TALLINN UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

Department of Computer Science

Andrei Mihnevitš 123856IASB

IAG0582 Programming 2

**Using MySQL in C++**

Supervisor: docent Vladimir Viies

Tallinn 2013

Contents

[Declaration of authorship 3](#_Toc352414263)

[Preface. 4](#_Toc352414264)

[Choosing MySQL++ 5](#_Toc352414265)

[Setting up MySQL 6](#_Toc352414266)

[Creating a project in Visual Studio 7](#_Toc352414267)

[The source code. 8](#_Toc352414268)

[Character encodings 10](#_Toc352414269)

[Back to the code 12](#_Toc352414270)

[Conclusion 15](#_Toc352414271)

[List of literature used 16](#_Toc352414272)

# Declaration of authorship

I hereby certify that the thesis I am submitting is entirely my own original work except where otherwise indicated.

Date: 31.03.2013

Student: Andrei Mihnevitš

# Preface.

I decided to write this document in English to make it more international, so that estonian students could also benefit from my work.

During the process I encountered several problems I had to deal with in order to do the task correctly, so the whole research I have made is a bit deeper than a simple MySQL database query. Although some of the aspects described here are not explicitly connected to MySQL operation itself I still encourage you to review these aspects thoroughly because in my opinion this informatrion is vital for understanding.

There is one more thing that I would like to mention here. The software project I’m going to represent is Windows oriented and is built in Visual Studio 2008 Express. So most of the information in this document refers to Windows operating system.

# Choosing MySQL++

First of all, I should mention that if you plan to use MySQL in your C programme then you may simply go with MySQL C API [[1]](#_List_of_literature).

API stands for Application Programming Interface. In general such an interface can be actually understood as a set of different routines or functions provided for communication between software components.

In our case MySQL C API provides an interface to communicate with a low-level MySQL client/server protocol.

With that being said, we can switch to MySQL++ [[2]](#_List_of_literature), which is basically a C++ wrapper for MySQL C API. The authors state that “its purpose is to make working with queries as easy as working with STL containers” [[3]](#_List_of_literature). Well, we are going to check that out.

# Setting up MySQL

The first thing you have to be aware of is that you need to have the Windows version of the MySQL server installed on your development system. The reason for this is that along with the server itself the Windows installation setup provides the client-side development files required by MySQL++.

Currently the latest version of MySQL++ is 3.1.0 and it only supports the MySQL server version 5.1 (despite the fact that there is already version 5.6 of the MySQL server available). This was a real headache for me when I first tried to set things up. So in order to spare your nerves I will try to give you the correct installation scheme:

1. Download the mysql-5.1.68-win32.msi setup file [[4]](#_List_of_literature) and run the setup. After the installation check the path where the server has been installed. For example, in my case it was:

C:\Program Files (x86)\MySQL\MySQL Server 5.1

You will need it later.

1. Download the mysql++-3.1.0.tar.gz library source code archive [[5]](#_List_of_literature) and extract it somewhere.
2. Go to the folder where you have extracted mysql++ source code and find the file named mysql++.bkl. Open this file with any text editor. Search for the “MYSQL\_WIN\_DIR” string. When you have found it check the path there and compare it to the MySQL server installation path. Update the mysql++.bkl file with the correct path if it is needed.
3. If during the previous step you had to update the mysql++.bkl file then download and install the bakefile-0.2.9-setup.exe file [[6]](#_List_of_literature). Once it’s installed open the folder which stores the extracted MySQL++ files and run the rebake.bat file. It will update all the Visual Studio project files with the correct MySQL server path.
4. Now assuming you have Visual Studio 2008 Express [[7]](#_List_of_literature) installed open the folder named “vc2008” and run mysql++.sln file. When Visual Studio loads you may cross your fingers and try to build the solution in Debug mode. Hope this goes well…You might as well build the release version but it’s not that necessary right now.
5. When the projects are built you can run install.hta file located in the main directory where you extracted MySQL++ files. This will collect the recently built binaries and put them into directory of your choice.
6. Now you have to update your system environment variable called PATH. On Windows 7 just press the “Start” button and type in the search field “view advanced system settings”. In the opened dialog window press the “Environment variables” button. There, among the system variables, you can find the “Path” variable, select it and press the “Edit…” button. In the “Variable value” field simply add two strings separated by a semicolon:

“your\_mysql\_server\_installation\_path/bin”

“directory\_path\_where\_you\_have\_installed\_mysql++\_binaries\_with\_install\_hta/lib”

Once you launch your application which executes MySQL queries the operating system will know where to find the required DLL files. Otherwise you get an error that a required dynamic library cannot be found.

# Creating a project in Visual Studio

Once MySQL++ binaries have been installed we can go on with our own code.

Simply create a new empty project for a console application. Then open the project properties dialog window. There are a few important options you have to check:

1. C/C++ → General → Additional Include Directories:

The first path will be “your\_mysql\_server\_installation\_path/include”. This one is for mysql\_version.h header file.

The second path will be “directory\_path\_where\_you\_have\_installed\_mysql++\_binaries\_with\_install\_hta/include”

1. C/C++ → Advanced → Compile As:

There should be “Compile as C++ Code”.

1. Linker → General → Additional Library Directories:

You need to specify here only one path:

“directory\_path\_where\_you\_have\_installed\_mysql++\_binaries\_with\_install\_hta/lib”

1. Linker → Input → Additional Dependencies:

Type in that field “mysqlpp\_d.lib” for the debug mode version of your application.

After that you can either add new source files to this project (e.g. main.cpp) or add existing files (e.g. my source files, that should be attached to this document).

# The source code.

My intention is to review the vital parts of the source code separately bit by bit rather than merely paste the entire source into this document.

We shall start with the main function:

int main()

{

// simply print the current version of MySQL++

print\_lib\_version();

...

return 0;

}

Firstly I decided to print out the current version of MySQL++. You can see the function that does that in the following code snippet:

void print\_lib\_version()

{

unsigned int packed\_version = mysqlpp::get\_library\_version();

printf("MySQL version: %d.%d.%d\n\n",

((packed\_version & 0xff0000) >> 16),

((packed\_version & 0xff00) >> 8),

(packed\_version & 0xff));

}

The version is somewhat packed so in order to display it properly you have to decode it by using the corresponding masks.

By the way, you might ask me why I use C like i/o functions instead of C++ i/o streams? Well, there is a certain reason for this and it will become clear a bit later.

Now we shall proceed with our main function:

mysqlpp::Connection db\_connection(false);

if (db\_connection.connect(db\_name, server\_ip, user\_name, password))

{

...

}

else

{

printf("... ERROR: %s\n", db\_connection.error());

}

As you might imagine MySQL++ is all about classes and their objects. So here we create an object of the class mysqlpp::Connection passing ‘false’ to its constructor. In this case it means that the class won’t generate exceptions if an error occurs. If we passed ‘true’ we would have to catch those exceptions by using a try..catch block. Once the object is created we call one of its methods – connect() – passing the required character strings which represent our connection parameters.

Now comes the SQL query itself:

mysqlpp::Query db\_query = db\_connection.query(

"SELECT joe\_andmed.joenimi, \

joe\_andmed.pikkus\_km, \

seire\_jogi\_hydrol\_jaamad.jaama\_nimi \

FROM joe\_andmed \

INNER JOIN seire\_jogi\_hydrol\_jaamad \

ON joe\_andmed.id\_jogi = seire\_jogi\_hydrol\_jaamad.id\_jogi \

WHERE id\_AVK = 'EE2SU8' \

ORDER BY pikkus\_km");

mysqlpp::StoreQueryResult db\_query\_result = db\_query.store();

if (db\_query\_result)

{

...

}

}

else

{

printf("... ERROR: %s\n", db\_connection.error());

}

Here we create an object of the class mysqlpp::Query and initialize it by calling the method query() of the previously created db\_connection object.

The query can be explained the following way:

* We take two tables – joe\_andmed and seire\_jogi\_hydrol\_jaamad – and join them on the condition that id\_jogi column is equal in both tables.
* From the result of this joining we are interested only in those records that have id\_AVK column equal to ‘EE2SU8’ – so we filter the records by using the WHERE clause.
* Keeping in mind our assignment we only need three columns – joenimi, pikkus\_km and jaama\_nimi – so we select them by using the SELECT clause.
* Finally we need to sort our result by the values of pikkus\_km column. For that we use the ORDER BY keyword which sorts the records in ascending order by default.

Once the query is accomplished we have to store its result-set in an object of the class mysqlpp::StoreQueryResult.

Before we proceed forward with the code I would like to talk about the character encodings we are going to deal with in this project.

# Character encodings

Before we start discussing the subject I should tell you the reason why I pay attention to this matter.

At first my management of this task was straight forward. I simply wrote the read records to a file and thought that this would be the end of the mission. But then I found out that the file had incorrect characters among the strings. Those characters were various russian substitutions for estonian letters like ‘ä’, ‘õ’ and so on. The file looked messy. This was definitely not good.

I ought to mention though that if you have estonian system locale set for non-unicode programmes then you are fine…well, partially.

Of course, I wanted to have a generic solution which would look correct regardless of the system locale. So here we go.

There are three major encodings I would like to talk about:

1. ISO-8859-1 [[8]](#_List_of_literature)
2. UTF-8 [[9]](#_List_of_literature)
3. UTF-16 [[10]](#_List_of_literature)

When you create a database you have to specify a number of collations which serve as rules for text sorting and comparing. To put it simply – you have to define character sets your database is going to use to manage the data. Often people put some default stuff like ‘latin1\_swedish\_ci’ collation. In this case the default character set for a database connection will usually be ISO-8859-1. This character-encoding scheme uses an 8-bit code value to encode a single character. It means that the character set can encode 256 different symbols. Although this scheme is able to encode some european specific language letters it’s useless in displaying these letters properly in Windows because Windows wants UTF-16 to handle such kind of situations.

UTF-16 produces a sequence of 16-bit code values. In Microsoft Windows these are known as wide characters, i.e. C/C++ type wchar\_t.

The trick is that if we want to convert ISO-8859-1 to UTF-16 then we have to go through UTF-8.

“UTF-8 <…> is a variable-width encoding that can represent every character in the Unicode character set.” [[9]](#_List_of_literature)

This means that, for example, some characters can be encoded by 8-bit code value and some – by 16-bit code value. Thus we can actually interpret an UTF-8 character string as an array of bytes. In Microsoft Windows UTF-8 is usually known as multibyte character string.

To make things clearer I’m going to show a table which illustrates those encodings:

|  |  |  |  |
| --- | --- | --- | --- |
| Character | ISO-8859-1 | UTF-16 | UTF-8 |
|  |  |  |  |
| A | 0x41 | 0x0041 | 0x41 |
| a | 0x61 | 0x0061 | 0x61 |
| Z | 0x5A | 0x005A | 0x5A |
| z | 0x7A | 0x007A | 0x7A |
| Ä | 0xC4 | 0x00C4 | 0xC3, 0x84 |
| ä | 0xE4 | 0x00E4 | 0xC3, 0xA4 |
| Ü | 0xDC | 0x00DC | 0xC3, 0x9C |
| ü | 0xFC | 0x00FC | 0xC3, 0xBC |

As you can see the basic characters from the Latin alphabet are represented the same way both in ISO-8859-1 and UTF-8 encodings. The differences apear when we start to handle specific characters such as umlauts. You can also see that UTF-16 encoding seems to be quite straight forward always encoding a character with a two-byte code value.

So at this point we can describe our encoding task as follows: ISO-8859-1 → UTF-8 → UTF-16.

# Back to the code

Actually the first phase of the conversion – ISO-8859-1 → UTF-8 – can be done by a database, we just have to execute a corresponding query:

mysqlpp::Query db\_charset\_query = db\_connection.query("SET NAMES 'utf8'");

if (db\_charset\_query.exec())

{

...

}

else

{

printf("... ERROR: %s\n", db\_connection.error());

}

By performing this query – “SET NAMES ‘utf8’” – we tell the database that we want to work with the data using UTF-8 encoding. It is important to put this piece of code right after the connection with the database.

The next step is to do the conversion UTF-8 → UTF-16:

wchar\_t\* convert\_utf8\_to\_utf16(const char \*utf8\_ptr)

{

if (!utf8\_ptr || \*utf8\_ptr == '\0')

{

return 0;

}

int utf16\_length = MultiByteToWideChar(

CP\_UTF8, // code page to use for conversion

MB\_ERR\_INVALID\_CHARS, // Fail on invalid input characters

utf8\_ptr, // pointer to the character string to convert

-1, // process the entire input string

0, // unused, no conversion is done in this step

0 // request the size of destination buffer, in characters

);

if (!utf16\_length)

{

return 0;

}

// utf16\_length includes the size of the terminating null character

wchar\_t \*utf16\_ptr = new wchar\_t[utf16\_length];

if (!utf16\_ptr)

{

return 0;

}

if (!MultiByteToWideChar(CP\_UTF8, 0, utf8\_ptr, -1, utf16\_ptr,

utf16\_length))

{

delete[] utf16\_ptr;

return 0;

}

return utf16\_ptr;

}

The code snippet above should be quite self-explanatory due to the comments I have put there. As a result of this function we have a unicode string in the required encoding. Now we just have to write it to a file properly.

Since we are dealing with unicode strings it’s obvious that a file should also be opened in unicode format:

FILE \*out\_file\_ptr = \_wfopen(L"river\_info.txt", L"w, ccs = UNICODE");

if (out\_file\_ptr)

{

...

}

The most interesting thing in the code above is the second parameter of the \_wfopen function. Apart from the ‘w’ mode we also specify the ‘ccs’ flag which determines the desired encoding of a file.

At this point I must confess that this simplicity – I mean that ‘ccs’ flag – is the main reason why I use here C like i/o functions instead of C++ i/o streams. In the latter case it appears to be too complicated to write unicode data to a stream properly. I saw one solution on a website [[11]](#_List_of_literature) but it was hardly understandable for me what was going on there and I usually avoid using code examples which aren’t clear to me.

So now there is only one thing left – we need to iterate through the queried database records and write them to the opened file:

for (size\_t i = 0; i != db\_query\_result.num\_rows(); ++i)

{

wchar\_t \*river\_name\_ptr = convert\_utf8\_to\_utf16(

db\_query\_result[i]["joenimi"]);

wchar\_t \*river\_length\_ptr = convert\_utf8\_to\_utf16(

db\_query\_result[i]["pikkus\_km"]);

wchar\_t \*station\_name\_ptr = convert\_utf8\_to\_utf16(

db\_query\_result[i]["jaama\_nimi"]);

fwprintf(out\_file\_ptr, L"%15s%15s%30s\n",

(river\_name\_ptr) ? river\_name\_ptr : L"n/a",

(river\_length\_ptr) ? river\_length\_ptr : L"n/a",

(station\_name\_ptr) ? station\_name\_ptr : L"n/a");

if (river\_name\_ptr) delete[] river\_name\_ptr;

if (river\_length\_ptr) delete[] river\_length\_ptr;

if (station\_name\_ptr) delete[] station\_name\_ptr;

}

The first thing that you need to pay attention to is that the fields in a row are accessed by their name instead of their index. This is obviously slower but the code looks a lot clearer.

The second thing is the ternary operator which is used to define the parameters for the fwprintf function. It states as follows: if the convert\_utf8\_to\_utf16 function returned a valid pointer then pass this pointer as a parameter, otherwise if the function for some reason returned null then use a string “n/a” as a parameter.

The code above has a minor performance flaw because the memory gets allocated and released on each iteration. This might look a bit ineffective.

But on the other hand, allocating a large block of memory in the beginning and then using it throughout the whole loop is not a good idea as we don’t know the length of each string beforehand.

I could use std::wstring as the returned value of the convert\_utf8\_to\_utf16 function, i.e. :

std::wstring convert\_utf8\_to\_utf16();

…but I’m not very fond of such implicit stuff.

As for my code, the memory handling is very obvious, which avoids any leaks.

# Conclusion

Although this assignment was quite ordinary I tried to make make it worth your while by providing some useful information regarding the task. I hope I managed.

If you would like to to take a closer look at the subject then you can simply check out the links I’ve given at the end of this document.

# List of literature used

1. MySQL Developer Zone

MySQL 5.0 Manual

[Online]

<http://dev.mysql.com/doc/refman/5.0/en/c-api.html>

(31.03.2013)

1. MySQL++

[Online]

<http://tangentsoft.net/mysql++/>

(31.03.2013)

1. MySQL++ v3.1.0 User Manual

Introduction

<http://tangentsoft.net/mysql++/doc/html/userman/index.html#intro>

(31.03.2013)

1. MySQL Developer Zone

Download MySQL Community Server

[Online]

<http://dev.mysql.com/downloads/mysql/5.1.html>

(31.03.2013)

1. MySQL++

Source Code

[Online]

<http://tangentsoft.net/mysql++/>

(31.03.2013)

1. Bakefile

Download

[Online]

<http://www.bakefile.org/index.html>

(31.03.2013)

1. Visual Studio Express products

[Online]

<http://www.microsoft.com/visualstudio/eng/products/visual-studio-express-products>

(31.03.2013)

1. Wikipedia, The Free Encyclopedia

ISO-8859-1

[Online]

<http://en.wikipedia.org/wiki/ISO/IEC_8859-1>

(31.03.2013)

1. Wikipedia, The Free Encyclopedia

UTF-8

[Online]

<http://en.wikipedia.org/wiki/UTF-8>

(31.03.2013)

1. Wikipedia, The Free Encyclopedia

UTF-16

[Online]

<http://en.wikipedia.org/wiki/UTF-16>

(31.03.2013)

1. CodeGuru

Unicode text file

[Online]

<http://forums.codeguru.com/showthread.php?457106-Unicode-text-file&p=1741409#post1741409>

(31.03.2013)