Programmer MySQL avec Visual C++ 6.0

TUTORIEL

PAR

Jacques Abada

Version 1.0
du 22 Octobre 2001

Document entièrement réalisé avec
StarOffice 5.2 sous Windows
Fichier PDF réalisé à l'aide de Acrobat 5

Copyright © 2001, Jacques Abada

Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
*A copy of the license is included in the section entitled "**GNU Free Documentation licence**" in the file licence.pdf.*

La copie, la distribution et/ou la modification de ce document
est permise selon les termes de la GNU Free Documentation licence, Version 1.1
ou ultérieure publiée par la Free Software Foundation;
Une copie de cette licence (en anglais) est incluse dans le fichier licence.pdf.

L'auteur reste le seul propriétaire du texte et du code source.

StarOffice est une marque déposée de Sun Microsystems
Acrobat est une marque déposée de Adobe Corp.
Windows est une marque déposée de Microsoft Corp.

Staroffice est disponible en téléchargement gratuit @:
http://www.sun.com/staroffice/

Acrobat est payant et propriétaire.

Ce document est principalement disponible sur mon site (http://www.arcanthea.com). Mais également sur le site http://www.mysql.com dans la section des contributions.

Je remercie Paul Dubois d'avoir accépté de revoir la version anglaise de ce tutoriel.

Table des matières

Introduction....4
I – Configuration requise....5
II - Configuration de Visual C++ pour MySQL....6
III - Création du squelette d'application....7
IV - Création des boites de dialogue de notre application....8
V – Codage de la boite de dialogue de connexion....11
VI - Codage de la connexion....14
VII – Affichage des données dans la ListView....20
VIII – Codage de la boite Infos....25
Conclusion....27

Introduction

Ce tutoriel a pour but d'explorer une partie de l'API de MySQL au travers d'une application écrite en C++ sous Windows, à l'aide de Visual C++. Je n'ai pas fait d'effort particulier de portabilité, puisque j'ai utilisé l'architecture de classes standard de Visual C++, les MFC, pour prendre en charge toute la logique sous-jacente des contrôles avec lesquels j'ai interfacé les fonctions de MySQL. En fait, le programme exemple qui accompagne ce tutoriel montre comment utiliser les fonctions de l'API de MySQL avec une architecture de classes C++. Les programmeurs maîtrisant un outil de développement C++ pourront aisément reprendre le code propre à MySQL et l'interfacer à leur architecture de classes de prédilection.

Dans un programme, il y a l'interface utilisateur et le code *utile*. Le code utilisé pour l'interface utilisateur est *dépendant*, en ce sens qu'il varie d'un système d'exploitation à l'autre et d'un compilateur ou langage de programmation à l'autre. Le code utile est commun à tous les programmes. Une fonction MySQL telle que *mysql_query()* possède les mêmes caractéristiques, du point de vue d'un programmeur, si elle est appelée par un programme C, C++, Delphi, que ce soit sous Windows ou Unix.

Le programme que je propose comme illustration de ce tutoriel est une application de type boite de dialogue qui permet de se connecter à un serveur MySQL (local ou résidant sur un serveur distant), de sélectionner une base de données de ce serveur (sous réserve qu'on en ai le droit), et d'afficher, soit les données contenues dans les tables, soit la structure des tables. Enfin, une boite de dialogue supplémentaire affiche quelques informations telles que version du serveur, processus, version du client, etc... Il démontre avec quelle facilité il est possible d'écrire une application Windows capable de gérer des données se trouvant sur un serveur Linux. MySQL offre une API transparente à ce niveau, puisque l'on n'a pas à se soucier de savoir de quelle manière la connexion s'opère, la fonction *mysql_real_connect()* fait tout le travail de bas niveau pour nous.

Le texte qui suit détaille toutes les étapes de la création de ce programme.

Le lecteur doit avoir un niveau acceptable en C++, et bien connaître l'environnement Visual C++ 6.0; il doit aussi connaître MySQL.

Sur MySQL, je renvoie le lecteur à l'excellent ouvrage de Paul Dubois[1].

Sur Visual C++, voici quelques références qui peuvent s'avérer utiles[2]. Il n'existe pas à ma connaissance d'ouvrages traitant de programmation MySQL avec Visual C++. Tous les aspects de la programmation des bases de données avec VC++ concernent les technologies propriétaires Microsoft.

Sur internet, deux sites[3] en anglais sont consacrés à Visual C++, et il existe aussi un site http://www.commentcamarche.com qui fourni des cours sur le C, C++, Linux, MySQL, PHP, etc., en français.

1 Paul Dubois: ***MySQL*** publié aux éditions New Riders. Edition française Campus Press (ISBN: 2-7440-0882-6). Cet ouvrage explique non seulement l'utilisation de MySQL mais présente la programmation avec divers langages (PHP, C, Perl).

2 Mike Blaszczak: ***Professional MFC with Visual C++ 6***, Wrox Press
Collectif: ***MFC Programming with Visual C++ 6***, collection Unleashed chez SAMS
Kruglinski & autres: ***Atelier Visual C++ 6.0*** Microsoft Press

3 Http://www.codeproject.com
http://www.codeguru.com

I – Configuration requise

Compilateur Visual C++ 6.0

Bibliothèques clientes et fichiers inclus de MySQL 3.23.4x

Station de travail sous Windows (NT ou 2000)

L'application exemple peut attaquer des bases MySQL qui résident soit sur la machine locale, soit sur un serveur à travers le réseau. Selon le cas, il faudra installer MySQL différemment sous Windows.

1. L'application cliente est sur une machine et les données sur un serveur:
   Dans ce cas on a besoin des bibliothèques et des fichiers inclus (répertoires lib\opt et \include de l'installation de MySQL. Le service mysqld-nt n'est pas nécessaire.
   Pour travailler, on n'a besoin que des fichiers *.h situés dans le répertoire \mysql\include et des *.lib et *.dll résirant dans le répertoire \mysql\lib\opt d'une installation Windows.
2. L'application cliente et les bases sont sur la même machine Windows: en ce cas il faut lancer le service mysqld-nt. La connexion se fait généralement sur le *localhost*.

L'idéal est bien sûr de lancer l'application sous Windows, et d'accéder à des bases de données situées sur un serveur Linux ou un autre système de type UNIX.

II - Configuration de Visual C++ pour MySQL

Il nous faut indiquer à Visual C++ où trouver les fichiers d'entête ainsi que la bibliothèque de liaison (*.lib). On ouvre la boite de dialogue du menu **Tools|Options** et on choisit l'onglet **Directories**. Là, on sélectionne successivement le chemin des « include files » (Fig. 1)

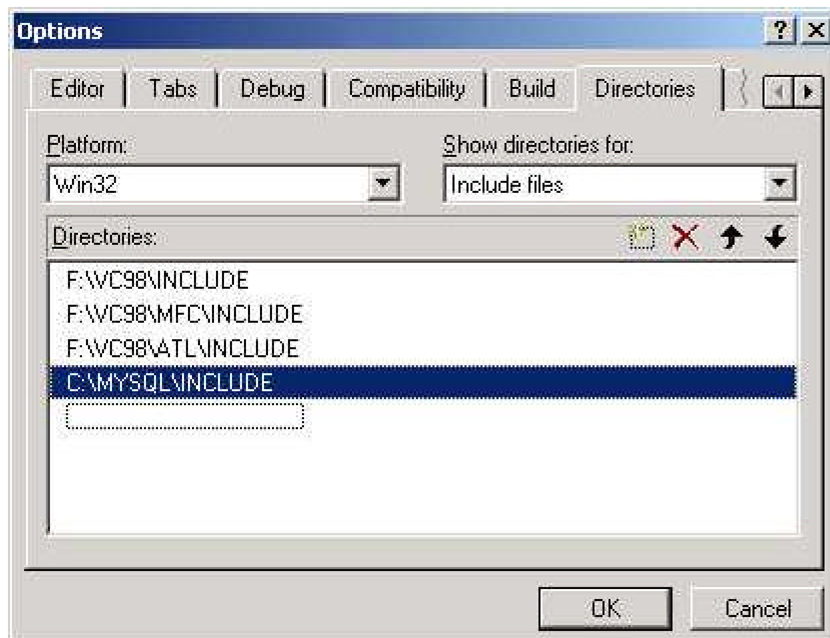


Figure 1: Répertoire où trouver les fichiers d'entête

et des « library files » (Fig.2)

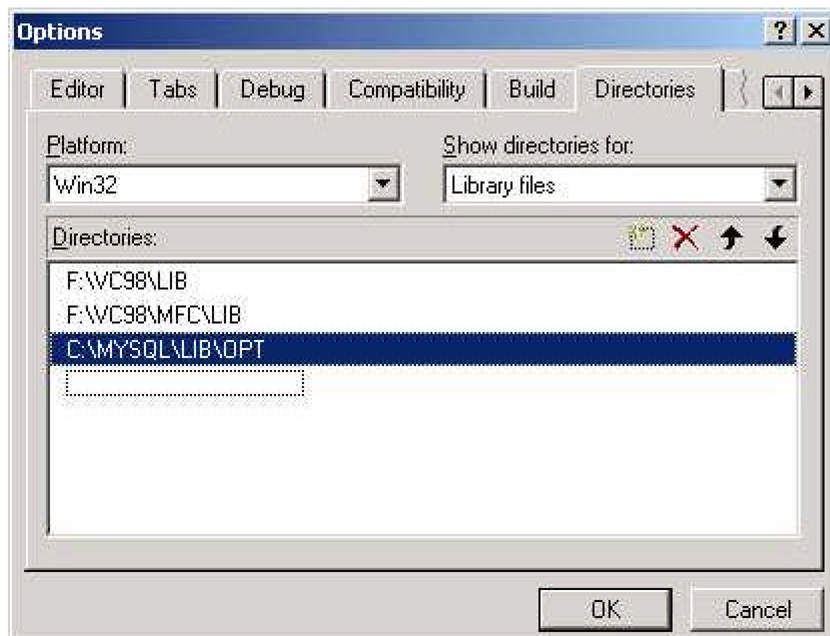


Figure 2: Répertoire où trouver les bibliothèques

III - Création du squelette d'application

On crée une application de type "Dialog Box". A l'étape 2, on s'assure de cocher la case ***Windows Sockets***. On peut déselectionner la case ActiveX Controls puisqu'on n'utilisera pas de contrôles ActiveX dans ce projet. A l'étape 3, on laisse les options par défaut. On clique ensuite sur le bouton Finish pour générer le squelette de l'application.

Un répertoire MyEssai est créé et contient tous les fichiers générés par AppWizard. Notre projet contient les classes:

1. CAboutDlg: classe de la boite de dialogue A Propos
2. CMyEssaiApp: classe définissant notre objet application (constructeur de notre boite de dialogue principale et initialisation de l'application)
3. CMyEssaiDlg: classe de notre boite de dialogue principale, où nous allons ajouter tout le code au fil des pages qui suivent.

Pour pouvoir utiliser la bibliothèque cliente de MySQL, il nous faudra inclure un seul fichier (mysql.h) à notre projet.

On ouvre le fichier **myessaidlg.h** et on ajoute la ligne

#include <mysql.h> (Fig. 3)

```
#if !defined(AFX_MYESSAIDLG_H
#define AFX_MYESSAIDLG_H__D72

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <mysql.h>
```

On peut ainsi accéder à toutes les fonctions de l'API MySQL

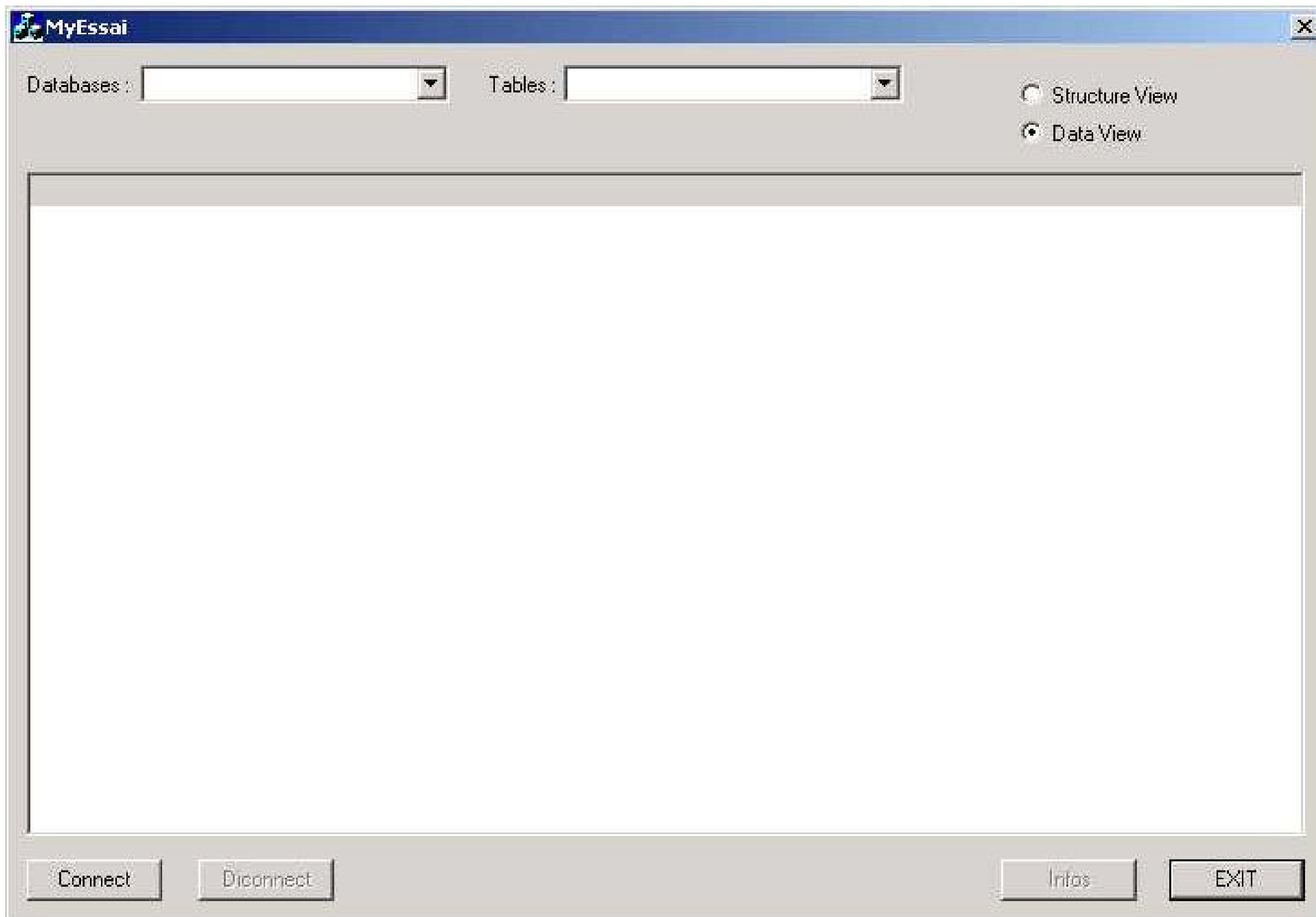
A noter que si vous omettez d'inclure le support Winsock, vous devrez éditer le fichier stdafx.h et ajouter la ligne #include <afxsock.h> sous le dernier #endif, comme sur la Figure 4.

```
#include <afxwin.h>         //
#include <afxext.h>         //
#include <afxdtctl.h>       //
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>         //
#endif // _AFX_NO_AFXCMN_SUPPOR

#include <afxsock.h>        //
```

Mysql.h inclut un certain nombre d'autres fichiers (le répertoire include contient en effet plusieurs autres fichiers *.h, mais seul mysql.h doit être inclus.

IV - Création des boites de dialogue de notre application

I - Boite de dialogue principale



On commence par créer la boite de dialogue principale. Par défaut, AppWizard crée pour nous une boite de dialogue munie d'un bouton OK et d'un bouton Cancel. Ces deux boutons sont mappés sur les fonctions CDialog::OnOK() et CDialog::OnCancel(), qui assurent un gestionnaire par défaut. Nous allons effacer ces deux boutons. Et ajouter les contrôles du tableau suivant:

Objet à insérer	ID de ressource
Boite de dialogue	IDD_MYESSAI_DIALOG
Un bouton libellé **Connexion**	IDC_CONNECT
Un bouton libellé **Deconnexion**	IDC_DISCONNECT
Un bouton libellé **Infos**	IDC_INFOS
Un bouton libellé **Quitter**	IDOK
Une boite liste précédée d'une étiquette **Bases de données**	IDC_DATABASES
Une boite liste précédée d'une étiquette **Tables**	IDC_TABLES
Un contrôle ListView réglé sur le mode **Report**	IDC_LV_DB
Un contrôle Radio Button libellé **Structure**	IDC_STRUCTURE_VIEW
Un contrôle Radio Button libellé **Données**	IDC_DATA_VIEW

Les deux boutons radio devront être mutuellement exclusifs, c'est-à-dire que lorsque l'un est coché, l'autre ne l'est pas et vice-versa. Pour faire cela, il faut sélectionner les deux contrôles, et, dans la boite de dialogue **properties**, cocher Auto. Ne pas cocher la case **Group**. Faire un essai dans l'éditeur de boites de dialogue afin de bien vérifier que les contrôles sont mutuellement exclusifs.

Boite de dialogue de connexion

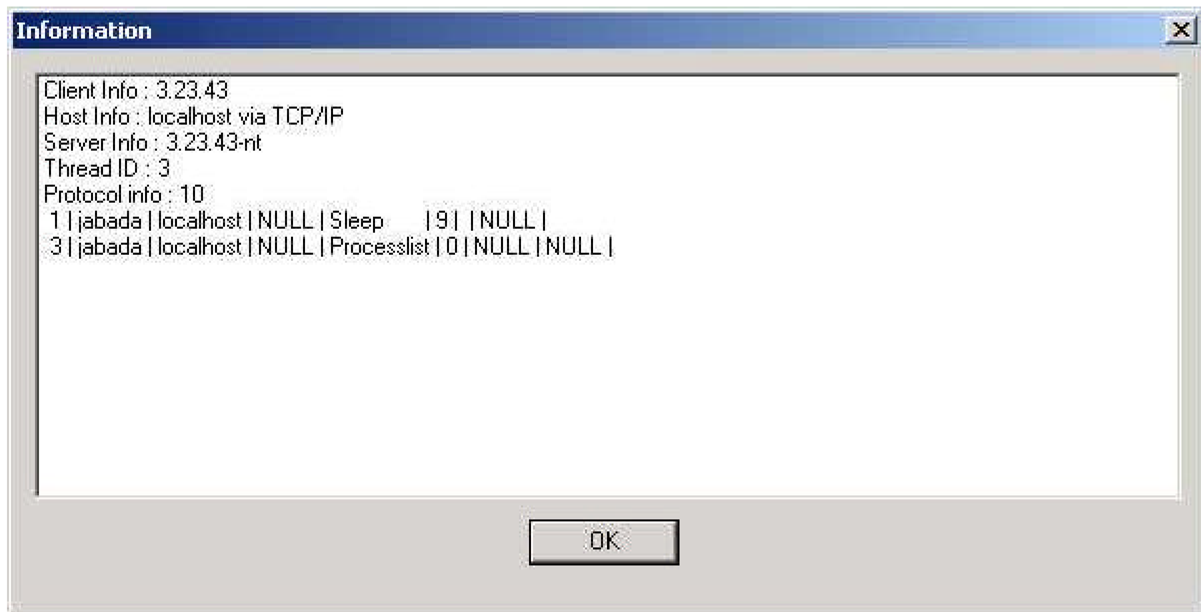
Pour créer la boite de dialogue de connexion, il faut cliquer avec le bouton droit de la souris sur le dossier *Dialog* de l'arborescence *Resource View*, et choisir *Insert Dialog*. On retaillera la boite pour qu'elle ressemble à celle présentée ci-dessous:

Elle sera composée des contrôles suivants:

Objet à insérer	ID de ressource
Boite de dialogue	IDD_CONNECT_DLG
Un contrôle d'édition avec une étiquette *Hôte*	IDC_HOST
Un contrôle d'édition avec une étiquette *Utilisateur*	IDC_USER
Un contrôle d'édition avec une étiquette *Mot de passe*	IDC_PASSWORD
Un contrôle d'édition avec une étiquette *Port*	IDC_PORT
Un bouton libellé modifié de *OK* ---> *Connexion*	IDOK (*)
Un bouton libellé modifié de *Cancel* --> *Annuler*	IDCANCEL (*)

(*) Ces deux contrôle sont déjà présents lors de la création de la nouvelle boite de dialogue. Ils ont pour valeur par défaut IDOK et IDCANCEL, deux valeurs qu'il ne faut pas modifier. Elles sont mappées sur les fonctions CDialog::OnOK() et CDialog::OnCancel(). L'ID IDOK nous permet de récupérer la valeur de la fonction CDialog::DoModal() que nous allons utiliser plus tard pour récupérer les valeurs des contrôles. En modifiant les ID par défaut, on risque de ne plus pouvoir ni récupérer les valeurs mais en plus la boite ne se refermera pas.

Boite de dialogue d'informations



Objet à insérer	ID de ressource
Boite de dialogue	IDD_INFOS_DLG
Un contrôle ListBox	IDC_INFOS

Le bouton Cancel sera supprimé, et le bouton OK sera laissé tel quel. On le replacera simplement dans la boite de dialogue pour qu'il figure sous la listbox.

V – Codage de la boite de dialogue de connexion

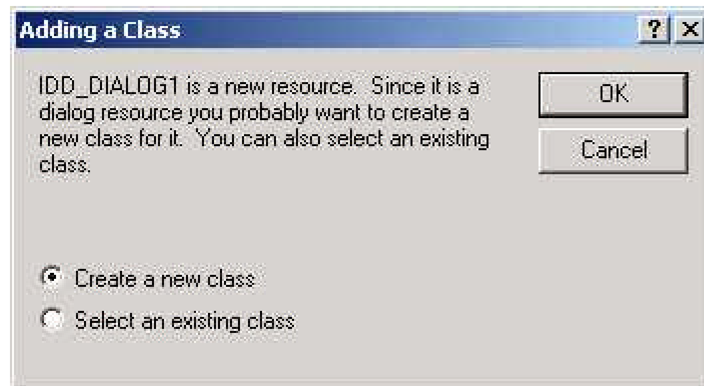
L'objectif est de récupérer les données que l'utilisateur entre dans la boite de dialogue de connexion une fois qu'il a cliqué sur le bouton **Connexion**. Ce bouton renvoie IDOK à la fonction appelante. Si l'utilisateur cliquer sur le bouton **Annuler**, c'est IDCANCEL qui est renvoyé.

Lorsque la fenêtre principale s'affiche, l'utilisateur clique sur le bouton **Connexion**. Celui-ci appelle la fonction CMyEssai::OnConnect(). Celle-ci commence par créer une variable de type CConnexionDlg et, via cette variable, affiche la boite de connexion en tant que boite modale, au moyen de la fonction CDialog::DoModal(). On vérifie dans le code que la fonction renvoie bien IDOK, au moyen du code suivant

```
void CMyEssaiDlg::OnConnect()
{
	// On récupère les données de la boite de dialogue connexion
	// et on les copie dans nos variables membres.

	CConnexionDlg dlg;
	if (dlg.DoModal() == IDOK)
```

Avant d'aller plus loin dans les explications, commençons par gérer notre boite de connexion. Il nous manque, d'une part, une classe pour cette boite de dialogue, et d'autre part des variables membres qui vont récupérer les données entrées par l'utilisateur. Nous avons besoin également d'accéder à la boite de dialogue de connexion depuis notre boite principale. Voici les étapes:

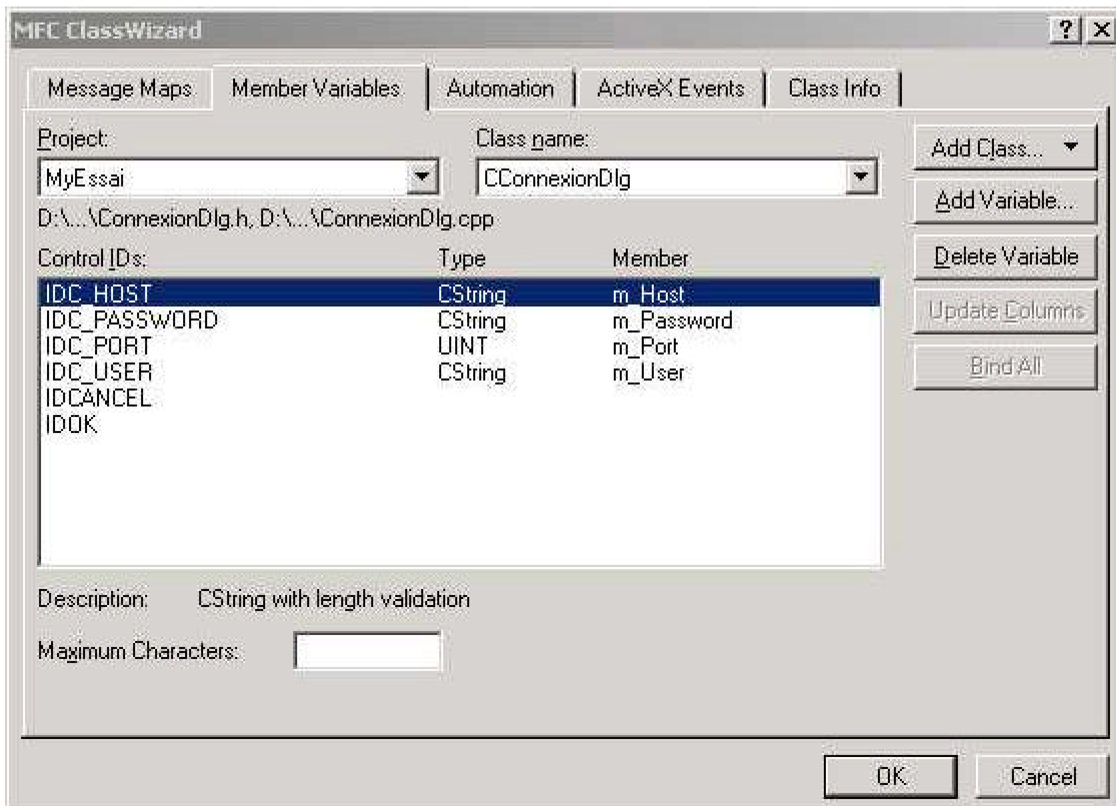
Une fois les contrôles placés dans la boite de connexion, et éventuellement sauvegarde des modifications, il faut lancer le ClassWizard. Il va détecter que la boite de dialogue est une nouvelle ressource.



Il faudra laisser l'option *Create a new class* cochée, et simplement cliquer sur OK, ce qui amènera une autre boite où il faudra saisir le nom de la classe. On entrera *CConnexionDlg* comme nom de classe et on laissera tous les autres champs à leur valeur par défaut. On cliquera sur OK pour valider nos modifications. A ce stade, ClassWizard a créé pour nous deux fichiers, ConnexionDlg.cpp et ConnexionDlg.h.
Pour pouvoir accéder aux données membres de la classe CConnexionDlg depuis notre classe CMyEssaiDlg, il nous faudra inclure le fichier ConnexionDlg.h à MyEssaiDlg.cpp au moyen de l'ajout de la ligne

```
#include "ConnexionDlg.h"
```

Ensuite, il nous manque les quatre données membres (variables membres) correspondant à nos quatre contrôle d'édition.
On lance à nouveau ClassWizard et on choisit l'onglet *Member Variables*.



On va successivement sélectionner IDC_HOST, IDC_PASSWORD, IDC_PORT et IDC_USER, et cliquer sur le bouton *Add Variable*, pour créer

1. Une variable *m_Host* de type *Cstring*
2. Une variable *m_Password* de type *Cstring*
3. Une variable *m_Port* de type *UINT*
4. et une variable *m_User* de type *CString*

Enfin, il faut initialiser le contrôle IDC_PORT. En effet, par défaut MySQL utilise le port 3306 pour la connexion, aussi on va modifier l'entrée dans le code du constructeur de la classe CConnexionDlg comme ci-après:

```
CConnexionDlg::CConnexionDlg(CWnd* pParent /*=NULL*/)
	: CDialog(CConnexionDlg::IDD, pParent)
{
	//{{AFX_DATA_INIT(CConnexionDlg)
	m_Host = _T("");
	m_Password = _T("");
	m_Port = 3306;
	m_User = _T("");
	//}}AFX_DATA_INIT
}
```

La variable *m_Port* est initialisée à 0 par défaut. On va remplacer le 0 par la valeur 3306. Et, cette valeur

apparaîtra dans le contrôle d'édition lorsque nous afficherons la boite de dialogue de connexion.

C'est tout ce qu'il faudra faire pour utiliser la boite de dialogue de connexion. En effet, à part mapper les quatre contrôles d'édition à des variables membre, initialiser la variable *m_Port* à 3306, tout le reste est géré par les fonctions que nous allons écrire dans la classe CMyEssaiDlg.

Ce que nous verrons à l'étape suivante.

VI - Codage de la connexion

Une connexion à un serveur MySQL se fait toujours en deux étapes:

1. Initialisation d'une structure MYSQL
2. Connexion proprement dite.

L'étape d'initialisation se fait au moyen de la fonction *mysql_init*, qui retourne un pointeur sur une structure MYSQL, ou NULL en cas d'échec dû à un manque de mémoire.

Pour ce faire, il faut d'abord créer une variable de type pointeur sur structure:

```
private:
	void ClearListView();

	CString	Host;
	CString	Password;
	UINT	Port;
	CString	User;
	MYSQL * myData;
```

Dans le fichier MyEssaiDlg.h, on crée une variable *MYSQL *myData*, myData étant un pointeur sur la structure MYSQL, définie dans le fichier *mysql.h*; on place cette variable dans la section private de notre classe.

Pour effectuer notre connexion au serveur, nous allons cabler une fonction au clic du bouton **Connexion** de la boite de dialogue principale. A l'aide de ClassWizard on va appeler notre fonction gestionnaire *OnConnect()*.

Plutôt que tout coder dans cette fonction, nous allons seulement récupérer les informations saisies dans la boite de dialogue de connexion, puis appeler une autre fonction qui fera la connexion proprement dite, puis au retour de cette fonction, nous appellerons une autre fonction de service qui récupèrera pour nous la liste des bases de données présentes sur le serveur et placera leur nom dans la boite combo correspondante.

Regardons de plus près le code ci-après:

```
void CMyEssaiDlg::OnConnect()
{
	// On récupère les données de la boite de dialogue connexion
	// et on les copie dans nos variables membres.

	CConnexionDlg dlg;
	if (dlg.DoModal() == IDOK)
	{
		Host = dlg.m_Host;
		User = dlg.m_User;
		Password = dlg.m_Password;
		Port = dlg.m_Port;

		if (!Connexion(Host, User, Password, Port))
			return;

		if (!GetDatabases())
			MessageBox("Impossible de récupérer les bases du serveur");

		m_MyInfosBtn.EnableWindow(TRUE);
		m_Connect.EnableWindow(FALSE);
		m_Disconnect.EnableWindow(TRUE);

	}
}
```

Après avoir créé une variable *dlg* de type CConnexionDlg, on affiche la boite de dialogue avec la fonction DoModal() et on teste si la valeur de retour est bien IDOK, c'est-à-dire si le bouton Connexion (IDOK) a été cliqué. Si c'est le bouton Annuler qui a été cliqué, on sort tout simplement de la fonction. Dans le cas contraire, on récupère la valeur des contrôles d'édition et on les copie dans les variables *Host, User, Password* et *Port*, créées dans notre classe CMyEssaiDlg, et ayant le même type que celles de la boite de dialogue de connexion (CString, CString, CString, UINT).

Puis, on appelle la fonction Connexion en lui passant comme paramètres les valeurs que nous avons récupérées. Cette fonction a le prototype suivant

```
BOOL CMyEssaiDlg::Connexion(CString& Host, CString& User, CString& Password, UINT Port);
```

elle retourne TRUE si la connexion a réussi et FALSE dans le cas contraire. Si elle renvoie FALSE on sort de la fonction OnConnect(), sinon on appelle une deuxième fonction CMyEssaiDlg::GetDatabases(). Cette fonction a le prototype suivant:

```
BOOL CMyEssaiDlg::GetDatabases();
```

et renvoie également TRUE en cas de réussite et FALSE en cas d'échec. Les trois dernières lignes de code activent ou désactivent des boutons de la boite de dialogue principale. Nous verrons ces petits détails à la fin.

Voyons maintenant la fonction CMyEssaiDlg::Connect(CString&, CString&, CString&, UINT).

```
BOOL CMyEssaiDlg::Connexion(CString& Host, CString& User, CString& Password, UINT Port)
{
	if ( (myData = mysql_init(NULL)) &&
	mysql_real_connect( myData,
					Host,
					User,
					Password,
					NULL,
					Port,
					NULL,
					0) )

	{
		m_Disconnect.EnableWindow(TRUE);
		m_Disconnect.EnableWindow(FALSE);
		IsConnected = TRUE;
		return TRUE;
	}
	else
	{
		CString msg;
		msg.Format("La connexion a échoué à cause de l'erreur\nn°: %d\n"
				"Texte de l'erreur : %s",
				mysql_errno(myData),
				mysql_error(myData));
		MessageBox(msg, "Erreur", MB_ICONERROR);
		return FALSE;
	}
}
```

La première ligne de code initialise myData (de type MYSQL) par l'appel de la fonction *mysql_init(MYSQL*)*. Cette fonction reçoit NULL comme paramètre. Et doit retourner un objet MYSQL en cas de succès et NULL en cas d'échec, puis on appelle la fonction *mysql_real_connect()*, en lui passant les paramètres suivants:

1. adresse de myData
2. Valeur du champ HOTE
3. Valeur du champ USER
4. Valeur du champ PASSWORD
5. NULL (on n'ouvre pas de base de données par défaut)
6. Valeur du champ PORT
7. NULL (on n'utilise pas les sockets UNIX)
8. 0 (pas de flags particuliers).

Je vous renvoie à la documentation sur cette fonction, et notamment la section qui décrit les flags (dernier parmètre), qui contient des possibilités très intéressantes.

mysql_real_connect renvoie un handle de connexion valide en cas de succès, et NULL en cas d'échec. En cas d'échec, nous formatons un message contenant le numéro de l'erreur (obtenu par l'appel de la fonction *mysql_errno*), et le texte de l'erreur (obtenu par l'appel de la fonction *mysql_error*).

Examinons maintenant la fonction GetDatabases().

```
BOOL CMyEssaiDlg::GetDatabases()
{
	if (!IsConnected)
		return FALSE;

	MYSQL_RES *res;
	res = mysql_list_dbs(myData, NULL);
	MYSQL_ROW row;
	while ( (row = mysql_fetch_row(res)))
	{
		m_Databases.AddString(row[0]);
	}
	mysql_free_result(res);
	return TRUE;
}
```

On a besoin de créer une variable de type BOOL, nommée *IsConnected*. Ce flag est positionné à FALSE au démarrage de notre application, et à TRUE quand une connexion a été correctement faite. On s'en sert tout au long du programme pour vérifier qu'on a toujours une connexion valide avant d'appeler les fonctions de MySQL..

On vérifie donc qu'on a bien une connexion active. Ensuite, on crée une variable *res* de type pointeur sur MYSQL_RES et on récupère l'ensemble résultat obtenu par la fonction *mysql_list_dbs()*. Cette fonction accepte en premier paramètre un handle de connexion et en second paramètre un patron de recherche. On passe NULL pour tout récupérer.

Ensuite, on récupère les données ligne par ligne dans la boucle *while* qui appelle la fonction *mysql_fetch_row()* autant de fois qu'il y a de lignes. Elle renvoie NULL quand il n'y a plus de lignes dans l'ensemble résultat.

A chaque itération de *mysql_fetch_row()*, on ajoute la colonne d'indice 0 (première colonne) à la combo. *m_Databases* est une donnée membre de type CComboBox, et on appelle la fonction AddString, membre de cette classe.

A la fin du traitement, on libère les ressources en appelant la fonction *mysql_free_result()*.

A ce point, on va maintenant coder l'affichage des tables dans la boite combo de droite (*m_Tables*). On n'est pas en mesure d'afficher les tables d'une base de données particulière, puisque le programme ne connait pas les bases que l'utilisateur aura sur son système.

A ce stade, on a effectué une connexion au serveur et affiché les bases dans une boite combo. Pour afficher les tables correspondantes dans la seconde combo, il suffit de câbler une fonction sur le message CBN_SELCHANGE qui est envoyé lorsque la sélection de la combo change.

L'algorithme consiste à récupérer le texte sélectionné (le nom de la base de données), et de le transmettre à la fonction *mysql_select_db()* qui va changer la base de données active, puis appeler la fonction *mysql_list_tables()* pour afficher la liste des tables de notre nouvelle base dans la combo de droite, en s'appliquant à effacer son contenu avant d'y insérer les noms des tables.
Pour cela il nous manque une fonction pour changer la base de données active, que nous allons appeler *ChangeDB()*.

Son prototype est le suivant:

BOOL CMyEssaiDlg::ChangeDB(CString& db_name);

Elle prend comme unique paramètre le nom de la nouvelle base de données à rendre active, et renvoie une valeur TRUE en cas de succès et FALSE dans le cas contraire. En cas d'échec, on formate un message affichant le nom de la base de données que l'on a tenté d'accéder, le numéro de l'erreur ainsi que le texte du message d'erreur renvoyé par le serveur.

La fonction est la suivante:

```
BOOL CMyEssaiDlg::ChangeDB(CString& db_name)
{
	if (!IsConnected)
		return FALSE;

	int Err = mysql_select_db(myData, db_name);
	if (Err == 0)
		return TRUE;
	else
	{
		CString msg;
		msg.Format("L'accès à la base %s a échoué à cause de l'erreur\nn°: %d\n"
		            "Texte de l'erreur : %s", db_name, mysql_errno(myData),
		             mysql_error(myData));
		MessageBox(msg, "Erreur", MB_ICONERROR);
		return FALSE;
	}
}
```

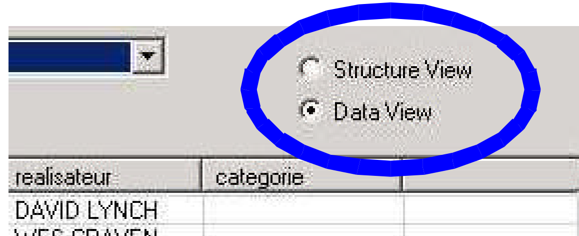
Maintenant que nous avons changé de base de données active, nous pouvons rechercher et afficher la liste des tables que celle-ci contient. La fonction *mysql_list_tables()* est assez similaire à *mysql_list_dbs()*, le premier paramètre étant le handle de connexion, et le second un patron de recherche. On passe *NULL* ici pour indiquer qu'on souhaite récupérer la liste de toutes les tables contenues dans notre base de données.

```
void CMySampleDlg::OnSelchangeDatabases()
{
	if (!IsConnected)
		return;

	CString currDB;

	int nIndex = m_Databases.GetCurSel();
	m_Databases.GetLBText(nIndex, currDB);

	if (!ChangeDB(currDB))
		return;

	m_Tables.ResetContent();

	MYSQL_RES *res;
	res = mysql_list_tables(myData, NULL);
	MYSQL_ROW row;
	while ( ( row = mysql_fetch_row(res)))
	{
		m_Tables.AddString(row[0]);
	}
	mysql_free_result(res);
}
```

Le fonctionnement du code est identique à celui qui alimente la boite combo des bases. On utilise un objet *MYSQL_RES** et on parcourt l'ensemble résultat retourné par la fonction *mysql_list_tables()*. Dans notre boucle *while*, c'est-à-dire, tant qu'il y a des lignes, on récupère la première colonne que l'on insère dans le contrôle combo. En fin de parcourt, on libère les ressources à l'aide de la fonction *mysql_free_result()*.

VII – Affichage des données dans la ListView

Maintenant que nous avons récupéré la liste des bases de données du serveur, sélectionné une base, et affiché la liste des tables qu'elle contient, nous pouvons gérer le message CBN_SELCHANGE de la combo contenant la liste des tables. Ici, lorsque notre utilisateur va sélectionner une table, on va devoir vérifier lequel des deux boutons radio il a coché (Données ou Structure), puis, construire notre Vue Liste et la remplir.



Les deux contrôles radio sont mutuellement exclusifs, pour qu'un seul ne soit actif en même temps. On va créer une variable membre pour chacun des contrôles, afin de tester s'il est coché ou non. Le bouton radio *Structure View* est mappé sur la variable *m_bDataView* de type *CButton*. Le bouton radio *Data View*, quant à lui, est mappé sur la variable *m_bStructureView* également de type *CButton*. Pour tester si l'un ou l'autre est coché, on utilise tout bonnement la fonction *CButton::GetCheck()*, qui retourne un entier indiquant l'état du bouton radio. S'il est coché, la fonction renvoie 1.

Tout ce passe donc dans le code de la fonction gestionnaire de l'événement CBN_SELCHANGE de la boite combo IDC_TABLES.

On commence donc par récupérer le texte sélectionné dans la boite combo contenant la liste des tables. Puis, on formate la requête. A ce point, on teste quel bouton radio est coché, et si c'est Structure, on formate la requête en tant que « *describe nom_table* »; si c'est Données, on formate la requête en tant que « *select * from nom_table* ».

```
	// Get the select text in the combo box
	int nIndex = m_Tables.GetCurSel();
	m_Tables.GetLBText(nIndex, currTable);

	// Check whether Structure View or Data View is checked
	// and format the query accordingly
	if (m_bDataView.GetCheck() == 1)
		query.Format("SELECT * FROM %s", currTable);
	else if (m_bStructureView.GetCheck() == 1)
		query.Format("DESCRIBE %s", currTable);
```

On lance ensuite la requête en appelant la fonction *mysql_query()* avec les paramètres corrects.

```
	if (( mysql_query(myData, query) == 0))
	{
		res = mysql_store_result(myData);
		num_fields = mysql_num_fields(res);
		fd = mysql_fetch_fields(res);
```

La fonction *mysql_query()* renvoie 0 en cas de succès. Une fois la fonction exécutée, on appelle la fonction *mysql_store_result()* pour récupérer l'ensemble résultat dans l'objet *res* de type *MYSQL_RES**. On récupère

ensuite le nombre d'enregistrements retournés à l'aide de la fonction *mysql_num_fields()*. Ce nombre d'enregistrements nous est nécessaire pour contruire la vue liste, avec le nombre correct de colonnes. En effet, on ne connait pas à l'avance le nombre de colonnes que contient une table, d'où la nécessité de créer une fonction *BuildListView()* qui va afficher le nombre de colonnes nécessaire. Cette fonction contient le code suivant:

```
BOOL CMySampleDlg::BuildListView(UINT num_Cols, MYSQL_FIELD *fd)
{
	UINT i;
	LVCOLUMN m_pCol;

	for (i = 0; i < num_Cols; i++)
	{
		m_pCol.pszText = fd[i].name;
		m_lvDbTable.InsertColumn(i, m_pCol.pszText, LVCFMT_LEFT, 100);
	}
	return TRUE;
}
```

La vue liste est mappée sur une variable *m_lvDbTable* de type *CListCtrl*. Le premier paramètre de cette fonction est le nombre de colonnes retourné par la fonction *mysql_nul_fields()*. Le second paramètre un pointeur sur une colonne. Cette fonction construit un nombre de colonnes égal au nombre des colonnes retournées dans la requête, et affiche le nom du champ de la table dans l'entête de colonne du contrôle. Elle retourne TRUE en fin de travail.

Maintenant, on va afficher de jolis traits de séparation entre les lignes et les colonnes de notre contrôle:

```
		// Build the listview headers based on the number of columns returned
		if (!BuildListView(num_fields, fd))
			return;

		// Set some nice grid lines and effect to the listview
		m_lvDbTable.SetExtendedStyle(LVS_EX_FULLROWSELECT |
									LVS_EX_ONECLICKACTIVATE |
									LVS_EX_GRIDLINES |
									LVS_EX_INFOTIP);
```

On appelle pour cela la fonction SetExtendedStyle de la classe CListCtrl, qui prend en paramètre une combinaison de styles. Je vous renvoie à la documentation MFC pour les styles possibles.

Maintenant que notre liste est correctement dessinée (nombre de colonnes, petits traits sympas), on a besoin de la remplir. C'est ce que le code suivant réalise:

```
            // Get all the rows in the result set
            while ( (row = mysql_fetch_row(res)))
            {
                  for (j = 0; j < num_fields; j++)
                  {
                        // Fill in the listview items and subitems
                        lvItem.mask = LVIF_TEXT;
                        lvItem.iItem = i;
                        lvItem.iSubItem = j;
                        lvItem.pszText = row[j];
                        m_lvDbTable.InsertItem(&lvItem);
                        m_lvDbTable.SetItem(&lvItem);
                  }
                  i++; // Next Item
            }
      }
      else  // Something went wrong
      {
            CString msg;
            msg.Format("Error %d in query %s\n%s", mysql_errno(myData), query,
mysql_error(myData));
            MessageBox(msg, "Error in Query", MB_ICONERROR);
      }
      // Now free the resources
      mysql_free_result(res);
```

On commence par faire une itération dans les lignes (boucle *while*). C'est la fonction *mysql_fetch_row()* qui récupère les données ligne par ligne. Puis on remplit la vue liste cellule par cellule en faisant une itération dans les colonnes. En cas de problème avec la fonction *mysql_fetch_row()*, on affiche le code et le message d'erreur correspondant. En fin de parcourt, on libère les ressources.

Le code que nous avons décortiqué est en entier dans les pages suivantes.

A l'étape suivante, nous allons examiner la boite de dialogue d'informations. Elle n'est pas très compliquée, et est là pour montrer les fonctions MySQL qui retournent des informations sur le serveur, le client, le protocole, etc...

```
void CMySampleDlg::OnSelchangeTables()
{
	CString query;
	CString currTable;
	LVITEM lvItem;
	MYSQL_RES *res;
	MYSQL_FIELD *fd;
	UINT num_fields;
	MYSQL_ROW row;
	UINT i = 0;
	UINT j = 0;

	// Clear any previous display of data
	ClearListView();

	// Get the select text in the combo box
	int nIndex = m_Tables.GetCurSel();
	m_Tables.GetLBText(nIndex, currTable);

	// Check whether Structure View or Data View is checked
	// and format the query accordingly
	if (m_bDataView.GetCheck() == 1)
		query.Format("SELECT * FROM %s", currTable);
	else if (m_bStructureView.GetCheck() == 1)
		query.Format("DESCRIBE %s", currTable);

	// Run the query
	if (( mysql_query(myData, query) == 0))
	{
		res = mysql_store_result(myData);
		num_fields = mysql_num_fields(res);
		fd = mysql_fetch_fields(res);

		// Build the listview headers based on the number of columns returned
		if (!BuildListView(num_fields, fd))
			return;

		// Set some nice grid lines and effect to the listview
		m_lvDbTable.SetExtendedStyle(LVS_EX_FULLROWSELECT |
						LVS_EX_ONECLICKACTIVATE |
						LVS_EX_GRIDLINES |
						LVS_EX_INFOTIP);


		// Get all the rows in the result set
		while ( (row = mysql_fetch_row(res)))
		{
			for (j = 0; j < num_fields; j++)
			{
				// Fill in the listview items and subitems
				lvItem.mask = LVIF_TEXT;
				lvItem.iItem = i;
				lvItem.iSubItem = j;
				lvItem.pszText = row[j];
				m_lvDbTable.InsertItem(&lvItem);
				m_lvDbTable.SetItem(&lvItem);
			}
			i++; // Next Item
```

```
                    }
                }
                else  // Something went wrong
                {
                    CString msg;
                    msg.Format("Error %d in query %s\n%s",
                                mysql_errno(myData),
                                query,
                                mysql_error(myData));
                    MessageBox(msg, "Error in Query", MB_ICONERROR);
                }
                // Now free the resources
                mysql_free_result(res);
}
```

VIII – Codage de la boite Infos

J'ai choisi de coder la boite d'infos comme une boite de dialogue non modale et de gérer sa création et l'affichage des données qu'elles contient dans une fonction OnMyInfos() située dans la classe principale de notre application (CMySampleDlg).

```
void CMySampleDlg::OnMyINFOS()
{
	if (!IsConnected)
		return;

	CMyInfos *pDlg;
	pDlg = new CMyInfos();

	if (pDlg != NULL)
	{
		BOOL ret = pDlg->Create(IDD_INFOS_DLG, this);
		if (!ret)
		{
			MessageBox("Cannot create the informations dialog box");
		}

		pDlg->m_MyInfos.ResetContent();
		int nIndex = pDlg->m_MyInfos.GetCount();
		CString buff;

		buff.Format("Client Info : %s", mysql_get_client_info());
			pDlg->m_MyInfos.InsertString(nIndex++, buff);

		buff.Format("Host Info : %s", mysql_get_host_info(myData));
			pDlg->m_MyInfos.InsertString(nIndex++, buff);

		buff.Format("Server Info : %s", mysql_get_server_info(myData));
			pDlg->m_MyInfos.InsertString(nIndex++, buff);

		buff.Format("Thread ID : %ld", mysql_thread_id(myData));
			pDlg->m_MyInfos.InsertString(nIndex++, buff);

		buff.Format("Protocol info : %ld", mysql_get_proto_info(myData));
			pDlg->m_MyInfos.InsertString(nIndex++, buff);
```

Le fragment de code ci-dessus montre l'utilisation des fonctions MySQL *mysql_get_client_info()*, *mysql_get_host_info()*, *mysql_get_server_info()*, *mysql_get_thread_id()*, et *mysql_get_proto_info()*. Ces fonctions retournent respectivement la version de la dll client (libmysql.dll), le type de connexion, la version du serveur, l'ID du processus client (en fait numéro du thread de la session MySQL que nous avons ouvert), et enfin, un entier décrivant le protocole utilisé. La documentation de MySQL contient toutes les informations relatives à ces fonctions.

```
            MYSQL_RES *res;
            MYSQL_ROW row;
            MYSQL_FIELD *fd;
            unsigned int i;
            CString tmp;

            res = mysql_list_processes(myData);

            while ((row = mysql_fetch_row(res)) != NULL)
            {
                  mysql_field_seek(res, 0);
                  for (i = 0; i < mysql_num_fields(res); i++)
                  {
                        fd = mysql_fetch_field(res);
                        if (row[i] == NULL)
                              buff.Format(" %-*s |", fd->max_length, "NULL");
                        else if (IS_NUM(fd->type))
                              buff.Format(" %*s |", fd->max_length, row[i]);
                        else
                              buff.Format(" %-*s |", fd->max_length, row[i]);
                        tmp += buff;
                  }
                  pDlg->m_MyInfos.InsertString(nIndex++, tmp);
                  tmp.Empty();
            }


            pDlg->ShowWindow(SW_SHOW);
            pDlg->CenterWindow();
            mysql_free_result(res);
      }
      else
            MessageBox("Error creating object");
}
```

La suite du code appelle la fonction *mysql_list_processes()*, qui correspond à la requête SQL « SHOW PROCESSLIST ». On obtiendrait le même résultat en formatant cette requête et en appelant la fonction *mysql_query()*. La fonction *mysql_list_processes()* retourne un ensemble résultat, qu'il faut ensuite décoder.

Maintenant, il suffit de détruire notre boite de dialogue modale. Il suffit de placer le code suivant dans le gestionnaire du bouton OK:

```
void CMyInfos::OnOK()
{
      this->DestroyWindow();
}
```

Conclusion

Ca y est, notre application est terminée! Pour la lancer sur une autre machine, il suffit de copier le programme exécutable et la dll. Si la machine peut accéder à un serveur MySQL, alors vous pourrez afficher les données des bases pour lesquelles vous avez des droits. Il n'y a aucun fichier supplémentaire à copier sur la machine hote.

Ce petit programme est une introduction à tout ce qu'il est possible de faire avec MySQL et un langage de programmation. Ca change de l'univers fermé de certain logiciel que je ne nommerai pas. MySQL ne possède ni interface graphique, ni assistants. Il ne contient que ce qui est nécessaire à un bon développeur de bases de données.

Le C++ n'est pas le seul langage pouvant s'interfacer avec MySQL. Le plus célèbre est certainement PHP dont la présence sur les sites web dynamiques, allié à l'excellent serveur Apache, en font un redoutable outil de développement.

Sous Windows, il existe plusieurs outils gratuits (sur le site http://www.mysql.com), qui permettent d'utiliser MySQL:

1. MyODBC: pour travailler sur des bases MySQL via ODBC (Open Database Connectivity)
2. MyOleDB: Provider OleDB pour MySQL, à utiliser avec Visual Basic et des jouets dans le même genre.
3. TMySQL: un composant pour Delphi, qui permet d'utiliser les bases MySQL.
4. Zeos: une série de composants pour Delphi et Kylix, open source et pour travailler avec des bases MySQL.
5. MyAccess97 et MyAccess2000: add-ins pour M$ Access, permettant de travailler des bases MySQL de manière graphique, sous M$ Access. Mais compte tenu du prix d'Access tout seul, je ne conseille pas de les utiliser. Il suffit d'utiliser un des nombreux programmes d'administration de bases MySQL gratuits, disponibles sur le site de MySQL. Ou simplement, avec un serveur web Apache, PHP et MySQL installés sous Windows, d'utiliser phpMyAdmin.

J'espère que ce tutoriel vous apportera beaucoup. N'hésitez pas à me contacter si vous détectez des erreurs, ou si vous avez des questions. Je ne sais pas si je pourrai répondre à toutes les questions, mais j'essaierai.

Bon MySQL à tous!

Jacques Abada