Université de Nice-Sophia Antipolis
Polytech ELEC4

Lundi 20 avril 2020
Durée: 2h

MySQL et C++
Travaux Dirigés – Séance n. 13

---

1 Introduction

MySQL est un système de gestion de bases de données *relationnelles*, distribué librement ou sous licence propriétaire. Il est actuellement la propriété d'Oracle. Le site `www.mysql.fr` vous donnera toutes les informations nécessaires à son bon usage.

L'accès aux bases de données *MySQL*, gérées par le *serveur MySQL*, se fait à l'aide du langage de requête *SQL* (*Structured Query Language*), et par l'intermédiaire du *client mysql* (sous Linux), mais il est évidement possible d'accéder aux bases de données *MySQL* depuis des programmes écrits dans différents langages de programmation. Pour cela, l'environnement *MySQL* met à la disposition des utilisateurs des *connecteurs* qui assurent l'interface entre les bases de données *MySQL* et les langages de programmation. Pour C++, le connecteur est une API qui contient en particulier une bibliothèque et des fichiers d'en-têtes (.h).

Dans un premier temps, nous allons voir comment installer, sous linux, le connecteur pour C++, *MySQL Connector/C++*. Nous verrons ensuite comment, par son intermédiaire, écrire des programmes en C++ pour accéder aux bases de données *MySQL*.

2 MySQL Connector/C++

Sous Ubuntu, l'installation du connecteur se fait simplement :

```
sudo apt-get update
sudo apt-get install libmysqlcppconn-dev
```

Pour les autres systèmes, allez à l'url `dev.mysql.com/downloads/connector/cpp` et récupérez la dernière version (actuellement 1.1.8) pour votre système d'explotation et pour votre architecture de machine. Puis, déballez l'archive dans un répertoire d'installation, comme par exemple `/usr/local/lib/c++`. Le déballage crée un répertoire de distribution avec un numéro de version et d'architecture. Je vous conseille de faire un lien sur ce répertoire que nous nommerez simplement (par exemple) `mysql-connector-c++`. Par la suite, c'est ce nom de répertoire que nous utiliserons.

Le connecteur pour C++ utilise la bibliothèque *boost* (`www.boost.org`) qu'il faudra donc préalablement installer. Sous ubuntu, cela se fait simplement à l'aide de la commande :

```
sudo apt-get install libboost-all-dev
```

Si vous avez installé le connecteur *mysql* dans le répertoire `/usr/local/lib/c++/`, il faut permettre son chargement dynamique lors de l'exécution des programmes C++. Pour cela, ajoutez à la fin du fichier `/etc/ld.so.conf.d/libc.conf`, le chemin d'accès à cette bibliothèque `/usr/local/lib/c++/mysql-connector-c++/lib`, sauvegardez et ensuite exécutez la commande `ldconfig`.

Voilà, vous pouvez maintenant accéder aux bases de données *MySQL* depuis un programme C++. Par la suite, on considère que vous avez un serveur *MySQL* opérationnel sur votre ordinateur.

3 Programmation C++

Pour compiler vos programmes C++ qui utilisent le connecteur MySQL, il faudra ajouter des options de compilation, d'une part pour accéder aux fichiers d'en-tête (.h). Pour l'installation standard Ubuntu :

```
-I/usr/include/cppconn
```

Et pour l'installation dans `/usr/local/lib/c++` :

```
-I/usr/local/lib/c++/mysql-connector-c++/include
```

et d'autre part pour accéder à la bibliothèque *mysqlcppconn* fournie par le connecteur. Pour l'installation standard Ubuntu :

```
-lmysqlcppconn
```

Et pour l'installation dans `/usr/local/lib/c++` :

```
-L/usr/local/lib/c++/mysql-connector-c++/lib -lmysqlcppconn
```

3.1 Connexion au serveur MySQL

Pour établir la connexion avec le serveur *MySQL* qui gère les bases de données, il faut d'abord charger le pilote, instance de `sql::Driver`. À partir de ce pilote, on obtient ensuite une connexion en spécifiant l'adresse du serveur MySQL, l'identifiant de l'utilisateur et son mot de passe. Le programme suivant établit la connexion au serveur *MySQL* qui tourne sur la machine locale *localhost*. On se connecte sous l'identité *id* avec le mot de passe *secret*. La connexion doit être libérée une fois son utilisation achevée. Enfin, notez que des exceptions de type `sql::SQLException` peuvent être émises.

```
#include <stdlib.h>
#include <iostream>

#include <driver.h>
#include <connection.h>

int main() {

  sql::Driver *driver;
  sql::Connection *conn;

  try {
    driver = get_driver_instance();
    conn = driver->connect("localhost", "id", "secret");
    //
    // La connexion est établie => on peut soumettre des requêtes à la BD
    // ....
    // .... requêtes .....

    // libérer la connexion
    delete conn;
  }
  catch (sql::SQLException &e) {
    std::cerr << "# ERR: SQLException in " << __FILE__;
    std::cerr << "(" << __FUNCTION__ << ") on line " << __LINE__ << std::endl;
```

```
    std::cerr << "# ERR: " << e.what();
    std::cerr << " (MySQL error code: " << e.getErrorCode();
    std::cerr << ", SQLState: " << e.getSQLState() << " )" << std::endl;
  }
  return EXIT_SUCCESS;
}
```

La validité de la connexion peut être testée. Si la connexion est perdue, on peut rétablir la connexion, comme le montre le code suivant :

```
if (!conn->isValid()) {
      conn->reconnect();
}
```

exercice 1) À l'aide du programme précédent, testez la connexion à votre système de gestion de bases de données *MySQL*.

3.2 Requêtes simples

On souhaite créer une base de données que nous appellerons *Test_BD* avec une table *article* pour décrire des articles vendus par un bazar. Cette table possède les attributs *label* (une chaîne de caractères), *prix* (un réel), et une clé d'identification primaire unique *id_ref* (un entier).

Il existe plusieurs fonctions pour exécuter une requête *SQL* depuis un programme C++. La première d'entre elles, celle de base, est `sql::Statement::execute`. Elle renvoie un booléen égal à *vrai* si la requête renvoie plusieurs résultats, ou *faux* si la requête ne renvoie aucun résultat.

L'application de la méthode `execute` nécessite la construction préalable d'une instance de `sql::Statement` à partir de la connexion ouverte :

```
sql::Statement *stmt;
stmt = conn->createStatement();
```

Le fragment de code suivant crée la base données *Test_BD* (après suppression si elle existait préalablement), crée la table *article* (après suppression si elle existait préalablement) et insère des articles particuliers.

```
#include <statement.h>
....
// créer la base de données Test_BD
stmt->execute("DROP DATABASE IF EXISTS Test_BD");
stmt->execute("CREATE DATABASE Test_BD");
// utilisez cette BD
stmt->execute("USE Test_BD");
// créer la table « article »
stmt->execute("DROP TABLE IF EXISTS article");
stmt->execute("CREATE TABLE article(id_ref INT PRIMARY KEY NOT NULL, \
                                    label VARCHAR(100), prix DOUBLE)");
// insérer des articles dans la table
stmt->execute("INSERT INTO article(id_ref, label, prix) \
                           VALUES (11179, 'ballon',  10.0)");
stmt->execute("INSERT INTO article(id_ref, label, prix) \
                           VALUES (11123, 'matelas', 300.5)");
stmt->execute("INSERT INTO article(id_ref, label, prix) \
                           VALUES (11159, 'lampe', 55.99)");
stmt->execute("INSERT INTO article(id_ref, label, prix) \
                           VALUES (11199, 'tapis', 239.99)");
//
delete stmt;
....
```


exercice 2) Testez le code précédent. Vérifiez à l'aide de *phpMyAdmin* que la base de données *Test_BD* et la table *article* ont bien été créées.

exercice 3) Modifiez votre programme en ajoutant des articles supplémentaires.

On veut modifier un attribut d'un article déjà présent dans la table. Pour cela, on utilise la requête *SQL* `UPDATE`. Par exemple, le prix de la lampe a baissé, il faut le modifier. Cette modification peut se faire comme suit :

```
stmt->execute("UPDATE article SET prix=49.99 WHERE id_ref=11159");
```

exercice 4) Testez la requête précédente, et écrivez sur la sortie standard la valeur booléenne résultats de son exécution.

La méthode `sql::Statement::executeUpdate` est plus adaptée à la requête `UPDATE` dans la mesure où elle renvoie le nombre de modifications effectuées par la requête. On récrira la requête précédente comme suit :

```
stmt->executeUpdate("UPDATE article SET prix=49.99 WHERE id_ref=11159");
```

exercice 5) Testez la requête précédente, et écrivez sur la sortie standard le nombre de modifications effectuées.

exercice 6) Que se passe-t-il si vous essayez de remplacer une valeur existante par cette même valeur ?

Notez que l'exécution d'`executeUpdate` est en fait équivalente à l'appel d'`execute` suivie de celui de `sql::Statement::getUpdateCount`.

exercice 7) Testez la méthode `getUpdateCount`.

3.3 Requêtes avec résultats

La requête *SQL* `SELECT` permet de rechercher des données à partir d'une ou plusieurs tables. Le résultat est présenté sous forme d'une table.

Pour une requête `SELECT`, on utilise la méthode `sql::Statement::executeQuery` qui renvoie un pointeur sur un objet de type `sql::ResultSet`. Notez que l'appel la méthode `executeQuery` est équivalent à l'appel de `execute` suivi de de celui de `sql::Statement::getResultSet`.

Dans l'exemple suivant, on recherche tous les articles enregistrés dans la table *article*. Le résultat est en ordre croissant des références.

```
sql::ResultSet  *res;
....
res = stmt->executeQuery("SELECT * FROM article ORDER BY id_ref ASC");
...
delete res;
```

Un objet `ResultSet` est une suite de lignes accessibles individuellement par un *curseur*. Ce curseur détermine la ligne accessible, appelée ligne courante. Le déplacement du curseur se fait de façon *séquentielle* à l'aide de méthode `sql::ResultSet::next`. Initialement, le curseur est positionné sur la 1ère ligne de la table.

L'accès aux différentes valeurs de la ligne courante se fait par des méthodes `get` suffixées par le nom d'un type compatible avec le type *SQL* des valeurs. Les méthodes `get`*Xxx* possèdent comme paramètre, soit le nom de la colonne *SQL*, soit un numéro d'ordre (la première colonne a le numéro 1).

Le fragment de code C++ suivant affiche l'intégralité du résultat contenu par l'exécution de la méthode `executeQuery` précédente.

```
while (res->next()) {
  std::cout << res->getInt("id_ref") << " ";
  std::cout << res->getString("label") << " ";
  std::cout << res->getDouble("prix") << std::endl;
}
```

Remarque : la bibliothèque du connecteur C++/MySQL ne fournit pas la méthode `getDate` pour traiter des valeurs de type Date. On traitera les dates comme des chaînes de caractères.

exercice 8) Testez le code précédent.

exercice 9) Écrivez la requête qui permet d'obtenir tous les articles qui dont le prix est inférieur à 100.0 euros, et affichez le résultat sur la sortie standard.

3.4 Requêtes préparées

Les requêtes préparées sont représentées par des objets de type `sql::PreparedStatement`. En général, on utilise une requête préparée lorsqu'on a plusieurs fois le même type de requête à exécuter. À sa création, on lui passe la requête. Celle-ci est précompilée, ce qui en permettra une exécution multiple plus rapide par le SGBD.

Le deuxième intérêt des `sql::PreparedStatement`, c'est la possibilité de paramétrer les requêtes. Imaginons que dans un vecteur nous ayons un ensemble d'articles représentés par une classe `Article` qui contient les variables pour l'identifiant, le label et le prix de l'article. Cette classe possède également les accesseurs pour ces varaibles.

On possède, par exemple, un vecteur d'articles déclaré comme suit :

```
std::vector<Article> va = {
    Article (11179, "ballon", 10.0),
    Article (11123, "matelas", 300.5),
    Article (11159, "lampe", 49.99),
    Article (11199, "tapis", 239.99)
};
```

On souhaite insérer tous les articles contenus dans le vecteur dans la base de données. On commence par construire un objet `PreparedStatement` avec la requête *SQL* paramétrée d'insertion dans la table *article*.

```
#include <prepared_statement.h>
...
sql::PreparedStatement *prep_stmt;
...
prep_stmt = conn->prepareStatement("INSERT \
                                    INTO article(id_ref, label, prix) \
                                    VALUES (?, ?, ?)");
...
delete prep_stmt;
```

Notez les trois points d'interrogation dans la requête *SQL*. Ils correspondent aux trois paramètres données de la requête. Le premier est l'identifiant, le second le label, et enfin le troisième est le prix. Le code qui suit effectue l'insertion à partir des articles contenus dans le vecteur :

```
// insérer des articles dans la table
for (Article a : va) {
  // fixer les valeurs de chacun des paramètres
```



```
  prep_stmt->setInt(1, a.getId_Ref());
  prep_stmt->setString(2, a.getLabel());
  prep_stmt->setDouble(3, a.getPrix());
  // lancer la requête INSERT
  prep_stmt->execute();
}
```

Les paramètres de la requête sont numérotés à partir de 1, et on leur affecte des valeurs à l'aide de méthodes `set` suffixées par un type compatible avec celui de la valeur *SQL* du paramètre (*e.g.* `setInt`, `setDouble`, `setString`...).

exercice 10) Testez le code précédent.

exercice 11) Un vecteur contient une suite de couples *(id_ref, prix)*. Chaque couple définit un nouveau *prix* pour un article d'identification *id_ref*. À l'aide d'une requête préparée, mettez à jour les nouveaux prix des articles référencés.

3.5 Procédures stockées

Une *procédure stockée* est une suite d'instructions nommée et *enregistrée dans* la base de données. Elle est créée à l'aide de la requête *SQL* `CREATE PROCEDURE`. Elle peut être ensuite appelée à l'aide de la requête *SQL* `CALL`.

Avec le connecteur, une procédure stockée peut être appelée à l'aide d'une d'instance de `Statement` ou `PreparedStatement`. On pourra distinguer 3 types de procédure stockée :

1. celle qui ne renvoie aucun résultats ;
2. celle qui ne renvoie des résultats par l'intermédiaire de paramètre « résultat » ;
3. celle qui renvoie un `ResultSet`.

exercice 12) Ajoutez à la base de données *Test_BD* une procédure stockée, appelée *compterArticles* qui calcule le nombre d'articles inférieurs à certains prix. Le prix est un paramètre « donnée » et le nombre d'articles calculé est un paramètre « résultat ». Cette procédure est définie comme suit :

```
mysql> CREATE PROCEDURE compterArticles(IN p DOUBLE, OUT n INT)
    -> SELECT COUNT(*) INTO n FROM article WHERE prix < p;
```

L'utilisation de cette procédure stockée se fait par la séquence de requêtes sql suivante :

```
mysql> set @n=0;
mysql> call compterArticles(200, @n);
mysql> SELECT @n;
```

La première requête initialise la variable *n* (résultat) à 0. Notez l'emploi de `@`. La seconde appelle la procédure *compterArticles* stockée dans la bd. La dernière récupère le résultat contenue dans la variable *n*.

exercice 13) Écrivez la procédure `stockerProc` qui stocke la procédure *compterArticles* dans la base de données *Test_BD*.

exercice 14) Écrivez la fonction `executerProc` qui exécute la procédure stockée *compterArticles*. Vous passerez en paramètre le prix et la fonction renverra le nombre d'articles.

exercice 15) Testez vos deux fonctions précédentes.

On souhaite classer les articles précédents par rayons dans lesquels ils sont proposés aux clients

du bazar. Par exemple, les articles *matelas*, *lampe* et *tapis* se trouvent au rayon *maison*, alors que l'article *ballon* est au rayon *jeux*

exercice 16) Dans la base de données *Test_BD*, créez une seconde table `rayon` avec les 4 colonnes : une clé primaire, le nom du rayon, la référence à un article du rayon, et la quantité en stock de cet article.

exercice 17) Déclarez la classe `Rayon` pour représenter un rayon du bazar. Vous définirez un constructeur et des accesseurs.

exercice 18) Initialisez un vecteur de `Rayon` pour mémoriser les 4 articles du vecteur d'articles précédent. Pour chaque article, vous mettrez des quantités de votre choix.

exercice 19) écrivez la procédure `printSolde` qui exécute un requête sql pour récupérer le nom des articles, avec le nom de leur rayon, dont la quantité est inférieure à une valeur `n` passée en paramètre. La procédure affichera ensuite le résultat sur la sortie standard.

Par exemple, la procédure pourra afficher les articles dont la quantité est inférieure à 20 comme suit :

```
matelas maison  10
tapis   maison  15
```

4 Problème

exercice 20) Développez une petite application pour gérer un magasin formé de rayons contenant des articles mémorisés dans une base de données.

Vous définirez une classe `Bazar` dont le constructeur établit la connexion avec la base de données. Vous écrirez des méthodes qui permettent à un utilisateur de créer les tables *rayon* et *article*, d'ajouter des rayons, d'approvisionner en articles les rayons du magasin, de visualiser le contenu des rayons...

Dans une méthode `main`, vous créerez un objet `Bazar`, et vous testerez vos méthodes.