 2nd préfixe appelé comme celui à "traduire", en lui rajoutant un attribut *lang="codeLangue"*, où codeLangue est le code à deux lettres pour identifier un pays (exemple : France = fr). Pour les fichiers à "traduire", vous devrez leur donner le même alias dans toutes les langues. Un petit exemple ?

Code : XML - qrc

```
<!DOCTYPE RCC>
<RCC>
    <qresource prefix="/icones">
        <file alias="annuler">ressources/icones/annuler.png</file>
        <file alias="ok">ressources/icones/ok.png</file>
        <file alias="prog">ressources/icones/prog.png</file>
    </qresource>
    <qresource prefix="/icones" lang="fr">
        <file alias="annuler">ressources/iconesfr/annuler.png</file>
        <file alias="ok">ressources/iconesfr/ok.png</file>
        <file alias="prog">ressources/iconesfr/prog.png</file>
    </qresource>
```

```
</RCC>
```

Ainsi, si l'utilisateur est français, *annuler* aura pour chemin *ressources/iconesfr/annuler.png*. Sinon, *annuler* aura pour chemin *ressources/icones/annuler.png* !

Avec Qt Creator

Qt Creator, l'IDE de Qt, inclut un éditeur de fichiers qrc, très bien fait.

Je considèrerai ici que vous connaissez cet IDE, et que vous le maitrisez !

Premièrement, créez ou ouvrez un projet. Ensuite, faites un clic droit sur le nom du projet, et faites "Ajouter nouveau...", puis choisissez "Fichier de ressource Qt". Donnez-lui un nom, puis créez-le.
Après, est-ce bien la peine d'expliquer ? Faites "ajouter un préfixe" pour ajouter un préfixe, et "ajouter un fichier" pour ajouter un fichier ( sérieux ?)...

Bonus : l'Éditeur de ressources Qt !

Mais... Si j'ai pas le Qt Creator, je suis obligé de créer mes .qrc à la main ?

Bon, je vous l'ai pas dit plus tôt parce que c'était important pour que vous compreniez bien le fonctionnement des ressources que vous appreniez à créer **vous-mêmes** vos fichiers qrc. Mais maintenant que vous avez appris tout ça, je vais vous montrer une dernière alternative (bien que d'autres programmes du même genre existent sûrement) : l'Éditeur de ressources Qt (abrégé qrc editor) !

Il s'agit d'un programme fait par moi-même permettant de créer des fichiers qrc en interface graphique, un peu comme avec l'éditeur de ressources intégré de Qt Creator.
Il est sous licence GNU/GPL, et donc gratuit, et *open-source* : vous pouvez télécharger le code source, le modifier, et le redistribuer toujours gratuitement, sous condition de citer le créateur original (moi, donc). Dans le cas où vous y apporteriez des modifications, ça serait sympa de me contacter pour m'en informer  !

Pour le télécharger, c'est par ici !

Utiliser le fichier ressource

C'est bien joli tout ça, mais moi, je sais toujours pas comment on les utilise dans le programme en lui-même !

Justement, il est temps de l'apprendre ! Alors là, accrochez-vous, c'est très dur ! Prêts ?

Dans un programme Qt, pour utiliser un fichier mis en ressource, il faut mettre au tout début de son chemin un ":".
Un exemple ?

Si vous avez créé le fichier de ressources suivant :

Code : XML - qrc

```
<!DOCTYPE RCC>
<RCC>
    <qresource prefix="/icones">
        <file alias="prog">fichiers/progIcone.png</file>
    </qresource>
</RCC>
```

Vous avez créé le programme suivant :

Code : C++

```
#include <QApplication>
#include <QPixmap>
#include <QLabel>


int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QLabel w;

QPixmap p(":/icones/fichiers/progIcone.png");
    w.setPixmap(p);

    w.show();
    return app.exec();
}
```

Vous voyez à la ligne 11 comment utiliser une ressource ! Il faut donc mettre un chemin sur le format suivant :

Code : Autre

```
: + [préfixe] + / + [chemin complet de la ressource]
```

Mais... À quoi il sert l'alias alors ?

Eh bien, j'ai commencé par vous montrer comment faire sans  . Avec, c'est encore plus simple : le chemin ressemble à ça :

Code : Autre

```
: + [préfixe] + / + [alias]
```

Si on reprend notre programme, la ligne 11 ressemblerait à ça :

Code : C++

```
QPixmap p(":/icones/prog");
```

Ça marche pas chez moi ! Ton code m'affiche une fenêtre sans rien dedans ! 

Dans ce cas, deux/trois trucs à revoir :

- Le fichier .qrc est-il bien dans le même dossier que les fichiers de code source ?
- Avez-vous lancé **qmake -project** depuis l'ajout du .qrc dans le dossier des fichiers source ? (dans le doute, relancez-le)

Si oui, ouvrez le fichier .pro. Une ligne devrait contenir quelque chose comme

Code : Autre

```
RESOURCES += mesressources.qrc
```

Si ce n'est pas le cas, rajoutez-la quelque part (de préférence après les fichiers .cpp et les .h, ça fait plus propre), en modifiant bien sur "mesressources.qrc" par le nom de votre fichier qrc.

Si ça marche toujours pas, c'est sûrement les chemins dans le fichier .qrc qui sont en cause, vérifiez-les !
Voilà, vous savez maintenant créer et utiliser les fichiers de ressource Qt !

Si vous voulez jeter un coup d'œil sur la documentation officielle, c'est par ici !

Hacher avec Qt

Dans ce chapitre, je vais vous expliquer comment **hacher des données avec Qt** !

Hacher ? C'est quoi ça ? Partager dans plusieurs variables ?

Mais non, rien à voir ! Si vous préférez, le hachage est une méthode de chiffrage ne permettant pas de décryptage.

Qt permet de hacher très facilement des données ! Je vais vous montrer comment faire tout au long de ce tuto. Prêt ? Suivez moi !

Théorie

Comme je le disais plus haut, vous allez ici apprendre à **hacher avec Qt** !

Le hachage, pour faire simple, est un "chiffrage" qui ne peut pas être déchiffré.

Je vous vois déjà me demander :

Mais... Si on chiffre, c'est bien pour pouvoir déchiffrer après non ? Si on a pas l'intention de se resservir des données, autant ne pas les enregistrer !

En fait, le hachage ne permet pas de "chiffrer" toutes vos données. Vous pouvez, évidemment, mais il serait un peu inutile de hacher par exemple les meilleurs scores de votre jeu, vu que vous ne pourriez pas les afficher ensuite ! En fait, le hachage est surtout utilisé pour "chiffrer" des données qui seront à nouveau entrées par l'utilisateur, comme son mot de passe.

Ben oui, mais bon... Son mot de passe, on aura quand même besoin de le déchiffrer pour vérifier qu'il est bon ! Non ?

Pas forcément ! Prenons un exemple : On veut vérifier que 3x2 est égal à 6. On a alors deux possibilités de le vérifier.

- Soit on fait 3x2, on obtient 6, et on se rend alors compte que 6 = 6, et que donc 3x2 = 6
- Soit on fait 6/2 et 3x2/2, et on se rend compte que 3 = 3, et que donc 3x2 = 6.

Si on applique cet exemple à notre mot de passe appelé ici "MdP" et son chiffrage appelé ici "MdP chiffré" (ici, pour illustrer les deux exemples, on dira que le mot de passe est chiffré et non haché, donc déchiffrable) , et qu'on a une fonction de chiffrage appelée "chiffre" et une de déchiffrage, "déchiffre" :

- Soit on fait chiffre(MdP), et on se rend compte que chiffre(MdP) = MdP chiffré
- Soit on fait déchiffre(MdP chiffré), et on se rend compte que MdP = déchiffre(MdP chiffré)

Seulement, avec un hachage, on ne peut pas déchiffrer. Sans la fonction "déchiffrer", il ne nous reste donc plus que la première possibilité !

Donc, pour résumer, pour comparer des données à des données hachées, on hache les données de la même manière que celles déjà hachées, et on compare les deux.

C'est bien joli, tout ça, mais après tout, à quoi ça me servirait ? Si l'utilisateur entre son mot de passe, il sera stocké sur son DD, donc à priori, il y a pas trop de risques de piratage !

Tout faux. Déjà, qui vous dit que l'utilisateur ne veut pas éviter que ses parents/ses fils/son chien n'accèdent à son mot de passe ? Ça peut donc être utile de hacher un mot de passe, même sur son propre disque dur.

Ensuite, imaginez que vous programmez une messagerie instantanée, par exemple (si vous préférez, un genre d'MSN). Tous les comptes créés seront sûrement stockés soit dans un fichier sur votre serveur, soit dans une base de données.

Si un pirate accédait à ce fichier ou cette base de données, vous imaginez un peu la catastrophe, dans le cas où les mots de passe sont en clair ! Il aurait alors accès à tous les comptes de tous les utilisateurs de votre programme ! Alors que vu qu'un hash (on appelle "hash" le résultat d'un hachage) est indéchiffrable... Le pirate ne pourra rien en faire.

Prêts pour passer au codage ? Alors, c'est parti !

Hacher avec Qt

Tous les liens portant le nom d'une classe Qt seront des liens vers la doc' de Qt 4.6 !

Bon, c'est fini avec la théorie, cette fois, on va vraiment programmer ! 

Pour hacher, on va utiliser une classe du doux nom (un peu long aussi) de QCryptographicHash. Tout se fera via la méthode statique hash.
Voyons déjà son prototype :

Code : C++

```
QByteArray QCryptographicHash::hash ( const QByteArray & data,
Algorithm method )   [static]
```

On voit donc plusieurs choses :

- Cette méthode est statique, mais ça je vous l'ai déjà dit  . Pour rappel, une méthode statique peut être appelée sans avoir créé d'objet, avec `Classe::methode()` .
- La méthode retourne un QByteArray, soit un tableau d'octets.
- Elle prend en premier paramètre un autre QByteArray, qui ne sera pas modifié (const).
- Elle prend en second paramètre Algorithm, qui est une énumération définie dans QCryptographicHash.

Détaillons ses paramètres :

- **data** est tout simplement ce que la fonction va chiffrer.
- **method** est la méthode de hachage. Car oui, il y a plusieurs méthodes !

Reprenons donc le code de base de Qt :

Code : C++

```
#include <QApplication>
#include <QWidget>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget w;

    w.show();
    return app.exec();
}
```

On ne va ici utiliser que le main.

Ajoutons la fonction de hachage :

Code : C++

```
#include <QApplication>
#include <QWidget>
#include <QCryptographicHash>
#include <QByteArray>

int main(int argc, char *argv[])
```

```
{
    QApplication app(argc, argv);
    QWidget w;

    QByteArray ba = QCryptographicHash::hash(/*arguments*/);

    w.show();
    return app.exec();
}
```

Eh, au fait, tu nous as toujours pas dit qu'est-ce qu'on met comme arguments ! Et si j'ai pas de QByteArray, mais que je veux hacher... Je sais pas, moi ! Une chaine !
Et puis, d'abord, on met quoi en deuxième argument, concrètement ?

Je vais commencer par répondre à la première, ~~parce que c'est la plus simple~~ pour faire ça dans l'ordre :

Pour envoyer une chaine de caractères (ce qui sera le cas si on veut hacher un mot de passe, par exemple), il faut que ça soit une QString, et envoyer à **hash()** sa conversion en UTF-8 :

Code : C++

```
QCryptographicHash(chaine.toUtf8(), /*arg 2*/); // chaine est une
QString
```

Ensuite, pour les différentes méthodes. Qt en propose à ce jour 3 :

- En **MD4** (wikipedia) : Cette méthode n'est pas sûre, et peut être cassée, mais elle a l'avantage d'être rapide. Dans l'absolu, mieux vaut éviter de l'utiliser, sauf si la rapidité de l'algorithme est vraiment importante. Pour l'utiliser, le second paramètre est **QCryptographicHash::Md4**.
- En **MD5** (wikipedia) : Une méthode assez sûre, bien qu'elle ne le soit de moins en moins avec l'apparition de "dictionnaires Md5/texte", et la possibilité de générer des collisions (merci à Phacog pour l'info sur les collisions !). Pour l'utiliser, le second paramètre est **QCryptographicHash::Md5**.
- En **SHA-1** (wikipedia) : La méthode considérée comme la plus sûre des trois, et celle que je vous conseille d'utiliser. Pour l'utiliser, le second paramètre est **QCryptographicHash::Sha1**.

Essayez-donc de hacher en SHA-1 une chaine appelée **chaine** et définie un peu plus haut dans le code ! À vous de jouer !

Secret (cliquez pour afficher)

Code : C++

```
#include <QApplication>
#include <QWidget>
#include <QCryptographicHash>
#include <QByteArray>
#include <QString>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget w;

    QString chaine="Ceci sera haché.";
    QByteArray ba = QCryptographicHash::hash(chaine.toUtf8(),
QCryptographicHash::Sha1);


    w.show();
```

```
        return app.exec();
    }
```

Et voilà ! Vous avez haché votre chaine !

J'ai essayé d'afficher le QByteArray dans un QMessageBox. Ça m'affiche un truc bizarre, plein de caractères spéciaux. Ça ressemble à ça un hash ?

Non . C'est normal que ça affiche ça, car le retour de **hash** n'est PAS une QString ! Pour le convertir en QString, il faut utiliser la méthode **toHex()** :

Code : C++

```
QString hash = ba.toHex();
```

Là, vous obtiendrez une chaine qui ressemble à **0e6d2d873c7ec7ce703e48b009723c9b820f29bd**. Eh bien, ceci est votre hash !

Si on résume le code, on a donc :

Code : C++

```
#include <QApplication>
#include <QWidget>
#include <QCryptographicHash>
#include <QByteArray>
#include <QString>
#include <QMessageBox>


int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget w;

    QString chaine="Ceci sera haché."; // On crée la chaine à
hacher
    QByteArray ba = QCryptographicHash::hash(chaine.toUtf8(),
QCryptographicHash::Sha1); // on la hache

    QString out=ba.toHex(); // on convertit le hash en QString
    QMessageBox::information(&w, "test", out); // on l'affiche

    w.show();
    return app.exec();
}
```

Plus de sécurité ?

Comment ça, "plus de sécurité" ? Tu nous avais dit que c'est indécryptable ! Tu nous aurais menti ?

D'une certaine manière, non. En effet, c'est, à proprement parler, indéchiffrable. Mais... Il existe des correspondances entre les mots courants et leurs hachages MD5/SHA-1 ou autres. Évidemment, ces bases de données ne sont pas complètes ! Mais si votre hash est dedans... Pour cela, il a été mis au point une technique permettant ~~d'assaisonner les aliments~~ de rendre ces

méthodes quasi-inutiles, et de rendre plus difficiles les attaques par force brute (test de toutes les combinaisons) : le **salage** (wikipedia).

Cette technique est très simple : pour compliquer les données à hacher, on rajoute avant ou après (ou les deux) des données qui sont toujours les mêmes. Ces données sont appelées **sels**.

Ainsi, si on veut hacher la chaine "siteduzero" avec le sel "GRAINDESEL" (où GRAINDESEL est un define d'une chaîne de caractères, méthode que je vous recommande) avant, on mettra en argument "data" de la fonction **hash**
`QString(GRAINDESEL"siteduzero").toUtf8`

Il y a même encore mieux : hacher les sels eux-mêmes, de préférence avec un autre algorithme de hachage (par exemple, MD5 si vous utilisez du SHA-1 pour le hash final). Vous pouvez faire de même avec votre chaîne à chiffrer elle même :

Code : C++

```
#include <QApplication>
#include <QWidget>
#include <QCryptographicHash>
#include <QByteArray>
#include <QString>
#include <QMessageBox>

#define SEL_AVANT "D4Dqdz68$E" // une technique approuvée pour vos sels est de s'endormir sur le clavier.
#define SEL_APRES "7HHo£hh7YH" // Si ils ne veulent rien dire, c'est mieux.


int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget w;

    QString chaine="Ceci sera haché."; // On crée la chaine à hacher
QByteArray ba = QCryptographicHash::hash(QByteArray(QCryptographicHash::hash(QString(SEL_AVANT).toUtf8(), QCryptographicHash::Md5) + QCryptographicHash::hash(chaine.toUtf8(), QCryptographicHash::Md5) + QCryptographicHash::hash(QString(SEL_APRES).toUtf8(), QCryptographicHash::Md5)), QCryptographicHash::Sha1); // on la hache

    QString out=ba.toHex(); // on convertit le hash en QString
    QMessageBox::information(&w, "test", out); // on l'affiche

    w.show();
    return app.exec();
}
```

Nous sommes donc au final arrivés à un code pas si gros, mais très sécurisé ! Par contre, il faut savoir une chose : plus vous mettez de sécurités, sels, hachages de sels, etc., plus le temps de calcul est long ! Pour hacher vos sels, l'utilisation du MD4 est éventuellement possible, si vous trouvez que le temps de calcul est trop long (mais franchement, vous devriez même pas le voir passer, sauf si vous générez 100 000 hashs, et encore).
Eh bien voilà ! Vous êtes maintenant des pros du hachage avec Qt !

Les tables de hachage

QUOI ??? On en a pas fini avec les hashs depuis le dernier chapitre ?

Rassurez-vous, ici, on ne va pas parler de hachage mais de **tables de hachage** ! Il s'agit de tableaux qui, au lieu de contenir leurs valeurs à l'index 0, 1, 2, etc., contiennent leurs valeurs dans des index identifiés par **une chaine de caractères** (ou autre chose, on verra ça après) ! Eh oui ! Vous pourrez donc accéder aux données en faisant `tableau["id"]` au lieu de `tableau[1]` . Quel avantage ? Vous n'aurez plus besoin de stocker de valeurs pour retrouver vos données, dans le cas d'un tableau qui contient des données pas forcément prévisibles !

Théorie

Pour faire original, on va commencer par la théorie () !

Une table de hachage est un tableau dont l'index est représenté non pas par un nombre, comme d'habitude, mais par une chaîne de caractères (dans la plupart des cas). Ainsi, pour un tableau contenant dans l'ordre le prénom, le nom et l'âge d'une personne, on peut avoir le format suivant :

Tableau classique

ID	Contenu
0	Jean
1	Dupont
2	25

Table de hachage

Clé	Contenu
prenom	Jean
nom	Dupont
age	25

NB : Dans une table de hachage, on appelle l'ID "clé".

C'est cool tout ça, mais... Je vois franchement pas le rapport avec le hachage.

En fait, l'ordinateur ne peut faire que des index numériques (avec des nombres). Le principe de la table de hachage est donc simple : hacher la clé pour avoir un index numérique ! Ainsi, le tableau suivant serait plus complet (dans ce code, on considère qu'il existe une fonction hacher(QString), qui prend en paramètre une chaine à hacher) :

Table de hachage

Clé	ID	Contenu
prenom	hacher("prenom")	Jean
nom	hacher("nom")	Dupont
age	hacher("age")	25

Aucun rapport avec la sécurité, la cryptographie, et tout ça ?

Non ! Les tables de hachage n'ont rien à voir avec !

Avec Qt

Prêts à passer à la pratique ?

Qt gère les tables de hachage avec la classe QHash.

Pour commencer, voyons le prototype du constructeur :

Code : C++

```
QHash ()
```

Bon, ok, on pouvait pas faire plus simple...
Seulement, sa définition se fait un peu comme une QList (un exemple de QList ?) :

Code : C++

```
QList<QWidget*> liste; // crée une QList contenant des QWidget
```

Pour créer un QHash, il faut donc lui indiquer le type des valeurs qui seront stockées. Mais pas seulement ! Il faut aussi lui indiquer de quel type sera la clé ! Eh oui, on peut créer des tables de hachage dont les clés sont... n'importe quel type de variables, objets compris ! Les clés peuvent donc par exemple être des pointeurs sur des QPushButtons, ça ne posera aucun problème (eh oui, c'est ça que je vous cachais depuis tout à l'heure) !
Dans la plupart des cas, la clé reste une chaîne de caractères.

Pour créer une table de hachage, on aura donc un code ressemblant à celui-ci :

Code : C++

```
QHash<typeCle, typeValeur> table1; // clés de type typeCle, valeurs
de type typeValeur
QHash<QString, int> table2; // clés de type QString, valeurs de
type int
QHash<QString, QWidget*> table3; // clés de type QString, valeurs de
type pointeur sur QWidget
QHash<QPushButton*, int> table4; // clés de type pointeur sur
QPushButton, valeurs de type int
```

Ce code créera 4 tables de hachage (explications dans les commentaires).

J'imagine qu'ensuite, vous avez l'intention d'ajouter des valeurs, en supprimer, les lire, etc., n'est-ce pas ? Eh bien, il y a plusieurs manières de faire. On va commencer par... la première.

Méthode 1 : comme avec un tableau

La première méthode est de faire comme s'il s'agissait d'un simple tableau.

Cette méthode est déconseillée dans le cas où on lit une case qui peut ne pas exister, car la case inexistante est alors créée, ce qui prend de la mémoire pour rien. Je vous conseille donc de prendre l'habitude d'utiliser la 2nde méthode, même si connaître celle-ci peut être utile pour lire le code de quelqu'un d'autre par exemple.

Dans cette méthode, nous allons considérer qu'on a affaire à un tableau classique (mais extensible quand même, vous pouvez donc ajouter une valeur). Nous allons donc utiliser l'objet suivant :

Code : C++

```
QHash<QString, int> table;
```

Tiens, d'ailleurs, normalement, vous devriez être capables de me dire quel type de clés et quel type de valeurs attend cette table !

Solution :

Secret (cliquez pour afficher)

- Les clés seront de type QString
- Les valeurs seront de type int

Pour ajouter un item dans un tableau de ce genre, on ferait donc :

Code : C++

```
table["reponse"]=42; // la clé est "reponse", la valeur est "42".
```

Le code pour modifier la valeur associée à la clé "réponse" est exactement le même. Lorsqu'on modifie pour la première fois la valeur associée à une clé, on crée cet item.

Et pour supprimer un item ?

Ah ben ça... Avec cette méthode on peut pas  . Avec la méthode suivante, on verra ça !

Méthode 2 : avec des méthodes

Oui, je sais, le titre est un peu bizarre. Par "avec des méthodes", je veux dire "avec des fonctions contenues dans une classe".

Nous allons réutiliser notre QHash de tout à l'heure :

Code : C++

```
QHash<QString, int> table;
```

Il y a des méthodes pour faire ce qu'on a vu en 1, et même plus !

Ajouter un item

Nous utiliserons pour cela la méthode **insert** :

Code : C++

```
iterator QHash::insert ( const Key & key, const T & value )
```

Cette méthode prend en premier paramètre la nouvelle clé (key) et en second paramètre la nouvelle valeur (value). Notez que si l'item est déjà existant, la méthode remplacera l'ancienne valeur par la nouvelle. On utilisera donc la même méthode pour modifier cette valeur.

C'est quoi ce "iterator" que ça retourne ? Et c'est quoi ce "T" ?

Pour le "iterator", nous verrons ça après  !
Et pour ce qui est du "T", il s'agit du type que vous avez défini pour les valeurs de cette table de hachage, ici "int" (QHash<QString, **int**> table). Il s'agit d'un **template**.

Donc, si vous avez bien compris, pour ajouter à notre table de hachage "table" l'item de valeur "42" et de clé "reponse", le code sera...

Secret (cliquez pour afficher)

Code : C++

```
table.insert("reponse", 42);
```

Tester l'existence d'une clé

Pour vérifier si une clé existe dans une table de hachage, on utilise la méthode **contains**, qui prend en paramètre la clé recherchée, et renvoie **true** si la clé existe, sinon **false**.

Exemple :

Code : C++

```
if(table.contains("reponse"))
    // la clé "reponse" existe
else
    // la clé "reponse" n'existe pas
```

Récupérer le nombre d'items contenus dans la table de hachage

On utilise la méthode **count** :

Code : C++

```
int nombre=table.count();
//nombre = nombre d'items dans "table"
```

Récupérer une valeur

La méthode est **value** :

Code : C++

```
const T QHash::value ( const Key & key ) const
```

Cette méthode prend pour paramètre la clé (key) de la valeur qu'on cherche, et retourne la valeur. Elle s'utilise très simplement :

Code : C++

```
table.insert("reponse", 42); // pour avoir quelque chose à lire
int reponse=table.value("reponse");
// reponse = 42
```

Dans le cas où la clé n'existe pas, cette méthode renvoie la valeur par défaut du type de la valeur.

Gné ?

En fait, c'est pas compliqué du tout : ça renvoie la valeur par défaut, celle qui est donnée à une variable qui vient d'être créée. Seulement, cette valeur est différente selon le type de valeurs que prend la table de hachage. Par exemple, pour une QString, la valeur par défaut est une chaine vide, alors que pour un int, la valeur par défaut est 0.

On peut aussi donner la valeur de retour qu'on veut dans le cas où la clé n'existe pas avec la méthode surchargée :

Code : C++

```
const T QHash::value ( const Key & key, const T & defaultValue )
const
```

Exactement pareil, sauf que le 2ème paramètre devient la valeur de retour par défaut. Elle peut s'utiliser comme ça :

Code : C++

```
table.insert("reponse", 42); // pour avoir quelque chose à lire
int reponse=table.value("reponse", -1);
QLabel label;
label.setText(QString::number(reponse)); // affichera 42, ou -1 si
un problème est survenu.
```

Récupérer une clé

Je ne vais pas m'étendre sur le sujet, tout ce qui a été dit sur la récupération des valeurs est valable pour celle des clés, si ce n'est que la méthode est **key** et que son premier paramètre est une valeur au lieu d'une clé (en même temps, logique ).

Supprimer un item

La méthode utilisée est **delete** :

Code : C++

```
int QHash::remove ( const Key & key )
```

On l'utilise donc en lui passant en paramètre la clé de l'item à supprimer. Cette méthode renvoie le nombre d'items supprimés.

Mais... Elle renvoie donc 1 si l'item a été supprimé, ou 0 si il n'existe pas, donc ! Pourquoi ne pas avoir mis de booléen ?

Eh bien, parce que... **Une clé peut contenir plusieurs valeurs** ! Eh oui ! Aussi bizarre que ça puisse paraître, c'est possible !

Ajouter une valeur à une clé

Pour avoir plusieurs valeurs par clé, on utilise non pas *insert*, mais **insertMulti** :

Code : C++

```
iterator QHash::insertMulti ( const Key & key, const T & value )
```

Cette méthode marche exactement comme *insert*, mais au lieu de remplacer si la valeur est déjà existante, elle "ajoute" la valeur.

Récupérer DES valeurs

Dans ce cas, value() retourne quoi ?

value retourne la dernière valeur ajoutée. Pour obtenir une liste de toutes les valeurs pour une clé donnée, on utilise **values** (attention au 'S' à la fin) :

Code : C++

```
QList<T> QHash::values ( const Key & key ) const
```

Cette méthode marche comme *value*, mais elle retourne une QList contenant toutes les valeurs pour la clé *key*.

Méthode de lecture : les itérateurs

Vous vous souvenez de cet "iterator" en retour de **insert** ? Le moment est venu de vous expliquer ce que c'est. Un "iterator" (in english) se traduit par un **itérateur**.

Alors là, on est bien avancés ! Mes pensées se résument en un mot : gné ? 

Pour vous définir ça, je pense que Wikipedia sera très bien :

Citation

> Un itérateur est un objet qui permet de parcourir tous les éléments contenus dans un autre objet, le plus souvent un conteneur (liste, arbre, etc).

Comme il est dit dans la définition, c'est un objet à part entière. La classe qui gère ça est QHashIterator.

Voyons d'abord son constructeur :

Code : C++

```
QHashIterator::QHashIterator ( const QHash<Key, T> & hash )
```

Il prend donc en paramètre le QHash qu'il devra parcourir, mais devra, comme QHash, être déclaré avec un template :

Code : C++

```
QHashIterator<QString, int> iterator(table);
```

Le gros avantage est la possibilité de faire des tests sur les positions précédentes et suivantes :

Les tables de hachage ne sont pas ordonnées. Ainsi, on ne peut pas vraiment prévoir l'ordre des valeurs.

Clé et valeur actuelle

On peut avoir la clé et la valeur actuelle en utilisant **key()** et **value()** :

Code : C++

```
// prototype de key
const Key & QHashIterator::key () const

// prototype de value()
const T & QHashIterator::value () const
```

Tester l'existence des items suivants/précédents

Je ne mettrai qu'un prototype pour deux fonctions, étant donné que celles-ci marchent de la même manière, à la différence qu'une agit sur l'item précédent, et l'autre sur le suivant.

On peut vérifier l'existence d'un item suivant ou précédent respectivement avec **hasNext** et **hasPrevious** :

Code : C++

```
bool QHashIterator::hasNext () const
```

La méthode retourne donc un booléen (**true** si l'item existe, sinon **false**), et ne prend aucun paramètre.

Avancer/reculer la position actuelle

Pour déplacer la position actuelle de l'itérateur, on utilise **next** et **previous** :

Code : C++

```
Item QHashIterator::next ()
```

Donc, la méthode ne prend aucun paramètre, et retourne... un Item, qui contient la valeur et la clé de l'item sur lequel on se trouve maintenant. Ces variables s'obtiennent respectivement avec **value()** et **key()**.

Item précédent/suivant

Les méthodes **next()** et **previous()** retournent un Item contenant la valeur et la clé de la nouvelle valeur, après déplacement. Avec les méthodes **peekNext()** et **peekPrevious()**, on obtient la même valeur de retour, mais **sans se déplacer** !

Code : C++

```
Item QHashIterator::peekNext () const
```

Aller au début/à la fin

Pour aller au début (avant le premier item) ou à la fin (après le dernier item), on utilise les méthodes **toFront()** (début) ou **toBack()** (fin) :

Code : C++

```
void QHashIterator::toBack ()
```

Rechercher vers l'avant/vers l'arrière

On peut également faire une recherche partant d'avant ou après l'item actuel avec les méthodes **findNext()** et **findPrevious()** qui prennent en paramètre la valeur à rechercher :

Code : C++

```
bool QHashIterator::findNext ( const T & value )
```

La méthode retourne un booléen indiquant si la valeur *value* a été trouvée ou non.
L'utilisation est à peine différente selon le sens de la recherche :

- **findNext() : vers l'avant**
  La recherche commence à l'item suivant l'item actuel. Si l'item est trouvé, la position courante après la fonction est **juste après l'item trouvé**. Sinon, la position après la fonction est après le dernier item (comme après un **toBack()**).
- **findPrevious() : vers l'arrière**
  La recherche commence à l'item avant l'item actuel. Si l'item est trouvé, la position courante après la fonction est **juste avant l'item trouvé**. Sinon, la position après la fonction est avant le premier item (comme après un **toFront()**).

Récupérer toutes les valeurs : foreach()

Alors là, rien de plus simple : il s'agit d'une boucle spéciale qui est exécutée pour toutes les valeurs de la table, mais qui ne permet pas de voir la clé associée.
On l'utilise ainsi :

Code : C++

```
foreach(int valeur, table)
{
    quelqueChose(valeur); // "valeur" contient la valeur actuelle
}
```

Eh bien voilà, normalement (enfin je l'espère), vous maîtrisez les tables de hachage !
Je ne vois pas grand chose de plus à dire à ce sujet...

Par contre, je peux vous dire que le prochain chapitre est... un TP ! Prêts ? Alors cliquez sur "suivant" !

TP : le zLogin

Nous voici arrivés au TP ! Une occasion pour vous de vérifier si vous avez bien tout retenu, et si vous êtes capables de le mettre en pratique ! On va donc appliquer tout ce qu'on a vu depuis le début :

- Les ressources
- Le hachage
- Les tables de hachage
- Et accessoirement, l'utilisation du réseau

HEIN ?! Pourquoi du réseau, on en a jamais parlé dans ce tuto !

J'ai choisi le réseau pour différentes raisons. Premièrement, M@t' en parle dans son tuto sur le C++, que je considère comme un prérequis pour lire mon tuto. Et ensuite, franchement, j'ai trouvé aucune idée de TP qui aboutisse à un programme "utile" (avoir à la fin un programme qui ne vous sert à rien ne vous aurait pas énormément plu, non ? ) sans utiliser le réseau. Et puis la dernière raison est que ça vous fait réviser ! (Puis bon, avouez qu'une fois qu'on a lu le tuto de M@t' dessus, c'est pas bien compliqué !)

Prêts à vous y attaquer ? Suivez-moi !

Présentation du sujet

Dans ce TP, nous allons créer le **zLogin** ! C'est un programme-serveur auquel peuvent se connecter d'autres programmes via un réseau (loopback, local, internet).

Vue d'ensemble

Son rôle est simple : un programme lui envoie un paquet en TCP, contenant un login et un mot de passe haché en MD5 (nous ne coderons pas ce programme). Le zLogin (que nous allons créer) se chargera de comparer ce mot de passe à une liste de mots de passe contenus dans un fichier, et dira au programme si le login et le mdp sont bons.

Et... Le hachage, les tables de hachage, et les ressources, elles sont où dans tout ça ?

Premièrement, pour le hachage, vous DEVREZ re-hacher le mdp envoyé, avec salage et toutes les sécurités qu'on a vues. Vous ne pensiez pas que nous allions stocker un mot de passe haché seulement en MD5, non ? Par contre, pensez bien à tenir compte du fait qu'il est déjà haché en MD5 lors de sa réception !

Ensuite, pour les tables de hachage. Vous devrez, à l'ouverture du programme, prendre tous les couples login/mdp et les stocker dans une table de hachage (clé = pseudo, valeur = mdp haché).

Et pour les ressources, on va faire simple : l'icône sera dans un fichier de ressources, si vous arrivez à faire ça, vous arriverez à tout faire.

Schéma des paquets

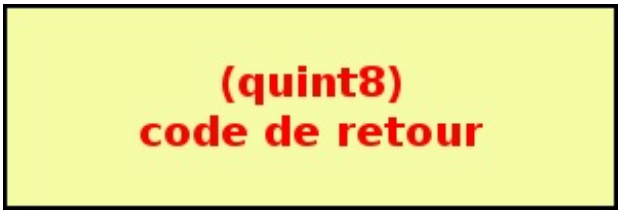
Vous êtes libres d'utiliser les architectures de paquets que vous voulez, mais si vous ne savez pas quoi prendre, je vous donne les structures que j'utilise :

Paquets entrants : pseudos/hashs mdp

J'utilise donc un paquet composé de 4 parties :

- Une variable **quint16** contenant la taille du pseudo,
- Une variable **QString** contenant de pseudo,
- Une variable **quint16** contenant la taille du hash du mdp,
- Une variable **QByteArray** contenant le hash du mdp.

paquets sortants : connexion acceptée/refusée



Cette fois, le paquet se compose d'un simple **quint8**. Celui-ci peut prendre différentes valeurs :

- 0 : Erreur indéterminée
- 1 : Connexion acceptée
- 2 : Mauvais login
- 3 : Mauvais mdp
- 4 : Erreur interne au zLogin
- 5 : Service indisponible

Fonctionnalités gérées par le zLogin

Voici une liste des fonctionnalités que doit avoir le zLogin. Bien sûr, rien ne vous empêche d'en rajouter !

Ne confondez pas "fonctionnalités" et "fonction [codée dans le programme]" ! Vous êtes libres de créer les fonctions que vous voulez pour implémenter les fonctionnalités données !

- Charger **dans une table de hachage** tous les logins/mdps au démarrage depuis un fichier (voir ce tuto d'Amnell et Shareman, impossible de se passer des fichiers dans un TP comme ça)
- Communiquer en réseau, et fonctionner comme programme serveur
- Pouvoir recevoir des données sur un port de votre choix (j'utiliserai le 31742)
- Traiter les données reçues en les comparant avec la table de hachage générée à l'ouverture du programme
- Hacher les mdps en utilisant des sels (eux mêmes hachés séparément), en utilisant les algorithmes de votre choix
- Renvoyer des données pour confirmer (ou non) la connexion
- Aucune fenêtre. Une icône du system tray (descendez un peu, jusqu'à croiser "system tray") sera la seule manifestation du zLogin (un petit tour dans la doc vous sera utile)
- Un menu sera accessible avec l'icône system tray, il contiendra :
  - Ajouter/supprimer des comptes : ouvre une boite de dialogue (voir la doc de QDialog, vous devrez créer une classe héritant de QDialog) permettant de voir les comptes créés, de les supprimer, et d'en ajouter. Quand on ajoute un compte, la liste des comptes doit être rechargée. Mais dans le cas où quelqu'un enverrait une requête à ce moment, **le programme ne doit pas fournir de fausse information** ! La "pause du service" doit être la plus courte possible.

    - Une option désactivée par défaut, qui, si activée, lors de chaque tentative de login, affiche une bulle d'information affichant le login de la personne, et le résultat sous forme de chaîne de caractères (et non de chiffre).
    - Une option "suspendre le programme" : si activée, aucune demande ne sera traitée, et le code 5 (service indisponible) sera renvoyé.
    - Quitter : Je ne pense pas avoir besoin de vous décrire à quoi cette action sert  . Pensez à faire un dialogue demandant confirmation, au lieu de quitter directement !
- L'icône du system tray sera chargée depuis un fichier de ressources

Dernières recommandations

Pour commencer, 2-3 choses qui peuvent vous être utiles :

- **Pack d'icônes :** Je vous ai préparé un petit pack d'icônes, libre à vous de les utiliser ou non. Téléchargez-le !
- **zLogin testeur :** J'ai également fait un programme client qui vous servira à tester votre zLogin, en se connectant à une adresse et un port (que vous choisissez) avec un pseudo et un mot de passe (que vous choisissez également). Téléchargez-le !
- **Une partie du code :** En codant le programme, j'ai remarqué qu'une grosse partie de celui-ci était composé uniquement de GUI, notamment au niveau de l'interface de gestion des comptes. Pour ceux qui le veulent seulement, et qui préfèrent se concentrer sur les nouveautés et ne pas réviser la GUI, je vous propose le code source de la gestion des comptes :

Secret (cliquez pour afficher)

J'ai utilisé pour ça deux classes : une qui gère le dialogue de gestion des comptes en lui-même, que j'ai appelé GereComptes, et une autre qui gère le dialogue d'édition/création de comptes (celui où on rentre le login/mdp du futur utilisateur), que j'ai appelée DialSetUser.
Je vais commencer par GereComptes :

gerecomptes.h (en général, mes noms de fichiers sont entièrement en minuscules par flemme de changer celui par défaut du QtCreator )

Code : C++

```
#ifndef GERECOMPTES_H
#define GERECOMPTES_H

// eh oui, il y en a pas mal
#include <QWidget>
#include <QTableView>
#include <QStandardItemModel>
#include <QHBoxLayout>
#include <QVBoxLayout>
#include <QPushButton>
#include <QIcon>
#include <QHash>
#include <QHashIterator>
#include <QCloseEvent>
#include <QList>
#include <QHeaderView>
#include <QMessageBox>
#include <QFile>
#include <QTextStream>
#include <QRegExp>

#include "dialsetuser.h"

class GereComptes : public QWidget
{
Q_OBJECT

public:
    GereComptes(QHash<QString, QString> authIn);

signals:
    void fini(); // émis à la fermeture du dialogue
```

```
    private slots:
        void slotAddUser();
        void slotSetUser();
        void slotDelUser();
        void slotOk();

    private:
        // methodes
        void majModele(); // appelé pour ajouter dans "modele"
    tout ce qui est contenu dans "authTable" (appelé dans le
    constructeur uniquement)
        void closeEvent(QCloseEvent *event); // Pour intercepter
    l'évènement de fermeture

        //attributs
        QHBoxLayout *l_principal;
        QTableView *tableau; // j'utilise une architecture MV
    (voir tuto de M@t')
        QStandardItemModel *modele;
        QVBoxLayout *l_boutons;
        QPushButton *b_addUser;
        QPushButton *b_setUser;
        QPushButton *b_delUser;
        QPushButton *b_ok;

        QHash<QString, QString> authTable;

        bool closing;
    };

    #endif // GERECOMPTES_H
```

Rien de bien compliqué, ça reste un .h  .

gerecomptes.cpp

Code : C++

```
    #include "gerecomptes.h"

    GereComptes::GereComptes(QHash<QString, QString> authIn)
    {
        setWindowTitle("zLogin - gestion des comptes");
        setWindowIcon(QIcon(":/icones/prog"));
        authTable=authIn;
        closing=false;

        // GUI
        l_principal=new QHBoxLayout;
        modele=new QStandardItemModel(0, 2);
        modele->setHeaderData(0, Qt::Horizontal, "Pseudo",
    Qt::DisplayRole);
        modele->setHeaderData(1, Qt::Horizontal, "Hash du MDP",
    Qt::DisplayRole);
        tableau=new QTableView;
        tableau->setModel(modele);
        tableau->setMinimumWidth(425);
        majModele();
        l_principal->addWidget(tableau);

        l_boutons=new QVBoxLayout;
        b_addUser=new QPushButton(QIcon(":/icones/addUser"),
    "Créer un compte");
        l_boutons->addWidget(b_addUser);
        b_setUser=new QPushButton(QIcon(":/icones/setUser"),
    "Modifier le compte");
```

```
    l_boutons->addWidget(b_setUser);
    b_delUser=new QPushButton(QIcon(":/icones/delUser"),
"Supprimer le compte");
    l_boutons->addWidget(b_delUser);
    b_ok=new QPushButton(QIcon(":/icones/accept"), "Ok");
    l_boutons->addWidget(b_ok);
    l_principal->addLayout(l_boutons);

    setLayout(l_principal);

    // connect
    connect(b_addUser, SIGNAL(clicked()), this,
SLOT(slotAddUser()));
    connect(b_setUser, SIGNAL(clicked()), this,
SLOT(slotSetUser()));
    connect(b_delUser, SIGNAL(clicked()), this,
SLOT(slotDelUser()));
    connect(b_ok, SIGNAL(clicked()), this, SLOT(slotOk()));
}

void GereComptes::majModele()
{
    // assez simple : on ajoute dans le modèle tous les
logins/hashs.
    QHashIterator<QString, QString> iterateur(authTable);

    while(iterateur.hasNext())
    {
        iterateur.next();
        QList<QStandardItem *> items;
        items << new QStandardItem(iterateur.key()) << new
QStandardItem(iterateur.value());
        modele->appendRow(items);
    }

    tableau->resizeColumnsToContents();
}

void GereComptes::slotAddUser()
{
    // On ajoutera le nouvel utilisateur directement dans le
fichier, qui devra être rechargé une fois le dialogue fermé

    QFile f_auth;
    QTextStream authStream;

    f_auth.setFileName("auth.zla");
    if(!f_auth.open(QIODevice::ReadWrite | QIODevice::Text))
    {
        QMessageBox::critical(0, "zLogin : erreur", "Le
programme \"zLogin\" a rencontré une erreur lors de
l'ouverture de la liste des logins/mdps du zLogin.\nErreur :
"+f_auth.errorString());
        close();
    }
    authStream.setDevice(&f_auth);


    User::Infos infos=DialSetUser::get(); // méthode statique

    if(infos.annule)
        return;

    QHashIterator<QString, QString> iterateur(authTable);

    while(iterateur.hasNext()) // vérif que le pseudo n'est
pas utilisé
    {
        iterateur.next();
        if(iterateur.key() == infos.pseudo)
```

```
            {
                QMessageBox::warning(this, "Erreur", "Le pseudo
choisi est déjà utilisé.");
                return;
            }
        }

        authStream.readAll(); // aller à la fin du fichier
        authStream << infos.pseudo+';'+infos.hashmdp+'\n';

        QList<QStandardItem *> items; // ajout du nouvel
utilisateur au modèle
        items << new QStandardItem(infos.pseudo) << new
QStandardItem(infos.hashmdp);
        modele->appendRow(items);
        tableau->resizeColumnsToContents();

        f_auth.close();
    }

    void GereComptes::slotSetUser()
    {
        QFile f_auth;
        QTextStream authStream;
        f_auth.setFileName("auth.zla");
        if(!f_auth.open(QIODevice::ReadWrite | QIODevice::Text))
        {
            QMessageBox::critical(0, "zLogin : erreur", "Le
programme \"zLogin\" a rencontré une erreur lors de
l'ouverture de la liste des logins/mdps du zLogin.\nErreur :
"+f_auth.errorString());
            close();
        }
        authStream.setDevice(&f_auth);


        QModelIndex setIndex=tableau->selectionModel()-
>currentIndex();
        if(!setIndex.isValid()) // index invalide, pas de
selection
        {
            QMessageBox::warning(this, "Aucune sélection", "Aucun
item n'est sélectionné.");
            return;
        }

        int setRow=setIndex.row(); // récupère l'ID de la ligne
sélectionnée
        QString basePseudo=modele->item(setRow, 0)-
>data(Qt::DisplayRole).toString(); // recup le pseudo actuel
        QString baseHash=modele->item(setRow, 1)-
>data(Qt::DisplayRole).toString(); // récup le hash actuel
        User::Infos infos=DialSetUser::get(basePseudo, baseHash);

        if(infos.annule) // si l'user a annulé
            return;

        if(infos.pseudo != basePseudo || infos.hashmdp !=
baseHash) // si il y a eu une modif
        {

            if(infos.pseudo!=basePseudo) //vérif que le pseudo
est libre
            {
                QHashIterator<QString, QString>
iterateur(authTable);

                while(iterateur.hasNext())
                {
                    iterateur.next();
```

```
                if(iterateur.key() == infos.pseudo)
                {
                    QMessageBox::warning(this, "Erreur", "Le
pseudo choisi est déjà utilisé.");
                    return;
                }
            }
        }

        // recherche de la ligne du fichier
        QString
setInfo=setIndex.data(Qt::DisplayRole).toString();
        bool isPseudo;
        if(setIndex.column() == 0)
            isPseudo=true;
        else if(setIndex.column() == 1)
            isPseudo=false;

        authStream.device()->seek(0);
        QString contenu;
        for(int i=0;!authStream.atEnd();i++)
        {
            QString ligne=authStream.readLine();
            QString info;
            if(isPseudo)
                info=ligne.section(';', 0, 0);
            else
                info=ligne.section(';', 1, 1);

            if(info != setInfo)
            {
                contenu+=ligne+'\n'; // on prend en mémoire
toutes les lignes sauf celle à modif
            }
        }

        contenu += infos.pseudo+';'+infos.hashmdp+'\n'; //
puis on ajoute à ça la ligne AVEC modifs

        f_auth.resize(0); // supprime le contenu du fichier
        authStream << contenu; // puis le réécrit

        modele->setItem(setRow, 0, new
QStandardItem(infos.pseudo)); // édite la ligne du modèle
        modele->setItem(setRow, 1, new
QStandardItem(infos.hashmdp));

        tableau->resizeColumnsToContents();
    }

    f_auth.close();
}

void GereComptes::slotDelUser()
{
    // idem slotSetUser() en grande partie
    // ====================================
    QFile f_auth;
    QTextStream authStream;
    f_auth.setFileName("auth.zla");
    if(!f_auth.open(QIODevice::ReadWrite | QIODevice::Text))
    {
        QMessageBox::critical(0, "zLogin : erreur", "Le
programme \"zLogin\" a rencontré une erreur lors de
l'ouverture de la liste des logins/mdps du zLogin.\nErreur :
"+f_auth.errorString());
        close();
    }
    authStream.setDevice(&f_auth);
```

```
    QModelIndex delIndex=tableau->selectionModel()->currentIndex();
    if(!delIndex.isValid()) // index invalide, pas de selection
    {
        QMessageBox::warning(this, "Aucune sélection", "Aucun item n'est sélectionné.");
        return;
    }

    if(QMessageBox::warning(this, "Êtes-vous sûr ?", "Êtes-vous sûr de vouloir supprimer ce compte définitivement ?",
                            QMessageBox::No | QMessageBox::Yes, QMessageBox::No) == QMessageBox::No)
    {
        return;
    }

    QString delInfo=delIndex.data(Qt::DisplayRole).toString();
    bool isPseudo;
    if(delIndex.column() == 0)
        isPseudo=true;
    else if(delIndex.column() == 1)
        isPseudo=false;

    authStream.device()->seek(0);
    QString contenu;
    for(int i=0;!authStream.atEnd();i++)
    {
        QString ligne=authStream.readLine();
        QString info;
        if(isPseudo)
            info=ligne.section(';', 0, 0);
        else
            info=ligne.section(';', 1, 1);

        if(info != delInfo)
        {
            contenu+=ligne+'\n';
        }
    }

    f_auth.resize(0);
    authStream << contenu;

    modele->removeRow(delIndex.row());
    tableau->resizeColumnsToContents();

    f_auth.close();
}

void GereComptes::slotOk()
{
    closing=true;
    emit fini();
    close();
}

void GereComptes::closeEvent(QCloseEvent *event)
{
    if(!closing) // si on est pas passé par slotOk()
        slotOk();
    else
        event->accept();
}
```

Pour les explications, elles sont dans la correction et dans quelques commentaires (évitez de lire la correction maintenant, vous risquez de lire la correction du reste) !

Passons à DialSetUser :

dialsetuser.h

Code : C++

```
#ifndef DIALSETUSER_H
#define DIALSETUSER_H

#include <QDialog>
#include <QVBoxLayout>
#include <QHBoxLayout>
#include <QFormLayout>
#include <QLineEdit>
#include <QPushButton>
#include <QString>
#include <QMessageBox>

#include "fonctions.h"

class User // oui, je suis flemmard. C'est plus court comme ça.
{
public:
    struct Infos // structure contenant toutes les infos renvoyées par le dialogue.
    {
        QString pseudo;
        QString hashmdp;
        bool annule;
    };
};

class DialSetUser : public QDialog
{
Q_OBJECT

public:
    DialSetUser(User::Infos *out_infos, QString in_pseudo=QString(), QString in_hash=QString()); // in_pseudo et in_hash sont des paramètres facultatifs. Si ils ont été donnés, ce sont les valeurs par défaut.
    static User::Infos get(QString pseudo=QString(), QString hash=QString()); // méthode statique permettant d'appeler le dialogue sans créer un objet.

private slots:
    void slotOk();
    void slotAnnule();

private:
    QVBoxLayout *l_principal;
    QFormLayout *l_form;
    QLineEdit *linePseudo;
    QLineEdit *linePass;
    QLineEdit *linePass2;
    QHBoxLayout *l_boutons;
    QPushButton *b_ok;
    QPushButton *b_annule;

    User::Infos *infos; // pointeur vers la structure passée en paramètre du constructeur
    QString defhash;
};
```

```
#endif // DIALSETUSER_H
```

dialsetuser.cpp

Code : C++

```
#include "dialsetuser.h"

DialSetUser::DialSetUser(User::Infos *out_infos, QString
in_pseudo, QString in_hash)
{
    QString passLabelEnd=""; // permet d'afficher un message
ou non en fonction des paramètres passés
    if(in_pseudo.isEmpty() && in_hash.isEmpty())
        setWindowTitle("Ajouter un utilisateur");
    else
    {
        passLabelEnd=" (laisser vide\npour aucun
changement)";
        setWindowTitle("Modifier un utilisateur");
    }

    defhash=in_hash; // hash par défaut

    setWindowIcon(QIcon(":/icones/prog"));

    infos=out_infos;

    infos->annule=true; // si fermé, considéré comme annuler

    // ===== GUI ======
    l_principal=new QVBoxLayout;
    l_form=new QFormLayout;
    linePseudo=new QLineEdit;
    linePseudo->setText(in_pseudo);
    l_form->addRow("Pseudo",linePseudo);
    linePass=new QLineEdit;
    linePass->setEchoMode(QLineEdit::Password);
    l_form->addRow("Mot de passe"+passLabelEnd, linePass);
    linePass2=new QLineEdit;
    linePass2->setEchoMode(QLineEdit::Password);
    l_form->addRow("Confirmez le mot de passe", linePass2);
    l_principal->addLayout(l_form);

    l_boutons=new QHBoxLayout;
    b_annule=new QPushButton(QIcon(":/icones/cancel"),
"Annuler");
    l_boutons->addWidget(b_annule);
    b_ok=new QPushButton(QIcon(":/icones/accept"), "Ok");
    b_ok->setDefault(true);
    l_boutons->addWidget(b_ok);
    l_principal->addLayout(l_boutons);

    setLayout(l_principal);

    // ===== connexions =====
    connect(b_annule, SIGNAL(clicked()), this,
SLOT(slotAnnule()));
    connect(b_ok, SIGNAL(clicked()), this, SLOT(slotOk()));
}

void DialSetUser::slotAnnule()
{
    infos->annule=true;
    close();
}
```

```
void DialSetUser::slotOk()
{
    if(linePseudo->text().isEmpty())
    {
        QMessageBox::warning(this, "Erreur", "Tous les champs
doivent être remplis !");
        return;
    }

    if(linePass->text() != linePass2->text())
    {
        QMessageBox::warning(this, "Erreur", "Les mots de
passe ne correspondent pas.");
        return;
    }

    if(linePseudo->text().contains(';'))
    {
        QMessageBox::warning(this, "Erreur", "Le pseudo
contient des caractères interdits.");
        return;
    }

    QString hashpass=linePass->text();
    if(hashpass.isEmpty())
    {
        if(defhash.isEmpty())
        {
            QMessageBox::warning(this, "Erreur", "Tous les
champs doivent être remplis !");
            return;
        }
        else
        {
            hashpass=defhash;
        }
    }
    else
        hashpass=Fonctions::hash(hashpass);

    infos->annule=false;
    infos->pseudo=linePseudo->text();
    infos->hashmdp=hashpass;
    close();
}

User::Infos DialSetUser::get(QString pseudo, QString hash)
{
    User::Infos retour; // on crée une structure à retourner,
    DialSetUser dial(&retour, pseudo, hash); // on crée un
dialogue
    dial.exec();
    return retour; // puis on retourne la structure
}
```

Et voilà, vous avez tout le code !

Ensuite, il faut pouvoir le faire marcher avec le code que vous allez faire ! Pour ça, il ne vous faut que trois choses :

- **Slot d'appel :** Vous devrez créer un slot qui créera un objet GereComptes, l'ouvrira (*show();*), puis le connectera à un autre slot avec son signal *fini()*. Seulement, pour que ça marche, la fenêtre doit être un attribut de votre classe principale ! À vous de gérer ça.
- **Slot de fin :** Vous devrez donc connecter le signal *fini()* de la fenêtre à ce slot, qui rechargera le fichier pour garder à jour la liste des logins/hashs.
- **Fonction de hachage :** J'ai choisi de la mettre dans une classe à part, *Fonctions*, contenue dans les fichiers *fonctions.h* et *fonctions.cpp*. La fonction est statique, et son prototype est :

Code : C++

```
QString Fonctions::hash(QString input, bool alreadyMd5 =
false);
```

Elle retourne une QString contenant le hash final, et prend en paramètre une QString contenant la chaîne à hacher, ainsi qu'un booléen facultatif (par défaut *false*) indiquant si *input* est déjà haché en MD5 ou pas. Ainsi, le retour doit être égal si on envoie en paramètre "sdz", false et si on envoie "sdz" haché en MD5, true.

Et maintenant, à vous de jouer ! Bonne programmation !

Correction

Alors ? Fini ?

Dans ce cas, attaquons la correction !

On va commencer soft, avec le main.cpp :

Code : C++

```
#include <QtGui/QApplication>
#include <QTextCodec>
#include <QTranslator>
#include <QLocale>
#include <QLibraryInfo>

#include "trayicon.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    QString locale = QLocale::system().name().section('_', 0, 0);
    QTranslator translator;
    translator.load(QString("qt_") + locale,
QLibraryInfo::location(QLibraryInfo::TranslationsPath));
    a.installTranslator(&translator);

    a.setQuitOnLastWindowClosed(false); // la fermeture de toutes
les fenêtres ne fermera pas le programme (utile pour un prog sans
fenêtres)

    TrayIcon t;
    t.show();
    return a.exec();
}
```

Rien de bien particulier, si ce n'est que j'utilise comme classe principale *TrayIcon*, une classe héritant de QSystemTrayIcon.

Bon, on va attaquer plus dur : TrayIcon !

TrayIcon

trayicon.h (mes noms de fichier sont en minuscules)

Code : C++

```
#ifndef TRAYICON_H
#define TRAYICON_H

// Eh oui, quand on utilise pas QtGui, ça donne ça !
#include <QSystemTrayIcon>
#include <QIcon>
#include <QMenu>
#include <QAction>
#include <QMessageBox>
#include <QHash>
#include <QByteArray>
#include <QFile>
#include <QTextStream>
#include <QCryptographicHash>
#include <QTcpServer>
#include <QTcpSocket>

#include "gerecomptes.h"
#include "fonctions.h"

class TrayIcon : public QSystemTrayIcon
{
Q_OBJECT

public:
    TrayIcon();
    void genereListePass(); // analyse le fichier contenant les
logins/hashs pour les mettre dans "passTable"

private slots:
    void slotGereComptes();
    void slotGereComptesDone(); // appelé quand la fenêtre de
gestion des comptes est fermée
    void slotDisplayAll();
    void slotSuspendre();
    void slotQuitter();

    // slots réseau
    void slotNewConnexion();
    void slotDonneesRecues();
    void slotDeconnexion();

private:
    // ========== METHODES ==========
    void envoyer(QTcpSocket *sock, quint8 reponse);

    // ========== ATTRIBUTS ==========
    // GUI
    QMenu *menu;
    QAction *actGereComptes;
    QAction *actDisplayAll;
    QAction *actSuspendre;
    QAction *actQuitter;

    GereComptes *fenComptes;

    // network
    QTcpServer *serv;
    quint16 size1; // stocke la taille de la 1ere partie des
paquets reçus
    QString data1; // 1ere part. paquets reçus
    quint16 size2; // idem size1 pour la 2nde
    QString data2; // idem data1 pour la 2nde
    QList<QTcpSocket *> listeClients;

    // pass
    QHash<QString, QString> passTable; // stocke les correspondances
logins/hashs

    // options
```

```
    bool displayAll; // si true, chaque connexion sera affichée
dans une infobulle
    bool suspendu; // si true, toute connexion renverra 5 (service
indisponible)
};

#endif // TRAYICON_H
```

Bon, lui je vous le donne d'un coup, c'est pas bien dur à comprendre. Vous voyez qu'on inclut *gerecomptes.h* et *fonctions.h*. gerecomptes.h contient la classe GereComptes créant l'interface graphique pour ajouter/modifier/supprimer des comptes, et fonctions.h contient la classe Fonctions, qui contient la méthode statique servant au hachage.

On va attaquer les méthodes une par une maintenant :

TrayIcon() (constructeur)

Code : C++

```
TrayIcon::TrayIcon()
{
    // inits
    setIcon(QIcon(":/icones/prog")); // définit l'icône
    displayAll=false; // par défaut, infobulles désactivées
    suspendu=false; // par défaut, service activé
    genereListePass();

    size1=0;
    data1="";
    size2=0;
    data2="";

    serv=new QTcpServer(this); // on démarre le serveur
    if(!serv->listen(QHostAddress::Any, 31742))
    {
        QMessageBox::critical(0, "zLogin : erreur fatale", "Le
programme zLogin ne peut établir de connexion. Il va donc
fermer.\nErreur : "+serv->errorString());
        exit(EXIT_FAILURE);
    }
    else
    {
        connect(serv, SIGNAL(newConnection()), this,
SLOT(slotNewConnexion()));
    }

    // Création du menu
    menu=new QMenu("Application");
    actGereComptes=menu->addAction("Gérer les comptes", this,
SLOT(slotGereComptes()));
    menu->addSeparator();
    actDisplayAll=menu->addAction("Activer les infobulles", this,
SLOT(slotDisplayAll()));
    actDisplayAll->setToolTip("Affiche une infobulle à chaque
tentative de connexion");
    actDisplayAll->setCheckable(true);
    actDisplayAll->setChecked(false);
    actSuspendre=menu->addAction("Suspendre le service", this,
SLOT(slotSuspendre()));
    actSuspendre->setToolTip("Refuse toute tentative de connexion,
tout en gardant le programme ouvert");
    actSuspendre->setCheckable(true);
    actSuspendre->setChecked(false);
    menu->addSeparator();
    actQuitter=menu->addAction("Quitter", this,
SLOT(slotQuitter()));
    setContextMenu(menu);
```

```
}
```

Encore une fois, rien de bien compliqué : on initialise le serveur (voir le tuto de M@t' si vous ne comprenez pas cette partie du code), puis on crée le menu de l'icône dans le system tray, en créant un menu comme pour une fenêtre, puis en l'attribuant à l'icône avec *setContextMenu(menu);*.

On va ensuite voir tout ce qui ne concerne pas le réseau, puis on s'attaquera à la gestion du réseau.

Non-réseau

genereListePass() (qui remplit la table de hachage en fonction du fichier "auth.zla")

Code : C++

```
void TrayIcon::genereListePass()
{
    QFile f_pass("auth.zla"); // zla = ZLogin Auth, fichier texte.
    if(!f_pass.open(QIODevice::ReadOnly | QIODevice::Text))
    {
        QMessageBox::critical(0, "zLogin : erreur critique", "Le
programme \"zLogin\" a rencontré une erreur critique lors de la mise
à jour de la liste des logins/mdps du zLogin. Le programme va donc
fermer.\nErreur : "+f_pass.errorString());
        exit(EXIT_FAILURE);
    }

    QTextStream stream(&f_pass);

    QHash<QString, QString> tempPassTable; // on crée une table
temporaire pour réduire le temps d'indisponiblité du service

    while(!stream.atEnd())
    {
        QString line=stream.readLine();
        QString pseudo=line.section(';', 0, 0);
        QString pass=line.section(';', 1, 1);
        tempPassTable.insert(pseudo, pass);
    }

    f_pass.close();

    suspendu=true; // On évite les erreurs si une requête est
effectuée à ce moment en désactivant le service
    passTable.empty();
    passTable=tempPassTable;
    suspendu=false; // Puis on le réactive
}
```

On ouvre donc le fichier auth.zla, qui contient les logins/hashs stockés ainsi :

Code : Autre

```
login;hash
autrelogin;autrehash
```

On boucle jusqu'à la fin du fichier en remplissant une table de hachage **temporaire**, puis on ferme le fichier. Utiliser une table de hachage temporaire permet de réduire le temps où le service sera indisponible.
Ensuite, on rend le service indisponible (en passant *suspendu* à true), on vide la table de hachage (la vraie cette fois), puis on

copie les données de la table de hachage temporaire dans la table de hachage, et enfin on réactive le service. Couper le service permet d'éviter des bugs dans le cas où une requête serait effectuée pile au moment où la table de hachage est vide.

Ensuite, les slots appelés quand on clique sur un élément du menu :
slotGereComptes() (appelé par "Gérer les comptes")

Code : C++

```
void TrayIcon::slotGereComptes()
{
    fenComptes=new GereComptes(passTable);
    fenComptes->show();
    connect(fenComptes, SIGNAL(fini()), this,
SLOT(slotGereComptesDone()));
}
```

Très court, on crée la fenêtre en lui passant la table de hachage qui contient les logins/hashs, on l'affiche, puis on connecte le signal personnalisé *fini()* au slot *slotGereComptesDone*.

slotGereComptesDone() (appelé quand la fenêtre de gestion des comptes est fermée)

Code : C++

```
void TrayIcon::slotGereComptesDone()
{
    genereListePass();
}
```

Encore plus court, ce slot se contente de mettre à jour, à partir du fichier que la fenêtre de gestion des comptes a peut-être modifié, la table de hachage qui contient les logins/hashs. J'ai utilisé un slot en lui-même au lieu de connecter directement *fini()* à *genereListePass()*, car si on a besoin ensuite de rajouter une action lors de la fermeture du dialogue, c'est plus simple .

slotDisplayAll() (appelé par "Activer les infobulles")

Code : C++

```
void TrayIcon::slotDisplayAll()
{
    displayAll=actDisplayAll->isChecked();

    if(displayAll)
        showMessage("Infobulles activées", "Les infobulles ont été
activées. Toute tentative de login sera signalée.",
QSystemTrayIcon::Information, 5000);
    else
        showMessage("Infobulles désactivées", "Les infobulles ont
été désactivées. Les tentatives de login ne seront plus signalées.",
QSystemTrayIcon::Information, 5000);
}
```

On modifie la valeur de *displayAll* en fonction de l'état d'*actDisplayAll* (l'élément du menu "Activer les infobulles"), puis on affiche que les infobulles ont été activées/désactivées.

slotSuspendre() (appelé par "Suspendre le service")

Code : C++

```
void TrayIcon::slotSuspendre()
{
    if(actSuspendre->isChecked()) // Si on demande de suspendre
    {
        if(QMessageBox::warning(0, "Êtes-vous sûr ?", "Voulez-vous
vraiment suspendre le service ? Toute tentative de connexion sera
refusée, mais le programme sera toujours ouvert.",
                                QMessageBox::No | QMessageBox::Yes,
QMessageBox::No) == QMessageBox::Yes)
        {
            suspendu=true;
            showMessage("Service suspendu", "Le service a bien été
suspendu. Toute tentative de login sera rejetée, et renverra le code
5 (service indisponible)", QSystemTrayIcon::Information, 5000);
        }
    }
    else
    {
        suspendu=false;
        showMessage("Service activé", "Le service a bien été activé.
Les tentatives de login seront maintenant traitées.",
QSystemTrayIcon::Information, 5000);
    }
}
```

Idem à *slotDisplayAll*, sauf qu'on modifie *suspendu*, et qu'on demande une confirmation pour le faire passer à true.

slotQuitter() (appelé par "Quitter")

Code : C++

```
void TrayIcon::slotQuitter()
{
    int ret = QMessageBox::warning(0, "Êtes-vous sûr ?", "Êtes-vous
sûr de vouloir quitter le zLogin ?",
                                   QMessageBox::No |
QMessageBox::Yes, QMessageBox::No);

    if(ret==QMessageBox::Yes)
        exit(EXIT_SUCCESS);
}
```

On demande une confirmation, puis on quitte, tout simplement.
On en a fini avec ce qui ne concerne pas le réseau !

Réseau

slotNewConnexion() (appelé lors d'une connexion au serveur)

Code : C++

```
void TrayIcon::slotNewConnexion()
{
    QTcpSocket *sock=serv->nextPendingConnection();
    listeClients << sock;
    connect(sock, SIGNAL(readyRead()), this,
SLOT(slotDonneesRecues()));
    connect(sock, SIGNAL(disconnected()), this,
SLOT(slotDeconnexion()));
}
```

Si vous avez le tuto de M@t' sur le réseau avec Qt en tête, c'est très simple : on récupère le client, on le stocke dans une liste, puis on connecte *readyRead()* et *disconnected* à des slots personnalisés.

Continuons avec...
envoyer() (appelé pour envoyer la réponse)

Code : C++

```
void TrayIcon::envoyer(QTcpSocket *sock, quint8 reponse)
{
    QByteArray paquet;
    QDataStream stream(&paquet, QIODevice::WriteOnly);

    stream << reponse;

    sock->write(paquet);
}
```

On crée un QByteArray stockant ce qu'on va envoyer, on y ajoute le paramètre *reponse*, puis on l'envoie.

slotDeconnexion() (appelé quand un client se déconnecte)

Code : C++

```
void TrayIcon::slotDeconnexion()
{
    QTcpSocket *sock=qobject_cast<QTcpSocket *>(sender());
    if(sock==0)
        return;

    sock->deleteLater();
    listeClients.removeOne(sock);
}
```

On identifie le client en question, puis on le supprime de la liste, et on le supprime tout court avec *deleteLater()*.

Attention, maintenant, on s'attaque au gros ! La plus grosse partie de la gestion réseau est bien évidemment celle qui gère la réception des paquets, puis ce qui sera retourné !
slotDonneesRecues() (gère la réception des paquets, puis le renvoi de la réponse)
Je vais faire exception à la règle, et vous donner le code petit bout par petit bout.
Déjà, on a deux cas possibles : le service est suspendu, ou ne l'est pas. Mais dans les deux cas, on doit savoir de qui vient la requête. Ce qui nous donne la base de code suivante :

Code : C++

```
void TrayIcon::slotDonneesRecues()
{
    // ==== Recup du socket
    QTcpSocket *sock = qobject_cast<QTcpSocket *>(sender());
    if(sock==0)
        return;

    QDataStream stream(sock);

    if(!suspendu) // si le service est actif
    {

    }
    else // service suspendu
```

```
        {

        }
    }
```

Commençons par le plus simple : le service est suspendu. Avant tout, il faut attendre qu'on aie reçu tous les sous-paquets. Pour cela, pas d'autre choix que de récupérer les deux tailles de parties ainsi :

Code : C++

```
    // ==== Recup du socket
        QTcpSocket *sock = qobject_cast<QTcpSocket *>(sender());
        if(sock==0)
            return;

        QDataStream stream(sock);

        if(!suspendu) // si le service est actif
        {

        }
        else // service suspendu
        {
            // on attend la fin de la transmission

            if(size1 == 0)
            {
                if(sock->bytesAvailable() < (int)sizeof(quint16))
                    return;
                stream >> size1;
            }
            if(data1 == "")
            {
                if(sock->bytesAvailable() < size1)
                    return;
                data1="ok";
            }
            if(size2 == 0)
            {
                if(sock->bytesAvailable() < (int)sizeof(quint16))
                    return;
                stream >> size2;
            }
            if(sock->bytesAvailable() < size2)
                return;

            // fin de la transmission
            envoyer(sock, 5);

            size1=0;
            data1="";
            size2=0;
            data2="";
        }
    }
```

On envoie ensuite dans tous les cas le code 5, soit "Service indisponible", puis on réinitialise les variables "size1", "data1", "size2" et "data2".

La partie "service actif" n'est pas bien différente, on y ajoute simplement le traitement des données reçues :

Code : C++

```
void TrayIcon::slotDonneesRecues()
{
    // ==== Recup du socket
    QTcpSocket *sock = qobject_cast<QTcpSocket *>(sender());
    if(sock==0)
        return;

    QDataStream stream(sock);

    if(!suspendu) // si le service est actif
    {
        // ==== Recup des donnees
        if(size1 == 0)
        {
            if(sock->bytesAvailable() < (int)sizeof(quint16))
                return;
            stream >> size1;
        }

        if(data1 == "")
        {
            if(sock->bytesAvailable() < size1)
                return;
            stream >> data1;
        }

        if(size2 == 0)
        {
            if(sock->bytesAvailable() < (int)sizeof(quint16))
                return;
            stream >> size2;
        }

        if(sock->bytesAvailable() < size2)
            return;

        stream >> data2;

        // ==== Traitement des donnees

        QString startInfo="Une tentative de connexion depuis l'IP
"+sock->peerAddress().toString()+" a été établie (pseudo :
"+data1+").\nRésultat : ";

        if(!passTable.contains(data1)) // existence du pseudo
        {
            if(displayAll) // infobulles activées
                showMessage("Tentative de connexion (échec)",
startInfo+"mauvais login.", QSystemTrayIcon::NoIcon, 5000);
            envoyer(sock, 2);

            size1=0;
            data1="";
            size2=0;
            data2="";
            return;
        }
        QString hashPass=passTable.value(data1);
        if(hashPass=="") // vérif que le hash ne soit pas vide
        {
            if(displayAll) // infobulles activées
                showMessage("Tentative de connexion (échec)",
startInfo+"erreur interne au zLogin (aucun mdp associé au login).",
QSystemTrayIcon::NoIcon, 5000);
            envoyer(sock, 4);

            size1=0;
            data1="";
            size2=0;
            data2="";
```

```
                return;
            }
            else if(hashPass != Fonctions::hash(data2, true)) // vérif
    mdp = mdp envoyé
            {
                if(displayAll) // infobulles activées
                    showMessage("Tentative de connexion (échec)",
    startInfo+"mauvais MDP.", QSystemTrayIcon::NoIcon, 5000);
                envoyer(sock, 3);

                size1=0;
                data1="";
                size2=0;
                data2="";
                return;
            }

            // Si on arrive là, login/mdp OK !
            if(displayAll) // infobulles activées
                showMessage("Tentative de connexion (succès)",
    startInfo+"succès.", QSystemTrayIcon::NoIcon, 5000);
            envoyer(sock, 1);

            size1=0;
            data1="";
            size2=0;
            data2="";
        }
        else // service suspendu
        {
            // on attend la fin de la transmission

            if(size1 == 0)
            {
                if(sock->bytesAvailable() < (int)sizeof(quint16))
                    return;
                stream >> size1;
            }
            if(data1 == "")
            {
                if(sock->bytesAvailable() < size1)
                    return;
                data1="ok";
            }
            if(size2 == 0)
            {
                if(sock->bytesAvailable() < (int)sizeof(quint16))
                    return;
                stream >> size2;
            }
            if(sock->bytesAvailable() < size2)
                return;

            // fin de la transmission
            envoyer(sock, 5);

            size1=0;
            data1="";
            size2=0;
            data2="";
        }
    }
```

On vérifie que le login existe, sinon, on renvoie 2 et on retourne. Puis on vérifie qu'un mot de passe est bien associé à ce login, sinon on renvoie 4 (erreur interne). Ensuite, on vérifie que le hash correspond bien à celui associé au login, sinon, on renvoie 3. À la fin, si tout est OK, on renvoie 1 (succès).

Et voilà, on en a fini avec TrayIcon !

Le code entier le trayicon.cpp :

Secret (cliquez pour afficher)

Code : C++

```
#include "trayicon.h"

TrayIcon::TrayIcon()
{
    // inits
    setIcon(QIcon(":/icones/prog"));
    displayAll=false; // par défaut, infobulles désactivées
    suspendu=false; // par défaut, service activé
    genereListePass();

    size1=0;
    data1="";
    size2=0;
    data2="";

    serv=new QTcpServer(this); // on démarre le serveur
    if(!serv->listen(QHostAddress::Any, 31742))
    {
        QMessageBox::critical(0, "zLogin : erreur fatale", "Le
programme zLogin ne peut établir de connexion. Il va donc
fermer.\nErreur : "+serv->errorString());
        exit(EXIT_FAILURE);
    }
    else
    {
        connect(serv, SIGNAL(newConnection()), this,
SLOT(slotNewConnexion()));
    }

    // Création du menu
    menu=new QMenu("Application");
    actGereComptes=menu->addAction("Gérer les comptes", this,
SLOT(slotGereComptes()));
    menu->addSeparator();
    actDisplayAll=menu->addAction("Activer les infobulles", this,
SLOT(slotDisplayAll()));
    actDisplayAll->setToolTip("Affiche une infobulle à chaque
tentative de connexion");
    actDisplayAll->setCheckable(true);
    actDisplayAll->setChecked(false);
    actSuspendre=menu->addAction("Suspendre le service", this,
SLOT(slotSuspendre()));
    actSuspendre->setToolTip("Refuse toute tentative de connexion,
tout en gardant le programme ouvert");
    actSuspendre->setCheckable(true);
    actSuspendre->setChecked(false);
    menu->addSeparator();
    actQuitter=menu->addAction("Quitter", this,
SLOT(slotQuitter()));
    setContextMenu(menu);
}

// ============================
// ====== NETWORK ============
// ============================

void TrayIcon::slotNewConnexion()
{
    QTcpSocket *sock=serv->nextPendingConnection();
    listeClients << sock;
    connect(sock, SIGNAL(readyRead()), this,
```

```
SLOT(slotDonneesRecues()));
    connect(sock, SIGNAL(disconnected()), this,
SLOT(slotDeconnexion()));
}

void TrayIcon::slotDonneesRecues()
{
    // ==== Recup du socket
    QTcpSocket *sock = qobject_cast<QTcpSocket *>(sender());
    if(sock==0)
        return;

    QDataStream stream(sock);

    if(!suspendu) // si le service est actif
    {
        // ==== Recup des donnees
        if(size1 == 0)
        {
            if(sock->bytesAvailable() < (int)sizeof(quint16))
                return;
            stream >> size1;
        }

        if(data1 == "")
        {
            if(sock->bytesAvailable() < size1)
                return;
            stream >> data1;
        }

        if(size2 == 0)
        {
            if(sock->bytesAvailable() < (int)sizeof(quint16))
                return;
            stream >> size2;
        }

        if(sock->bytesAvailable() < size2)
            return;

        stream >> data2;

        // ==== Traitement des donnees

        QString startInfo="Une tentative de connexion depuis l'IP
"+sock->peerAddress().toString()+" a été établie (pseudo :
"+data1+").\nRésultat : ";

        if(!passTable.contains(data1)) // existence du pseudo
        {
            if(displayAll) // infobulles activées
                showMessage("Tentative de connexion (échec)",
startInfo+"mauvais login.", QSystemTrayIcon::NoIcon, 5000);
            envoyer(sock, 2);

            size1=0;
            data1="";
            size2=0;
            data2="";
            return;
        }
        QString hashPass=passTable.value(data1);
        if(hashPass=="") // vérif que le hash ne soit pas vide
        {
            if(displayAll) // infobulles activées
                showMessage("Tentative de connexion (échec)",
startInfo+"erreur interne au zLogin (aucun mdp associé au
login).", QSystemTrayIcon::NoIcon, 5000);
            envoyer(sock, 4);
```

```
                size1=0;
                data1="";
                size2=0;
                data2="";
                return;
            }
            else if(hashPass != Fonctions::hash(data2, true)) // vérif
    mdp = mdp envoyé
            {
                if(displayAll) // infobulles activées
                    showMessage("Tentative de connexion (échec)",
    startInfo+"mauvais MDP.", QSystemTrayIcon::NoIcon, 5000);
                envoyer(sock, 3);

                size1=0;
                data1="";
                size2=0;
                data2="";
                return;
            }

            // Si on arrive là, login/mdp OK !
            if(displayAll) // infobulles activées
                showMessage("Tentative de connexion (succès)",
    startInfo+"succès.", QSystemTrayIcon::NoIcon, 5000);
            envoyer(sock, 1);

            size1=0;
            data1="";
            size2=0;
            data2="";
        }
        else // service suspendu
        {
            // on attend la fin de la transmission

            if(size1 == 0)
            {
                if(sock->bytesAvailable() < (int)sizeof(quint16))
                    return;
                stream >> size1;
            }
            if(data1 == "")
            {
                if(sock->bytesAvailable() < size1)
                    return;
                data1="ok";
            }
            if(size2 == 0)
            {
                if(sock->bytesAvailable() < (int)sizeof(quint16))
                    return;
                stream >> size2;
            }
            if(sock->bytesAvailable() < size2)
                return;

            // fin de la transmission
            envoyer(sock, 5);

            size1=0;
            data1="";
            size2=0;
            data2="";
        }
    }

    void TrayIcon::envoyer(QTcpSocket *sock, quint8 reponse)
    {
```

```
    QByteArray paquet;
    QDataStream stream(&paquet, QIODevice::WriteOnly);

    stream << reponse;

    sock->write(paquet);
}

void TrayIcon::slotDeconnexion()
{
    QTcpSocket *sock=qobject_cast<QTcpSocket *>(sender());
    if(sock==0)
        return;

    sock->deleteLater();
    listeClients.removeOne(sock);
}

// ============================
// ====== GUI & gestion pass ===
// ============================

void TrayIcon::genereListePass()
{
    QFile f_pass("auth.zla"); // zla = ZLogin Auth, fichier
texte.
    if(!f_pass.open(QIODevice::ReadOnly | QIODevice::Text))
    {
        QMessageBox::critical(0, "zLogin : erreur critique", "Le
programme \"zLogin\" a rencontré une erreur critique lors de la
mise à jour de la liste des logins/mdps du zLogin. Le programme va
donc fermer.\nErreur : "+f_pass.errorString());
        exit(EXIT_FAILURE);
    }

    QTextStream stream(&f_pass);

    QHash<QString, QString> tempPassTable; // on crée une table
temporaire pour réduire le temps d'indisponiblité du service

    while(!stream.atEnd())
    {
        QString line=stream.readLine();
        QString pseudo=line.section(';', 0, 0);
        QString pass=line.section(';', 1, 1);
        tempPassTable.insert(pseudo, pass);
    }

    f_pass.close();

    suspendu=true; // On évite les erreurs si une requête est
effectuée à ce moment en désactivant le service
    passTable.empty();
    passTable=tempPassTable;
    suspendu=false; // Puis on le réactive
}

void TrayIcon::slotGereComptes()
{
    fenComptes=new GereComptes(passTable);
    fenComptes->show();
    connect(fenComptes, SIGNAL(fini()), this,
SLOT(slotGereComptesDone()));
}

void TrayIcon::slotGereComptesDone()
{
    genereListePass();
}
```

```
void TrayIcon::slotDisplayAll()
{
    displayAll=actDisplayAll->isChecked();

    if(displayAll)
        showMessage("Infobulles activées", "Les infobulles ont été
activées. Toute tentative de login sera signalée.",
QSystemTrayIcon::Information, 5000);
    else
        showMessage("Infobulles désactivées", "Les infobulles ont
été désactivées. Les tentatives de login ne seront plus
signalées.", QSystemTrayIcon::Information, 5000);
}

void TrayIcon::slotSuspendre()
{
    if(actSuspendre->isChecked()) // Si on demande de suspendre
    {
        if(QMessageBox::warning(0, "Êtes-vous sûr ?", "Voulez-vous
vraiment suspendre le service ? Toute tentative de connexion sera
refusée, mais le programme sera toujours ouvert.",
                                QMessageBox::No |
QMessageBox::Yes, QMessageBox::No) == QMessageBox::Yes)
        {
            suspendu=true;
            showMessage("Service suspendu", "Le service a bien été
suspendu. Toute tentative de login sera rejetée, et renverra le
code 5 (service indisponible)", QSystemTrayIcon::Information,
5000);
        }
    }
    else
    {
        suspendu=false;
        showMessage("Service activé", "Le service a bien été
activé. Les tentatives de login seront maintenant traitées.",
QSystemTrayIcon::Information, 5000);
    }
}

void TrayIcon::slotQuitter()
{
    int ret = QMessageBox::warning(0, "Êtes-vous sûr ?", "Êtes-
vous sûr de vouloir quitter le zLogin ?",
                                   QMessageBox::No |
QMessageBox::Yes, QMessageBox::No);

    if(ret==QMessageBox::Yes)
        exit(EXIT_SUCCESS);
}
```

GereComptes

Cette classe est l'interface graphique servant à gérer les comptes.
On va attaquer avec gerecomptes.h :

gerecomptes.h

Code : C++

```
#ifndef GERECOMPTES_H
#define GERECOMPTES_H

// eh oui, il y en a pas mal
```

```
#include <QWidget>
#include <QTableView>
#include <QStandardItemModel>
#include <QHBoxLayout>
#include <QVBoxLayout>
#include <QPushButton>
#include <QIcon>
#include <QHash>
#include <QHashIterator>
#include <QCloseEvent>
#include <QList>
#include <QHeaderView>
#include <QMessageBox>
#include <QFile>
#include <QTextStream>
#include <QRegExp>

#include "dialsetuser.h"

class GereComptes : public QWidget
{
Q_OBJECT

public:
    GereComptes(QHash<QString, QString> authIn);

signals:
    void fini(); // émis à la fermeture du dialogue

private slots:
    void slotAddUser();
    void slotSetUser();
    void slotDelUser();
    void slotOk();

private:
    // methodes
    void majModele(); // appelé pour ajouter dans "modele" tout ce
qui est contenu dans "authTable" (appelé dans le constructeur
uniquement)
    void closeEvent(QCloseEvent *event); // Pour intercepter
l'évènement de fermeture

    //attributs
    QHBoxLayout *l_principal;
    QTableView *tableau; // j'utilise une architecture MV (voir
tuto de M@t')
    QStandardItemModel *modele;
    QVBoxLayout *l_boutons;
    QPushButton *b_addUser;
    QPushButton *b_setUser;
    QPushButton *b_delUser;
    QPushButton *b_ok;

    QHash<QString, QString> authTable;

    bool closing;
};

#endif // GERECOMPTES_H
```

Rien à expliquer ici je pense, si ce n'est que j'utilise une architecture MV (une MVC simplifiée, tuto de M@t' ici), comme en témoignent *QTableView *tableau;* et *QStandardItemModel *modele;*.

Attaquons-nous maintenant à l'implémentation de ces fonctions :

GereComptes() (constructeur)

Code : C++

```
GereComptes::GereComptes(QHash<QString, QString> authIn)
{
    setWindowTitle("zLogin - gestion des comptes");
    setWindowIcon(QIcon(":/icones/prog"));
    authTable=authIn;
    closing=false;

    // GUI
    l_principal=new QHBoxLayout;
    modele=new QStandardItemModel(0, 2);
    modele->setHeaderData(0, Qt::Horizontal, "Pseudo",
Qt::DisplayRole);
    modele->setHeaderData(1, Qt::Horizontal, "Hash du MDP",
Qt::DisplayRole);
    tableau=new QTableView;
    tableau->setModel(modele);
    tableau->setMinimumWidth(425);
    majModele();
    l_principal->addWidget(tableau);

    l_boutons=new QVBoxLayout;
    b_addUser=new QPushButton(QIcon(":/icones/addUser"), "Créer un
compte");
    l_boutons->addWidget(b_addUser);
    b_setUser=new QPushButton(QIcon(":/icones/setUser"), "Modifier
le compte");
    l_boutons->addWidget(b_setUser);
    b_delUser=new QPushButton(QIcon(":/icones/delUser"), "Supprimer
le compte");
    l_boutons->addWidget(b_delUser);
    b_ok=new QPushButton(QIcon(":/icones/accept"), "Ok");
    l_boutons->addWidget(b_ok);
    l_principal->addLayout(l_boutons);

    setLayout(l_principal);

    // connect
    connect(b_addUser, SIGNAL(clicked()), this,
SLOT(slotAddUser()));
    connect(b_setUser, SIGNAL(clicked()), this,
SLOT(slotSetUser()));
    connect(b_delUser, SIGNAL(clicked()), this,
SLOT(slotDelUser()));
    connect(b_ok, SIGNAL(clicked()), this, SLOT(slotOk()));
}
```

On crée la GUI, on crée *modele* et on l'associe à *tableau*, puis on appelle *majModele()*, qui crée, à partir de la table de hachage reçue en paramètre du constructeur, le modèle.

majModele() (appelé depuis le constructeur)

Code : C++

```
void GereComptes::majModele()
{
    // assez simple : on ajoute dans le modèle tous les
logins/hashs.
    QHashIterator<QString, QString> iterateur(authTable);

    while(iterateur.hasNext())
    {
        iterateur.next();
```

```
        QList<QStandardItem *> items;
        items << new QStandardItem(iterateur.key()) << new
QStandardItem(iterateur.value());
        modele->appendRow(items);
    }

    tableau->resizeColumnsToContents();
}
```

On crée un itérateur utilisant la table de hachage reçue en paramètre du constructeur, puis on le parcourt en ajoutant à chaque fois au modèle la clé et la valeur, soit le login et le hash. À la fin, *tableau->resizeColumnsToContents();* sert à ajuster les colonnes du tableau aux valeurs contenues.

slotAddUser() (appelé par un clic sur "Créer un compte")

Code : C++

```
void GereComptes::slotAddUser()
{
    // On ajoutera le nouvel utilisateur directement dans le
fichier, qui devra être rechargé une fois le dialogue fermé

    QFile f_auth;
    QTextStream authStream;

    f_auth.setFileName("auth.zla");
    if(!f_auth.open(QIODevice::ReadWrite | QIODevice::Text))
    {
        QMessageBox::critical(0, "zLogin : erreur", "Le programme
\"zLogin\" a rencontré une erreur lors de l'ouverture de la liste
des logins/mdps du zLogin.\nErreur : "+f_auth.errorString());
        close();
    }
    authStream.setDevice(&f_auth);


    User::Infos infos=DialSetUser::get(); // méthode statique

    if(infos.annule)
        return;

    QHashIterator<QString, QString> iterateur(authTable);

    while(iterateur.hasNext()) // vérif que le pseudo n'est pas
utilisé
    {
        iterateur.next();
        if(iterateur.key() == infos.pseudo)
        {
            QMessageBox::warning(this, "Erreur", "Le pseudo choisi
est déjà utilisé.");
            return;
        }
    }

    authStream.readAll(); // aller à la fin du fichier
    authStream << infos.pseudo+';'+infos.hashmdp+'\n';

    QList<QStandardItem *> items; // ajout du nouvel utilisateur au
modèle
    items << new QStandardItem(infos.pseudo) << new
QStandardItem(infos.hashmdp);
    modele->appendRow(items);
    tableau->resizeColumnsToContents();

    f_auth.close();
```

```
    }
```

On ouvre d'abord le fichier *auth.zla*, puis on ouvre un dialogue demandant à l'utilisateur le login et le mot de passe du nouvel utilisateur. Le dialogue se charge de hacher le MdP. On vérifie que l'utilisateur n'a pas annulé, puis on parcourt la table pour vérifier que le pseudo n'est pas déjà présent. On se place à la fin du fichier en le lisant entièrement, puis on y ajoute la ligne formée ainsi : login;hash\n (\n étant un retour à la ligne). Ensuite, on ajoute le nouvel utilisateur au modèle, puis on ferme le fichier.

slotDelUser() (appelé par un clic sur "Supprimer le compte")

Code : C++

```
void GereComptes::slotDelUser()
{
    QFile f_auth;
    QTextStream authStream;
    f_auth.setFileName("auth.zla");
    if(!f_auth.open(QIODevice::ReadWrite | QIODevice::Text))
    {
        QMessageBox::critical(0, "zLogin : erreur", "Le programme
\"zLogin\" a rencontré une erreur lors de l'ouverture de la liste
des logins/mdps du zLogin.\nErreur : "+f_auth.errorString());
        close();
    }
    authStream.setDevice(&f_auth);


    QModelIndex delIndex=tableau->selectionModel()->currentIndex();
    if(!delIndex.isValid()) // index invalide, pas de selection
    {
        QMessageBox::warning(this, "Aucune sélection", "Aucun item
n'est sélectionné.");
        return;
    }

    if(QMessageBox::warning(this, "Êtes-vous sûr ?", "Êtes-vous sûr
de vouloir supprimer ce compte définitivement ?",
                            QMessageBox::No | QMessageBox::Yes,
QMessageBox::No) == QMessageBox::No)
    {
        return;
    }

    QString delInfo=delIndex.data(Qt::DisplayRole).toString();
    bool isPseudo;
    if(delIndex.column() == 0)
        isPseudo=true;
    else if(delIndex.column() == 1)
        isPseudo=false;

    authStream.device()->seek(0);
    QString contenu;
    for(int i=0;!authStream.atEnd();i++)
    {
        QString ligne=authStream.readLine();
        QString info;
        if(isPseudo)
            info=ligne.section(';', 0, 0);
        else
            info=ligne.section(';', 1, 1);

        if(info != delInfo)
        {
            contenu+=ligne+'\n';
        }
    }
```

```
    f_auth.resize(0);
    authStream << contenu;

    modele->removeRow(delIndex.row());
    tableau->resizeColumnsToContents();

    f_auth.close();
}
```

Pareil, on ouvre le fichier *auth.zla*, puis on récupère la sélection pour savoir quelle ligne supprimer. Si l'index est invalide, il n'y a donc pas de sélection, on affiche une erreur et on retourne. On demande confirmation, et on prend en mémoire toutes les lignes sauf celle à supprimer. Avec *f_auth.resize(0)*, on supprime tout le contenu du fichier, puis on réinscrit dedans tout ce qu'on a gardé en mémoire. On supprime la ligne du modèle, puis on ferme le fichier.

slotSetUser() (appelé par un clic sur "Modifier le compte")

Code : C++

```
void GereComptes::slotSetUser()
{
    QFile f_auth;
    QTextStream authStream;
    f_auth.setFileName("auth.zla");
    if(!f_auth.open(QIODevice::ReadWrite | QIODevice::Text))
    {
        QMessageBox::critical(0, "zLogin : erreur", "Le programme
\"zLogin\" a rencontré une erreur lors de l'ouverture de la liste
des logins/mdps du zLogin.\nErreur : "+f_auth.errorString());
        close();
    }
    authStream.setDevice(&f_auth);


    QModelIndex setIndex=tableau->selectionModel()->currentIndex();
    if(!setIndex.isValid()) // index invalide, pas de selection
    {
        QMessageBox::warning(this, "Aucune sélection", "Aucun item
n'est sélectionné.");
        return;
    }

    int setRow=setIndex.row(); // récupère l'ID de la ligne
sélectionnée
    QString basePseudo=modele->item(setRow, 0)-
>data(Qt::DisplayRole).toString(); // recup le pseudo actuel
    QString baseHash=modele->item(setRow, 1)-
>data(Qt::DisplayRole).toString(); // récup le hash actuel
    User::Infos infos=DialSetUser::get(basePseudo, baseHash);

    if(infos.annule) // si l'user a annulé
        return;

    if(infos.pseudo != basePseudo || infos.hashmdp != baseHash) //
si il y a eu une modif
    {

        if(infos.pseudo!=basePseudo) //vérif que le pseudo est
libre
        {
            QHashIterator<QString, QString> iterateur(authTable);

            while(iterateur.hasNext())
            {
                iterateur.next();
                if(iterateur.key() == infos.pseudo)
```

```
                {
                    QMessageBox::warning(this, "Erreur", "Le pseudo
choisi est déjà utilisé.");
                    return;
                }
            }
        }

        // recherche de la ligne du fichier
        QString setInfo=setIndex.data(Qt::DisplayRole).toString();
        bool isPseudo;
        if(setIndex.column() == 0)
            isPseudo=true;
        else if(setIndex.column() == 1)
            isPseudo=false;

        authStream.device()->seek(0);
        QString contenu;
        for(int i=0;!authStream.atEnd();i++)
        {
            QString ligne=authStream.readLine();
            QString info;
            if(isPseudo)
                info=ligne.section(';', 0, 0);
            else
                info=ligne.section(';', 1, 1);

            if(info != setInfo)
            {
                contenu+=ligne+'\n'; // on prend en mémoire toutes
les lignes sauf celle à modif
            }
        }

        contenu += infos.pseudo+';'+infos.hashmdp+'\n'; // puis on
ajoute à ça la ligne AVEC modifs

        f_auth.resize(0); // supprime le contenu du fichier
        authStream << contenu; // puis le réécrit

        modele->setItem(setRow, 0, new QStandardItem(infos.pseudo));
// édite la ligne du modèle
        modele->setItem(setRow, 1, new
QStandardItem(infos.hashmdp));

        tableau->resizeColumnsToContents();
    }

    f_auth.close();
}
```

Ce slot est un simple mélange de *slotAddUser()* et de *slotDelUser*, je n'expliquerai donc rien ici, en croisant les deux explications vous devriez assez bien comprendre !

slotOk() (appelé par un clic sur "Ok")

Code : C++

```
void GereComptes::slotOk()
{
    closing=true;
    emit fini();
    close();
}
```

On définit *closing* à true, on émet *fini()*, puis on ferme la fenêtre.

closeEvent() (appelé par close(), ou un clic sur la croix)

Code : C++

```
void GereComptes::closeEvent(QCloseEvent *event)
{
    if(!closing) // si on est pas passé par slotOk()
        slotOk();
    else
        event->accept();
}
```

Pour que *fini()* soit émit si on clique sur la croix, on vérifie si *closing* vaut true (seulement vrai après un appel de *slotOk()*, si oui, on accepte la fermeture, sinon, on appelle *slotOk()*.

On en a également fini avec GereComptes !
Le code complet :

Secret (cliquez pour afficher)

Code : C++

```
#include "gerecomptes.h"

GereComptes::GereComptes(QHash<QString, QString> authIn)
{
    setWindowTitle("zLogin - gestion des comptes");
    setWindowIcon(QIcon(":/icones/prog"));
    authTable=authIn;
    closing=false;

    // GUI
    l_principal=new QHBoxLayout;
    modele=new QStandardItemModel(0, 2);
    modele->setHeaderData(0, Qt::Horizontal, "Pseudo",
Qt::DisplayRole);
    modele->setHeaderData(1, Qt::Horizontal, "Hash du MDP",
Qt::DisplayRole);
    tableau=new QTableView;
    tableau->setModel(modele);
    tableau->setMinimumWidth(425);
    majModele();
    l_principal->addWidget(tableau);

    l_boutons=new QVBoxLayout;
    b_addUser=new QPushButton(QIcon(":/icones/addUser"), "Créer un
compte");
    l_boutons->addWidget(b_addUser);
    b_setUser=new QPushButton(QIcon(":/icones/setUser"), "Modifier
le compte");
    l_boutons->addWidget(b_setUser);
    b_delUser=new QPushButton(QIcon(":/icones/delUser"),
"Supprimer le compte");
    l_boutons->addWidget(b_delUser);
    b_ok=new QPushButton(QIcon(":/icones/accept"), "Ok");
    l_boutons->addWidget(b_ok);
    l_principal->addLayout(l_boutons);

    setLayout(l_principal);

    // connect
    connect(b_addUser, SIGNAL(clicked()), this,
```

```
SLOT(slotAddUser()));
    connect(b_setUser, SIGNAL(clicked()), this,
SLOT(slotSetUser()));
    connect(b_delUser, SIGNAL(clicked()), this,
SLOT(slotDelUser()));
    connect(b_ok, SIGNAL(clicked()), this, SLOT(slotOk()));
}

void GereComptes::majModele()
{
    // assez simple : on ajoute dans le modèle tous les
logins/hashs.
    QHashIterator<QString, QString> iterateur(authTable);

    while(iterateur.hasNext())
    {
        iterateur.next();
        QList<QStandardItem *> items;
        items << new QStandardItem(iterateur.key()) << new
QStandardItem(iterateur.value());
        modele->appendRow(items);
    }

    tableau->resizeColumnsToContents();
}

void GereComptes::slotAddUser()
{
    // On ajoutera le nouvel utilisateur directement dans le
fichier, qui devra être rechargé une fois le dialogue fermé

    QFile f_auth;
    QTextStream authStream;

    f_auth.setFileName("auth.zla");
    if(!f_auth.open(QIODevice::ReadWrite | QIODevice::Text))
    {
        QMessageBox::critical(0, "zLogin : erreur", "Le programme
\"zLogin\" a rencontré une erreur lors de l'ouverture de la liste
des logins/mdps du zLogin.\nErreur : "+f_auth.errorString());
        close();
    }
    authStream.setDevice(&f_auth);


    User::Infos infos=DialSetUser::get(); // méthode statique

    if(infos.annule)
        return;

    QHashIterator<QString, QString> iterateur(authTable);

    while(iterateur.hasNext()) // vérif que le pseudo n'est pas
utilisé
    {
        iterateur.next();
        if(iterateur.key() == infos.pseudo)
        {
            QMessageBox::warning(this, "Erreur", "Le pseudo choisi
est déjà utilisé.");
            return;
        }
    }

    authStream.readAll(); // aller à la fin du fichier
    authStream << infos.pseudo+';'+infos.hashmdp+'\n';

    QList<QStandardItem *> items; // ajout du nouvel utilisateur
au modèle
    items << new QStandardItem(infos.pseudo) << new
```

```
QStandardItem(infos.hashmdp);
    modele->appendRow(items);
    tableau->resizeColumnsToContents();

    f_auth.close();
}

void GereComptes::slotSetUser()
{
    QFile f_auth;
    QTextStream authStream;
    f_auth.setFileName("auth.zla");
    if(!f_auth.open(QIODevice::ReadWrite | QIODevice::Text))
    {
        QMessageBox::critical(0, "zLogin : erreur", "Le programme
\"zLogin\" a rencontré une erreur lors de l'ouverture de la liste
des logins/mdps du zLogin.\nErreur : "+f_auth.errorString());
        close();
    }
    authStream.setDevice(&f_auth);


    QModelIndex setIndex=tableau->selectionModel()-
>currentIndex();
    if(!setIndex.isValid()) // index invalide, pas de selection
    {
        QMessageBox::warning(this, "Aucune sélection", "Aucun item
n'est sélectionné.");
        return;
    }

    int setRow=setIndex.row(); // récupère l'ID de la ligne
sélectionnée
    QString basePseudo=modele->item(setRow, 0)-
>data(Qt::DisplayRole).toString(); // recup le pseudo actuel
    QString baseHash=modele->item(setRow, 1)-
>data(Qt::DisplayRole).toString(); // récup le hash actuel
    User::Infos infos=DialSetUser::get(basePseudo, baseHash);

    if(infos.annule) // si l'user a annulé
        return;

    if(infos.pseudo != basePseudo || infos.hashmdp != baseHash) //
si il y a eu une modif
    {

        if(infos.pseudo!=basePseudo) //vérif que le pseudo est
libre
        {
            QHashIterator<QString, QString> iterateur(authTable);

            while(iterateur.hasNext())
            {
                iterateur.next();
                if(iterateur.key() == infos.pseudo)
                {
                    QMessageBox::warning(this, "Erreur", "Le
pseudo choisi est déjà utilisé.");
                    return;
                }
            }
        }

        // recherche de la ligne du fichier
        QString setInfo=setIndex.data(Qt::DisplayRole).toString();
        bool isPseudo;
        if(setIndex.column() == 0)
            isPseudo=true;
        else if(setIndex.column() == 1)
            isPseudo=false;
```

```
            authStream.device()->seek(0);
            QString contenu;
            for(int i=0;!authStream.atEnd();i++)
            {
                QString ligne=authStream.readLine();
                QString info;
                if(isPseudo)
                    info=ligne.section(';', 0, 0);
                else
                    info=ligne.section(';', 1, 1);

                if(info != setInfo)
                {
                    contenu+=ligne+'\n'; // on prend en mémoire
toutes les lignes sauf celle à modif
                }
            }

            contenu += infos.pseudo+';'+infos.hashmdp+'\n'; // puis on
ajoute à ça la ligne AVEC modifs

            f_auth.resize(0); // supprime le contenu du fichier
            authStream << contenu; // puis le réécrit

            modele->setItem(setRow, 0, new
QStandardItem(infos.pseudo)); // édite la ligne du modèle
            modele->setItem(setRow, 1, new
QStandardItem(infos.hashmdp));

            tableau->resizeColumnsToContents();
        }

        f_auth.close();
    }

    void GereComptes::slotDelUser()
    {
        QFile f_auth;
        QTextStream authStream;
        f_auth.setFileName("auth.zla");
        if(!f_auth.open(QIODevice::ReadWrite | QIODevice::Text))
        {
            QMessageBox::critical(0, "zLogin : erreur", "Le programme
\"zLogin\" a rencontré une erreur lors de l'ouverture de la liste
des logins/mdps du zLogin.\nErreur : "+f_auth.errorString());
            close();
        }
        authStream.setDevice(&f_auth);


        QModelIndex delIndex=tableau->selectionModel()-
>currentIndex();
        if(!delIndex.isValid()) // index invalide, pas de selection
        {
            QMessageBox::warning(this, "Aucune sélection", "Aucun item
n'est sélectionné.");
            return;
        }

        if(QMessageBox::warning(this, "Êtes-vous sûr ?", "Êtes-vous
sûr de vouloir supprimer ce compte définitivement ?",
                                QMessageBox::No | QMessageBox::Yes,
QMessageBox::No) == QMessageBox::No)
        {
            return;
        }

        QString delInfo=delIndex.data(Qt::DisplayRole).toString();
        bool isPseudo;
```

```
        if(delIndex.column() == 0)
            isPseudo=true;
        else if(delIndex.column() == 1)
            isPseudo=false;

        authStream.device()->seek(0);
        QString contenu;
        for(int i=0;!authStream.atEnd();i++)
        {
            QString ligne=authStream.readLine();
            QString info;
            if(isPseudo)
                info=ligne.section(';', 0, 0);
            else
                info=ligne.section(';', 1, 1);

            if(info != delInfo)
            {
                contenu+=ligne+'\n';
            }
        }

        f_auth.resize(0);
        authStream << contenu;

        modele->removeRow(delIndex.row());
        tableau->resizeColumnsToContents();

        f_auth.close();
    }

    void GereComptes::slotOk()
    {
        closing=true;
        emit fini();
        close();
    }

    void GereComptes::closeEvent(QCloseEvent *event)
    {
        if(!closing) // si on est pas passé par slotOk()
            slotOk();
        else
            event->accept();
    }
```

DialSetUser

Assez simple celui-là, ça va aller vite !

Commençons par **dialsetuser.h**

Code : C++

```
    #ifndef DIALSETUSER_H
    #define DIALSETUSER_H

    #include <QDialog>
    #include <QVBoxLayout>
    #include <QHBoxLayout>
    #include <QFormLayout>
    #include <QLineEdit>
    #include <QPushButton>
    #include <QString>
```

```
#include <QMessageBox>

#include "fonctions.h"

class User // oui, je suis flemmard. C'est plus court comme ça.
{
public:
    struct Infos // structure contenant toutes les infos renvoyées
par le dialogue.
    {
        QString pseudo;
        QString hashmdp;
        bool annule;
    };
};

class DialSetUser : public QDialog
{
Q_OBJECT

public:
    DialSetUser(User::Infos *out_infos, QString in_pseudo=QString(),
QString in_hash=QString()); // in_pseudo et in_hash sont des
paramètres facultatifs. Si ils ont été donnés, ce sont les valeurs
par défaut.
    static User::Infos get(QString pseudo=QString(), QString
hash=QString()); // méthode statique permettant d'appeler le
dialogue sans créer un objet.

private slots:
    void slotOk();
    void slotAnnule();

private:
    QVBoxLayout *l_principal;
    QFormLayout *l_form;
    QLineEdit *linePseudo;
    QLineEdit *linePass;
    QLineEdit *linePass2;
    QHBoxLayout *l_boutons;
    QPushButton *b_ok;
    QPushButton *b_annule;

    User::Infos *infos; // pointeur vers la structure passée en
paramètre du constructeur
    QString defhash;
};

#endif // DIALSETUSER_H
```

Seul truc un peu bizarre : je crée ici deux classes, dont une quasi-vide. En fait, la classe User ne contient qu'une déclaration structure, qui servira à retourner les informations entrées dans le dialogue. Par flemme de réécrire *DialSetUser::Infos* partout (oui, je confirme ce qu'on dit sur les programmeurs et la flemme ), j'ai créé une classe appelée User, comme ça, j'aurai juste besoin de marquer User::Infos. Sinon, *DialSetUser* est une classe héritant de *QDialog*.

On inclut *fonctions.h*, car on aura besoin de hacher le mot de passe. La QString *defhash* contient ici le hash passé en paramètre du constructeur, permettant de donner une valeur par défaut si on appelle le dialogue pour éditer un utilisateur.

DialSetUser() (constructeur)

Code : C++

```
DialSetUser::DialSetUser(User::Infos *out_infos, QString in_pseudo,
QString in_hash)
{
    QString passLabelEnd=""; // permet d'afficher un message ou non
```

```
    en fonction des paramètres passés
        if(in_pseudo.isEmpty() && in_hash.isEmpty())
            setWindowTitle("Ajouter un utilisateur");
        else
        {
            passLabelEnd=" (laisser vide\npour aucun changement)";
            setWindowTitle("Modifier un utilisateur");
        }

        defhash=in_hash; // hash par défaut

        setWindowIcon(QIcon(":/icones/prog"));

        infos=out_infos;

        infos->annule=true; // si fermé, considéré comme annuler

        // ===== GUI ======
        l_principal=new QVBoxLayout;
        l_form=new QFormLayout;
        linePseudo=new QLineEdit;
        linePseudo->setText(in_pseudo);
        l_form->addRow("Pseudo",linePseudo);
        linePass=new QLineEdit;
        linePass->setEchoMode(QLineEdit::Password);
        l_form->addRow("Mot de passe"+passLabelEnd, linePass);
        linePass2=new QLineEdit;
        linePass2->setEchoMode(QLineEdit::Password);
        l_form->addRow("Confirmez le mot de passe", linePass2);
        l_principal->addLayout(l_form);

        l_boutons=new QHBoxLayout;
        b_annule=new QPushButton(QIcon(":/icones/cancel"), "Annuler");
        l_boutons->addWidget(b_annule);
        b_ok=new QPushButton(QIcon(":/icones/accept"), "Ok");
        b_ok->setDefault(true);
        l_boutons->addWidget(b_ok);
        l_principal->addLayout(l_boutons);

        setLayout(l_principal);

        // ===== connexions =====
        connect(b_annule, SIGNAL(clicked()), this, SLOT(slotAnnule()));
        connect(b_ok, SIGNAL(clicked()), this, SLOT(slotOk()));
    }
```

On crée une QString qui sera affichée à la fin du texte associé au champ de MdP, ce qui permet de dire, dans le cas où on a passé un hash en paramètre, que le mot de passe est facultatif.
Ensuite, rien de spécial, on fait pointer *infos* sur *out_infos*, soit le paramètre qu'on a reçu, ce qui permet de retourner les infos saisies, puis on définit la GUI (en définissant comme valeur par défaut de *linePseudo in_pseudo*).

slotAnnule() (appelé par un clic sur "Annuler")

Code : C++

```
    void DialSetUser::slotAnnule()
    {
        infos->annule=true;
        close();
    }
```

On définit *annuler* (booléen contenu dans *infos*, la structure de retour) à *true*, puis on ferme le dialogue.

slotOk() (appelé par un clic sur "Ok")

Code : C++

```
void DialSetUser::slotOk()
{
    if(linePseudo->text().isEmpty())
    {
        QMessageBox::warning(this, "Erreur", "Tous les champs
doivent être remplis !");
        return;
    }

    if(linePass->text() != linePass2->text())
    {
        QMessageBox::warning(this, "Erreur", "Les mots de passe ne
correspondent pas.");
        return;
    }

    if(linePseudo->text().contains(';'))
    {
        QMessageBox::warning(this, "Erreur", "Le pseudo contient des
caractères interdits.");
        return;
    }

    QString hashpass=linePass->text();
    if(hashpass.isEmpty())
    {
        if(defhash.isEmpty())
        {
            QMessageBox::warning(this, "Erreur", "Tous les champs
doivent être remplis !");
            return;
        }
        else
        {
            hashpass=defhash;
        }
    }
    else
        hashpass=Fonctions::hash(hashpass);

    infos->annule=false;
    infos->pseudo=linePseudo->text();
    infos->hashmdp=hashpass;
    close();
}
```

On vérifie que la ligne du pseudo n'est pas vide, que les deux mdp sont identiques, que le pseudo ne contient pas de ';' (ça ferait pas bon ménage dans auth.zla, mais pour le mot de passe c'est pas grave, vu qu'il sera haché), et que le mot de passe, si aucun par défaut n'est défini, ne soit pas vide.
Puis on hache le mdp, on définit les valeurs de *infos* avec ce qu'il faut y mettre, puis on ferme le dialogue.

get() (appelé depuis l'extérieur)

Code : C++

```
User::Infos DialSetUser::get(QString pseudo, QString hash)
{
    User::Infos retour; // on crée une structure à retourner,
    DialSetUser dial(&retour, pseudo, hash); // on crée un dialogue
    dial.exec();
    return retour; // puis on retourne la structure
```

```
    }
```

C'est quoi cette méthode bizarre qui sert à rien ?

C'est une méthode que je crée tout le temps dans mes dialogues, qui m'évite de devoir créer un objet, l'exécuter, etc.
En fait, cette méthode statique se charge de créer un objet, de l'exécuter, et de retourner tout simplement les valeurs de retour.
Comme ça, quand j'ai besoin d'ouvrir ce dialogue, il me suffit de faire

Code : C++

```
User::Infos infos=DialSetUser::get();
```

Sinon, on devrait faire :

Code : C++

```
User::Infos infos;
DialSetUser dialogue(&infos);
dialogue.exec();
```

Avouez que c'est un peu plus long !

Eh bien voilà, on en a fini avec notre DialSetUser !
Code complet :

Secret (cliquez pour afficher)

Code : C++

```
#include "dialsetuser.h"

DialSetUser::DialSetUser(User::Infos *out_infos, QString
in_pseudo, QString in_hash)
{
    QString passLabelEnd=""; // permet d'afficher un message ou
non en fonction des paramètres passés
    if(in_pseudo.isEmpty() && in_hash.isEmpty())
        setWindowTitle("Ajouter un utilisateur");
    else
    {
        passLabelEnd=" (laisser vide\npour aucun changement)";
        setWindowTitle("Modifier un utilisateur");
    }

    defhash=in_hash; // hash par défaut

    setWindowIcon(QIcon(":/icones/prog"));

    infos=out_infos;

    infos->annule=true; // si fermé, considéré comme annuler

    // ===== GUI ======
    l_principal=new QVBoxLayout;
    l_form=new QFormLayout;
    linePseudo=new QLineEdit;
    linePseudo->setText(in_pseudo);
```

```
    l_form->addRow("Pseudo",linePseudo);
    linePass=new QLineEdit;
    linePass->setEchoMode(QLineEdit::Password);
    l_form->addRow("Mot de passe"+passLabelEnd, linePass);
    linePass2=new QLineEdit;
    linePass2->setEchoMode(QLineEdit::Password);
    l_form->addRow("Confirmez le mot de passe", linePass2);
    l_principal->addLayout(l_form);

    l_boutons=new QHBoxLayout;
    b_annule=new QPushButton(QIcon(":/icones/cancel"), "Annuler");
    l_boutons->addWidget(b_annule);
    b_ok=new QPushButton(QIcon(":/icones/accept"), "Ok");
    b_ok->setDefault(true);
    l_boutons->addWidget(b_ok);
    l_principal->addLayout(l_boutons);

    setLayout(l_principal);

    // ===== connexions =====
    connect(b_annule, SIGNAL(clicked()), this,
SLOT(slotAnnule()));
    connect(b_ok, SIGNAL(clicked()), this, SLOT(slotOk()));
}

void DialSetUser::slotAnnule()
{
    infos->annule=true;
    close();
}

void DialSetUser::slotOk()
{
    if(linePseudo->text().isEmpty())
    {
        QMessageBox::warning(this, "Erreur", "Tous les champs
doivent être remplis !");
        return;
    }

    if(linePass->text() != linePass2->text())
    {
        QMessageBox::warning(this, "Erreur", "Les mots de passe ne
correspondent pas.");
        return;
    }

    if(linePseudo->text().contains(';'))
    {
        QMessageBox::warning(this, "Erreur", "Le pseudo contient
des caractères interdits.");
        return;
    }

    QString hashpass=linePass->text();
    if(hashpass.isEmpty())
    {
        if(defhash.isEmpty())
        {
            QMessageBox::warning(this, "Erreur", "Tous les champs
doivent être remplis !");
            return;
        }
        else
        {
            hashpass=defhash;
        }
    }
    else
        hashpass=Fonctions::hash(hashpass);
```

```
        infos->annule=false;
        infos->pseudo=linePseudo->text();
        infos->hashmdp=hashpass;
        close();
    }

    User::Infos DialSetUser::get(QString pseudo, QString hash)
    {
        User::Infos retour; // on crée une structure à retourner,
        DialSetUser dial(&retour, pseudo, hash); // on crée un
    dialogue
        dial.exec();
        return retour; // puis on retourne la structure
    }
```

Fonctions

Cette fois, on attaque la dernière classe, très courte en plus !

Allez, c'est parti pour le **fonctions.h** :

Code : C++

```
#ifndef FONCTIONS_H
#define FONCTIONS_H

#include <QString>
#include <QCryptographicHash>
#include <QByteArray>

#define SEL_BEFORE "*D9)'Jj89K"
#define SEL_AFTER "^,iU078Jng"

class Fonctions
{
public:
    Fonctions();
    static QString hash(QString input, bool alreadyMd5=false);
};

#endif // FONCTIONS_H
```

Pas bien complexe, hein ?
En fait, cette classe n'est pas faite pour être instanciée, mais pour contenir des méthodes statiques.

Pourquoi ne pas faire simplement des fonctions, sans classe ?

J'y ai pensé. Mais ainsi, on regroupe toutes les fonctions dans un "ensemble" (même si il n'y en a qu'une, si le programme évolue, il peut y en avoir plus).
Je fais également deux defines, pour les sels.

fichier.cpp est tellement court que je vais vous le donner d'un seul coup :

Code : C++

```
#include "fonctions.h"
```

```
Fonctions::Fonctions()
{
}

QString Fonctions::hash(QString input, bool alreadyMd5)
{
    if(!alreadyMd5)
    {
        QByteArray ba_input=QCryptographicHash::hash(input.toUtf8(),
QCryptographicHash::Md5);
        input=ba_input.toHex();
    }
    QByteArray
ba_hash=QCryptographicHash::hash(QCryptographicHash::hash(QString(SEL_BEFORE).to
QCryptographicHash::Md4) +
                                                    input.toUtf8() +

QCryptographicHash::hash(QString(SEL_AFTER).toUtf8(), QCryptographicHash::Md4),
                                                    QCryptographicHash::Sha1); // ha
d'input, avec sels.
    return ba_hash.toHex();
}
```

Le constructeur est vide, puisque la classe ne sera pas instanciée.
La méthode statique *hash()* contient tout ce qu'on a vu dans le chapitre sur le hachage. Elle effectue les actions suivantes :

1. Si l'input n'est pas déjà haché en MD5, on le hache.
2. On hache avec des sels : SEL_BEFORE est haché en MD4, puis ajouté au MdP (déjà haché en MD5), et le tout est ajouté à SEL_AFTER, lui aussi haché en MD4. Puis, le tout est haché en SHA-1.
3. On convertit ça en chaîne de caractères avec *toHex()*
4. On retourne la chaîne obtenue

Eh bien, je vous annonce que **le zLogin est terminé !!!**

En réalisant ce TP, je l'ai pas trouvé très complexe, mais long. Si vous l'avez réussi, félicitations ! Maintenant, vous n'avez plus qu'à créer des programmes qui l'utilisent .

J'espère que ce TP vous a plu, que vous l'avez réussi et qu'il ne vous a pas paru trop long ou dur (le premier qui pense à Mozinor a gagné).

Si vous l'avez réussi, c'est que vous maîtrisez maintenant le hachage, les tables de hachage et les ressources Qt !

Et voilà, c'est fini !

Mais ce tuto n'est pas fermé ! Ainsi, si je pense à une autre fonctionnalité de Qt n'utilisant pas la GUI, j'ajouterai sûrement un chapitre à ce tuto. Je ne peux pas dire à l'avance si le cours sera continué ou non, et sur combien de chapitres !

Sinon, j'espère que vous avez compris sans problème ce cours, et que vous l'avez trouvé bien !

Bonne programmation à tous !
@+, Tobast.