PostgreSQL Programming Tutorial

OpenInformatics.Net

Database Development Tutorials

ATM Machine

C/C++ application

PostgreSQL® Programming Tutorial using PostgreSQL® Native C API (libpq)

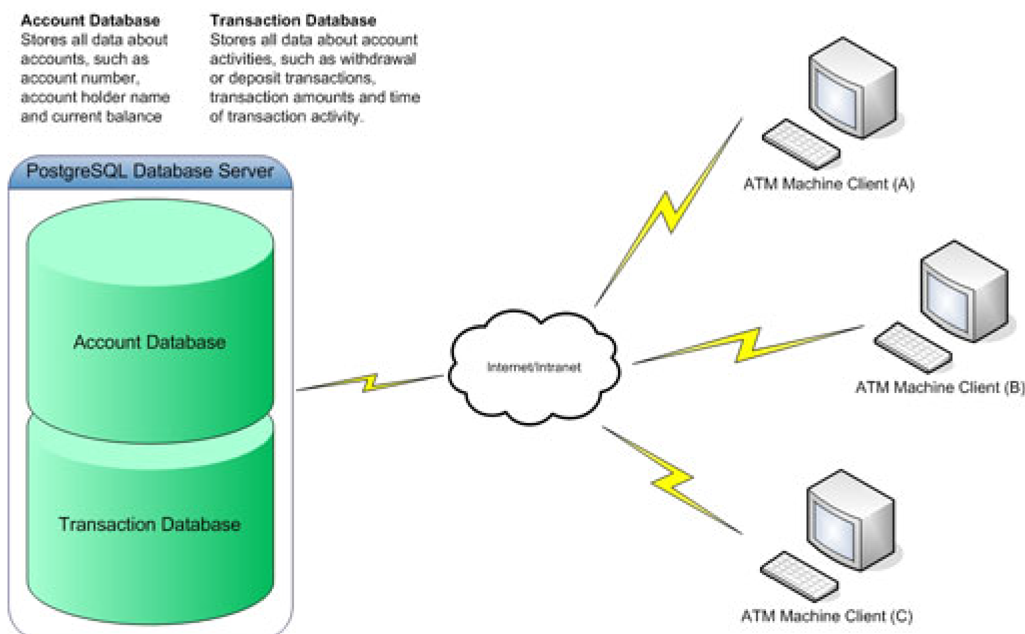
by Zlatan Klebic, contact: sourcecode84@gmail.com
www.openinformatics.net

Note: This tutorial assumes that the reader is familiar with C/C++ programming, SQL and database schema design. In addition, it is assumed that your PostgreSQL® Database Server is configured with a username 'postgres' and a password 'postgres', and is running on your local workstation instead of on another server on the network. This application has been tested using Microsoft® Visual C++ 2003 and Metrowerks® Codewarrior C++.

Note: The code of this application has been written such to help readers better understand PostgreSQL® programming, rather than showing the readers of any clever programming techniques, which certainly wouldn't be appealing to beginner programming.

In the following tutorial we will implement a simple ATM machine software project in C/C++, with the use of PostgreSQL® Server as our Database Management System. The software will be a simple console based application with a set of menu items. The menu items will prompt the user with the following set of options: Create Bank Account, Check existing Bank Account, Activate Bank Account, Deactivate Bank Account, Check Account Balance, Show all Account Transactions, Withdraw money from Account, Deposit money to Account, Exit.

The followig illustration presents the ATM machine application as network-based application, allowing multiple ATM machines at different locations to communicate with the PostgreSQL Database Server over a Wide Area Network.



The ATM Machine sample application consists of an entire set of functions developed specifically for the purpose of managing the data contained in the Account Database and the Transaction Database. These functions cooperate with the option menu items as described earlier. The overall ATM machine software manages data in two database tables, the 'Account Database', and the 'Transaction Database'. The following are the SQL Create Table queries responsible for creating our two tables according to the ATM machine software database schema.

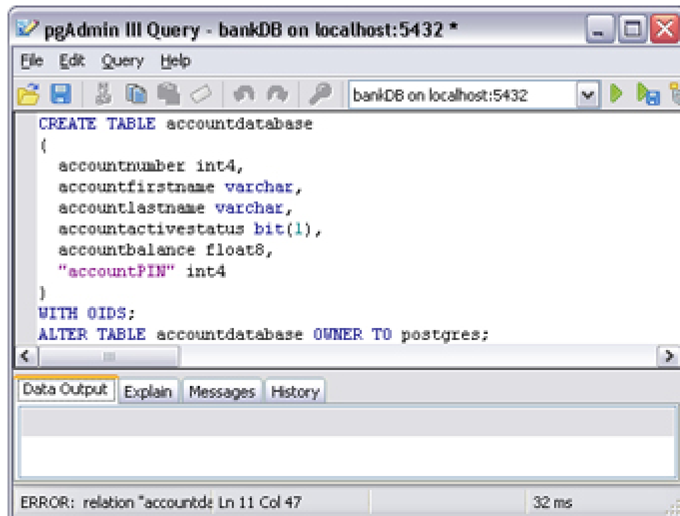
The accountdatabase table, used for the purpose of storing all bank account information of individual customers.

```
CREATE TABLE accountdatabase
(
  accountnumber int4,
  accountfirstname varchar,
  accountlastname varchar,
  accountactivestatus bit(1),
  accountbalance float8,
  "accountPIN" int4
)
WITH OIDS;
ALTER TABLE accountdatabase OWNER TO postgres;
```

The transactiondatabase table, used for the purpose of storing all Bank-to-ATM transaction activities.

```
CREATE TABLE transactiondatabase
(
  transactionnumber int4 NOT NULL,
  transactiontype bit(1),
  transactionamount float8,
  accountnumber int4,
  transactiontime int4,
  transactiondate int4,
  transactionsuccess bit(1)
)
WITH OIDS;
ALTER TABLE transactiondatabase OWNER TO postgres;
```

The given create table queries have been pasted into the Execute SQL Query window in pgAdmin III. Use pgAdmin for all your PostgreSQL® database administration needs. pgAdmin may be downloaded from the pgAdmin Download page.



To use the PostgreSQL® C API on a Windows compiler, two static link libraries are needed, the **libecpg.lib** and **libpq.lib**. During program run-time, dynamic link (.DLL) libraries are needed as well, the **comerr32.dll**, **krb5_32.dll**, **libiconv-2.dll**, **libintl-2.dll**, **libpq.dll**, **ssleay32.dll**, **libeay32.dll**. For compilation, the PostgreSQL® C API (libpq) header files are needed as well, of which is the header file **libpq-fe.h**. All of the needed files may be downloaded from the PostgreSQL® C API Download page.

The following listing declares the most important functions that facilitate the ATM Machine data management (the given function names are mostly self-explanatory but will be described in some detail later in this article):

```
/* Account Database Functions */
void SetAccountActive(string accountNumber);
void SetAccountInactive(string accountNumber);
bool IsAccountActive(string accountNumber);
void DepositToAccount(string accountNumber, double depositAmount);
bool WithdrawFromAccount(string accountNumber, double withdrawalAmount);
double GetAccountBalance(string accountNumber);
bool AccountExists(string accountNumber);
void ShowAllTransactions(string accountNumber);
void ShowAllWithdrawals(string accountNumber);
void ShowAllDeposits(string accountNumber);
string GetAccountFirstName(string accountNumber);
string GetAccountLastName(string accountNumber);
bool CreateBankAccount(string accountNumber,
                       string accountPIN,
                       string firstName,
                       string lastName,
                       string startingBalance);

/* Transaction Database Functions */
void CreateTransaction(string accountNumber);
int GetLastTransactionNumber();
string GetLastTransactionNumberString();
bool TransactionType(string transactionNumber);
void GetLastTransaction(string accountNumber);
void GetLastWithdrawAmount(string accountNumber, double *amount);
void GetLastDepositAmount(string accountNumber, double *amount);
void GetLastTenWithdrawals(string accountNumber);
void StoreWithdrawalTransaction(string transactionNumber,
                                string accountNumber,
                                int transactionSuccess,
                                double transactionAmount,
                                int transactionTime,
                                int transactionDate);
void StoreDepositTransaction(string transactionNumber,
                             string accountNumber,
                             int transactionSuccess,
                             double transactionAmount,
                             int transactionTime,
                             int transactionDate);
```

The provided API functions were designed specifically for use in this simple ATM Machine example. They provide the simplicity of programming in the top layer of the application, as well as scalability and easy design of applications that will further utilize these functions, for slightly similar purposes, not necessarily relating to ATM Machines whatsoever.

Now we will proceed with writing the application code further on, but from the very beginning. The application code is primarily kept in a single file, in order to keep this tutorial as simple as possible, rather than confusing with the usage of multiple source files.

At the beginning of the code, the MIN_REQUIRED_BALANCE is defined, which would later be used as a description defining the minimum value needed in the account balance.

```
#define MIN_REQUIRED_BALANCE 100
```

Other than the numerous included standard header files, most importantly the libpq-fe.h is included as well, which is the main PostgreSQL libpq C API header file that contains core PostgreSQL C API functions such as **PQconnectdb()**, **PQclear()**, **PQstatus()**, **PQgetisnull()**, **PQexec()**, **PQfinish()**, **PQresultStatus()**, and many others. These functions facilitate the core functionality of any C/C++ application utilizing PostgreSQL® through its native libpq C API.

```
#include <conio.h>
#include <stdio.h>
#include <iostream>
#include <cstdlib>
#include <string>
#include <windows.h>
#include <libpq-fe.h>
#include "testfunctions.h"
#include <iomanip>
using namespace std;
```

The function prototypes are written here. Make sure to include the function prototypes that were described earlier as functions facilitating the core functionality of the ATM machine, since they belong to this section of the code. The section where they belong is marked after the function declarations described as 'Maintenance Functions'. The function prototypes are mostly self-explanatory, and will only require the understanding of how the named functionality actually performs its task. This will be explained later on in this article.

```
/* Other Functions */
string doubleToString(double num);

/* Database Connection Declarations */
PGconn *accountDB = PQconnectdb("dbname=bankDB host=localhost user=postgres
password='postgres'");
PGresult *result;
ExecStatusType status;
char *info;

PGconn *transactDB = PQconnectdb("");
PGresult *transactResult;

/* Operations Functions */
void processMainMenu();
void processMaintenanceMenu();

void createUser(string username, string password, string uid);
bool checkUser(string username);
void blockUser(string username);
void deleteUser(string username);
void activateAccount(string uid);
void deactivateAccount(string uid);
void commandSQL(PGconn *conn, string command);

/* Maintenance Functions */
void showMenu();
void showMaintenanceMenu();
void maintenanceMode();
void createTables(PGconn *conn);
void outputDatabaseData();

/* Account Database Functions (as already provided earlier in the tutorial */
…………..
………
…..

/* Transaction Database Functions (as already provided earlier in the tutorial */
…………..
```

```
.........
.....
```

```
/* Menu Selection Functions */
/* Administrative Menu      */
void MenuCreateAccount(void);
void MenuCheckExistingAccount(void);
void MenuActivateBankAccount(void);
void MenuDeactivateBankAccount(void);
/* Customer Menu            */
void MenuCheckAccountBalance(void);
void MenuShowAllAccountTransactions(void);
void MenuWithdrawFromAccount(void);
void MenuDepositToAccount(void);
```

For a better overview of the functionality of the entire set of functions used and developed in this sample application, the ATM application is divided into three layers. Each layer defines the depth of its purpose in terms of what it facilitates. Just as in any other software that relies on the underlying API functions (such as a Win32 application relying on Windows Win32® Services), it provides for the further extension of the entire application; such that other database interfaces may be implemented for the ability to connect to databases other than PostgreSQL®; this is certain as long as the Medium Layer functions (or abstract functions) names and parameters remain untouched, but their procedures adapt to an entirely different, Low Layer API functions, in other words an API of an entirely different database. The following is an illustration of the three layers composing the ATM Application that will help visualize the idea of abstraction and the purpose of layers.

ATM Machine Software Functional Layers

High Level Functions

ATM Machine Menu Functions (end user)

```
void MenuCreateAccount();
void MenuCheckExistingAccount();
void MenuActivateBankAccount();
void MenuDeactivateBankAccount();
void MenuCheckAccountBalance();
void MenuShowAllAccountTransactions();
void MenuWithdrawFromAccount();
void MenuDepositToAccount();
```

Medium Level Functions

ATM Machine Data Management Functions (abstraction)

```
void SetAccountActive();
void SetAccountInactive();
bool IsAccountActive();
void DepositToAccount();
bool WithdrawFromAccount();
double GetAccountBalance();
bool AccountExists();
void ShowAllTransactions();
void ShowAllWithdrawals();
void ShowAllDeposits();
string GetAccountFirstName();
string GetAccountLastName();
bool CreateBankAccount();
void CreateTransaction();
int GetLastTransactionNumber();
string GetLastTransactionNumberString();
bool TransactionType();
void GetLastTransaction();
void GetLastWithdrawAmount();
void GetLastDepositAmount();
void GetLastTenWithdrawals();
void StoreWithdrawalTransaction();
void StoreDepositTransaction();
```

Low Level Functions

PostgreSQL® Server Interface Functions (system API)

```
PGconn* PQconnectdb();
PGresult* PQexec();
char* PQgetvalue();
int PQntuples();
int PQnfields();
void PQclear();
int PQresultStatus();
int PQstatus();
bool PQgetisnull();
```

The function **MenuCreateAccount()** facilitates the Create Account menu item. When called from the main menu of the ATM Machine, it prompts the user for information of which includes the preferred Account Number, Account PIN, Account Holder Name and the Starting Balance of the account. When all of the data is collected, the **CreateBankAccount()** function is called. The **CreateBankAccount()** function will be discussed more in detail later in this tutorial.

```
void MenuCreateAccount()
{
      string acctNum;
      string acctPIN;
      string frstNme;
      string lastNme;
      string strtBlc;
      cout << "CREATE BANK ACCOUNT WIZARD" << endl;
      cout << "Account number:           ";
      cin  >> acctNum;
      cout << "Account PIN number:       ";
      cin  >> acctPIN;
      cout << "Account First Name:       ";
```

```
    cin  >> frstNme;
    cout << "Account Last Name:        ";
    cin  >> lastNme;
    cout << "Account Starting Balance: ";
    cin  >> strtBlc;
    if(CreateBankAccount(acctNum, acctPIN, frstNme, lastNme, strtBlc))
    {
        cout << "ACCOUNT CREATED" << endl;
    }
    else
    {
        cout << "ACCOUNT COULD NOT BE CREATED" << endl;
    }
}
```

Function **MenuCheckExistingAccount()** facilitates the Check Existing Account menu item. When called, it prompts the user for an Account Number, thus then calls the **AccountExists()**. If the accounts exists it further shows details about the accounts. Afterwards it determines whether the given account is active or inactive. All of the functions performing the 'check' and 'get' operations are part of the Middle Layer, while the **MenuCheckExistingAccount()** is part of the High Level functions. High Level functions notoriously depend on the Medium Level functions, while Medium Level functions depend on Low Level functions.

```
void MenuCheckExistingAccount(void)
{
    string acctNum;
    cout << "CHECK EXISTING ACCOUNT WIZARD" << endl;
    cout << "Account Number:              " << endl;
    cin  >> acctNum;
    if(AccountExists(acctNum))
    {
        cout << "ACCOUNT " << acctNum << " FOUND" << endl;
        cout << "First name: " << GetAccountFirstName(acctNum) << endl;
        cout << "Last name:  " << GetAccountLastName(acctNum) << endl;
        cout << "Balance:    $" << GetAccountBalance(acctNum) << endl;
        if(IsAccountActive(acctNum))
        {
            cout << "ACCOUNT IS ACTIVE" << endl;
        }
        else
        {
            cout << "ACCOUNT IS DEACTIVATED" << endl;
        }
    }
    else
    {
        cout << "ACCOUNT " << acctNum << " NOT FOUND" << endl;
    }
}
```

The **MenuActivateBankAccount()** function facilitates the Activate Account menu item. When called it prompts the user to enter the account number to activate. After the account number is entered, it checks whether the account exists using the Medium Layer function **AccountExists()**, and if the account does exist, it then checks if the account is active or not. If the account is inactive, the **SetAccountActive()** function is called, thus activating the inactive account.

```
void MenuActivateBankAccount(void)
{
    string acctNum;
    cout << "ACCOUNT ACTIVATION WIZARD" << endl;
```

```
	cout << "Account Number:           ";
	cin  >> acctNum;
	if(AccountExists(acctNum))
	{
		if(IsAccountActive(acctNum))
		{
			cout << "ACCOUNT " << acctNum << " ALLREADY ACTIVE" << endl;
		}
		else
		{
			SetAccountActive(acctNum);
			cout << "ACCOUNT " << acctNum << " ACTIVATED" << endl;
		}


	}
	else
	{
		cout << "ACCOUNT " << acctNum << " NOT FOUND" << endl;
	}
}
```

The **MenuDeactivateBankAccount()** facilitates the Deactivate Account menu item. It does the exact opposite of what was described in the Activate Account menu, while in addition instead of calling **SetAccountActive()**, it calls the **SetAccountInactive()** function to deactivate the specified account on the prompt.

```
void MenuDeactivateBankAccount(void)
{
	string acctNum;
	cout << "ACCOUNT DEACTIVATION WIZARD" << endl;
	cout << "Account Number:           ";
	cin  >> acctNum;
	if(AccountExists(acctNum))
	{
		if(IsAccountActive(acctNum))
		{
			SetAccountInactive(acctNum);
			cout << "ACCOUNT " << acctNum << " DEACTIVATED" << endl;
		}
		else
		{
			cout << "ACCOUNT " << acctNum << " ALLREADY DEACTIVATED" << endl;
		}
	}
	else
	{
		cout << "ACCOUNT " << acctNum << " NOT FOUND" << endl;
	}
}
```

The **MenuCheckAccountBalance()** function facilitates the Check Account Balance menu item. Upon prompting the user to enter the desired bank account number, it checks whether the account exists, and then calling a set of Medium Layer functions, of which are **GetAccountFirstName()**, **GetAccountLastName()**, **GetAccountBalance()** and **IsAccountActive()**. The function **IsAccountActive()** returns a boolean value, whether the account is activated (True) or deactivated (False).

```
void MenuCheckAccountBalance(void)
{
	string acctNum;
```

```
	cout << "ACCOUNT BALANCE WIZARD" << endl;
	cout << "Account Number:       ";
	cin >> acctNum;
	if(AccountExists(acctNum))
	{
		cout << "ACCOUNT NUMBER:  " << acctNum << endl;
		cout << "ACCOUNT NAME:    " << GetAccountFirstName(acctNum) << endl;
		cout << "                 " << GetAccountLastName(acctNum) << endl;
		cout << "ACCOUNT BALANCE: " << GetAccountBalance(acctNum) << endl;
		cout << "ACCCOUNT STATUS: ";
		if(IsAccountActive(acctNum))
		{
			cout << "ACTIVATED" << endl;
		}
		else
		{
			cout << "DEACTIVATED" << endl;
		}
	}
}
```

The **MenuShowAllAccountTransactions()** is one of the most interesting of the menu items in the ATM Machine menu with the label Show All Transactions. After prompting for the account number, it calls the Medium Layer function, the **ShowAllTransactions()**. This function prints the last 20 activities of the specified accounts, of which include Deposit and Withdrawal activities to the account, as well as information on the amount of money and whether the operation has failed or succeeded.

```
void MenuShowAllAccountTransactions(void)
{
	string acctNum;
	cout << "PRINT ALL ACCOUNT TRANSACTIONS WIZARD" << endl;
	cout << "Account Number: ";
	cin >> acctNum;
	cout << endl;
	ShowAllTransactions(acctNum);
	cout << endl;
}
```

The **MenuWithdrawFromAccount()** facilitates the Withdraw Money from Account menu item. It prompts for the account number, checks if the account exists, shows the current account balance, prompts for the withdrawal amount. If the withdrawal amount is more than the available balance, the withdrawal fails and the failed transaction is recorded as well. If the withdrawal succeeds, the transaction is recorded as succeeded using the **StoreWithdrawalTransaction()** Medium Layer function. The **GetLastTransactionNumber()** function is also used in order to provide the next unique number in sequence due to the fact that each transaction has a unique ID number. This function returns an integer, and thus its return value is incremented by one.

```
void MenuWithdrawFromAccount(void)
{
	string acctNum;
	double amount;
	int transactionNumber;

	cout << "MONEY WITHDRAWAL WIZARD" << endl;
	cout << "Account Number:    ";
	cin >> acctNum;
	if(AccountExists(acctNum))
	{
		cout << "Current Balance:   ";
		cout << GetAccountBalance(acctNum) << endl;
```

```
            cout << "Withdrawal Amount: ";
            cin >> amount;
            if(WithdrawFromAccount(acctNum, amount))
            {
                  cout << "Amount:           " << amount << endl;
                  cout << "Account Balance: " << GetAccountBalance(acctNum) << endl;
                  transactionNumber = GetLastTransactionNumber();
                  transactionNumber++;
                  StoreWithdrawalTransaction(doubleToString(transactionNumber), acctNum, 1,
amount, 1111, 11111111);
            }
            else
            {
                  transactionNumber = GetLastTransactionNumber();
                  transactionNumber++;
                  StoreWithdrawalTransaction(doubleToString(transactionNumber), acctNum, 0,
amount, 1111, 11111111);
                  cout << "Withdrawal failed" << endl;
            }
      }
      else
      {
            cout << "ACCOUNT " << acctNum << " DOES NOT EXIST" << endl;
      }
}
```

The **MenuDepositAccount()** function facilitates the Deposit Money to Account menu item. This function as most others, prompts for the account number, checks the existence of the account using the **AccountExists()** function, prompts for the amount of money to be deposited, checks whether the account is active, and performs the deposit operation upon retrieving the last transaction number using the **GetLastTransactionNumber()** function. Afterwards, the retrieved transaction number is incremented by 1, and the deposited amount and the amount is deposited using the **DepositToAccount()** function. In addition, the **StoreDepositTransaction()** function is called in order to record the account transaction activity, where the newly incremented Transaction ID number is used as well.

```
void MenuDepositToAccount(void)
{
      string acctNum;
      double amount;
      int transactionNumber;

      cout << "MONEY DEPOSIT WIZARD" << endl;
      cout << "Account Number:    ";
      cin >> acctNum;
      if(AccountExists(acctNum))
      {
            cout << "Current Balance:   ";
            cout << GetAccountBalance(acctNum) << endl;
            cout << "Deposit Amount: ";
            cin >> amount;
            if(IsAccountActive(acctNum))
            {
                  transactionNumber = GetLastTransactionNumber();
                  transactionNumber++;
                  DepositToAccount(acctNum, amount);
                  StoreDepositTransaction(doubleToString(transactionNumber), acctNum, 1,
                                          amount, 1111, 11111111);
                  cout << "Deposited Amount: " << amount << endl;
                  cout << "New Balance:      " << GetAccountBalance(acctNum) << endl;
            }
            else
```

```
            {
                  cout << "ACCOUNT " << acctNum << " ACCOUNT IS NOT ACTIVE" << endl;
            }
      }
      else
      {
            cout << "ACCOUNT " << acctNum << " DOES NOT EXIST" << endl;
      }
}
```

The **TransactionType()** function returns FALSE(0) if the transaction was a withdrawal, and TRUE(1) if the transaction was a deposit, this function is part of the Medium Layer in the overall application design. The **TransactionType()** function consists of only one parameter which is the transaction number. The transaction number as mentioned earlier is a unique ID of a transaction. The createCommand string assembles an SQL query that selects the 'transactiontype' column in the transaction database, and receives the actual Boolean value that is stored for the selected transaction record.

```
/* ATM functions */
bool TransactionType(string transactionNumber) // WITHDRAWAL OR DEPOSIT
{
      string createCommand;
 createCommand = "SELECT transactiontype FROM transactiondatabase WHERE transactionnumber = ";
      createCommand += transactionNumber;
      createCommand += " LIMIT 1";
      PGresult *result;
      result = PQexec(accountDB, createCommand.c_str());
      int nrows = PQntuples(result);
      int mynumber;
      string mystring;
      if(nrows > 0)
      {
            mystring = string(PQgetvalue(result,0,0)); // return PQgetvalue in string type
            mynumber = atoi(mystring.c_str());         // First convert string object to C
                                                       // string then convert to integer
            PQclear(result);
            if(mynumber)
            {
                  return 1;
            }
            else
            {
                  return 0;
            }
      }
      return 0;
}
```

The **CreateBankAccount()** is a Medium Layer function which has five parameters. The parameters are the account number, account PIN number, account holder first name as well as last name and the starting balance of the new bank account. The createCommand string assembles a larger SQL query which thus inserts the provided data to the given **CreateBankAccount()** function parameters. After the SQL query is assembled, the **AccountExists()** function is used to test if the account number already exists. If it does exist, a message is displayed stating that the account exists otherwise the assembled SQL Query is submitted to the **commandSQL()** function, and the specified account creation values are inserted into the database.

```
/* Program Functions */
bool CreateBankAccount(string accountNumber, string accountPIN,
                       string firstName, string lastName, string startingBalance)
{
```

```
	string createCommand;
	createCommand = "INSERT INTO accountdatabase VALUES(";
	createCommand += accountNumber;
	createCommand += ",'";
	createCommand += firstName;
	createCommand += "','";
	createCommand += lastName;
	createCommand += "','0',";
	createCommand += startingBalance;
	createCommand += ",";
	createCommand += accountPIN;
	createCommand += ")";
	cout << createCommand << endl;
	if(AccountExists(accountNumber))
	{
		cout << "Account Number already exists." << endl;
		return 0;
	}
	else
	{
		commandSQL(accountDB, createCommand);
		cout << endl;
		cout << "####### ACCOUNT CREATED ####### " << endl;
		cout << endl;
		cout << "Account number:\t\t\t";
		cout << accountNumber << endl;
		cout << "Account Holder Name:\t\t";
		cout << firstName << " " << lastName << endl;
		cout << "Starting balance:\t\t";
		cout << startingBalance << endl;
		cout << "Account PIN:\t\t\t";
		cout << accountPIN;
		cout << endl;
		return 1;
	}
}
```

The **SetAccountActive()** function is a Medium Layer function which sets an account to 'ACTIVE' state, or more precisely, updates the 'accountactivestatus' column to a boolean value of TRUE(1) of the specified account number. This function plays one of the roles where the money withdrawal and account activity is allowed or not. Refer to the given SQL Create Table queries provided earlier to better understand the given columns of the database tables.

```
void SetAccountActive(string accountNumber)
{
	string createCommand;
	createCommand = "UPDATE accountdatabase SET accountactivestatus = '1' WHERE 
accountnumber = ";
	createCommand += accountNumber;
	//cout << createCommand << endl;
	commandSQL(accountDB, createCommand);
}
```

The **SetAccountInactive()** is a Medium Later function which does exactly the opposite of what was described earlier regarding the **SetAccountActive()** function. It updates the 'accountactivestatus' column to a boolean value of FALSE(0). For better understanding of the database schema, refer to the given SQL create table queries given at the beginning of this tutorial.

```
void SetAccountInactive(string accountNumber)
{
	string createCommand;
```

```
	createCommand = "UPDATE accountdatabase SET accountactivestatus = '0' WHERE 
accountnumber = ";
	createCommand += accountNumber;
	//cout << createCommand << endl;
	commandSQL(accountDB, createCommand);
}
```

The **IsAccountActive()** is a Medium Level function which returns TRUE(0) if the specified bank account is active or returns FALSE(1) if the bank account is not activated. Upon receiving its single parameter, the account number, it checks whether that account number exists. If the account exists, an SELECT SQL Query is assembled to select the 'accountactivestatus' column from the account database and check whether it is active or inactive based on the boolean value. Upon receiving the query result, the function returns the boolean value.

```
bool IsAccountActive(string accountNumber)
{
	string createCommand;
	if(AccountExists(accountNumber))
	{
		createCommand = "SELECT accountactivestatus FROM accountdatabase WHERE 
accountnumber = ";
		createCommand += accountNumber;
		createCommand += " LIMIT 1";
		PGresult *result;
		result = PQexec(accountDB, createCommand.c_str());
		int nrows = PQntuples(result);
		int mynumber;
		string mystring;

		mystring = string(PQgetvalue(result,0,0)); //return PQgetvalue in string type
		mynumber = atoi(mystring.c_str());         // First convert string object to C 
string then convert to integer

		if(mynumber)
		{
			return 1;
		}
		else
		{
			return 0;
		}
	}
	else
	{
		return 0;
	}
}
```

The **DepositToAccount()** is a Medium Level function which receives two parameters; the account number and the amount to be deposited. Upon receiving the two parameters it assembles an UPDATE SQL Query which adds the deposit amount to the current balance amount.

```
void DepositToAccount(string accountNumber, double depositAmount)
{
	string createCommand;
	double currentBalance;
	double newBalance;
	PGresult *result;
	currentBalance = GetAccountBalance(accountNumber);
	newBalance = currentBalance + depositAmount;
```

```
        createCommand = "UPDATE accountdatabase SET accountbalance = ";
        createCommand += doubleToString(newBalance);
        createCommand += " WHERE accountnumber = ";
        createCommand += accountNumber;
        result = PQexec(accountDB, createCommand.c_str());
}
```

The **WithdrawFromAccount()** Medium Level function receives two parameters; the account number and the withdrawal amount, as opposed to the **DepositToAccount()** function described previously. The **WithdrawFromAccount()** function assembles an UPDATE SQL Query which deducts the specified withdrawal amount from the current account balance.

```
bool WithdrawFromAccount(string accountNumber, double withdrawalAmount)
{
        string createCommand;
        double currentBalance;
        double newBalance;
        PGresult *result;
        currentBalance = GetAccountBalance(accountNumber);
        newBalance = currentBalance - withdrawalAmount;
        if(currentBalance > MIN_REQUIRED_BALANCE && withdrawalAmount < currentBalance)
        {
                createCommand = "UPDATE accountdatabase SET accountbalance = ";
                createCommand += doubleToString(newBalance);
                createCommand += " WHERE accountnumber = ";
                createCommand += accountNumber;
                result = PQexec(accountDB, createCommand.c_str());
                return 1;
        }
        else
        {
                return 0;
        }
}
```

The **GetAccountBalance()** is a Medium Level function that returns the account balance of a specified account number, which is the single parameter of the function. Upon receiving the parameter value, it assembles a SELECT SQL Query which selects the account balance and returns it as a float data type. This function plays a major role throughout the ATM machine software examples.

```
double GetAccountBalance(string accountNumber)
{
        string createCommand;
        double balance;
        char *s;
        createCommand = "SELECT accountbalance FROM accountdatabase WHERE accountnumber = ";
        createCommand += accountNumber;
        createCommand += " LIMIT 1";
        PGresult *result;
        result = PQexec(accountDB, createCommand.c_str());
        int nrows = PQntuples(result);
        string mystring;
        if(nrows > 0)
        {
                if(!PQgetisnull(result, 0, 0))
                {
                        switch(PQresultStatus(result))
                        {
                        case PGRES_TUPLES_OK:
                                /*************************************************************
                                * To get single value from result that is readable, must use *
```

```
                         * PQgetvalue(result, 0, 0) parameters!!!                     *
                         ***************************************************************/
                         mystring = string(PQgetvalue(result,0,0)); /* return PQgetvalue in
string type */
                         s = const_cast<char*>(mystring.c_str());
                         balance = atof(s);
                         break;
                   }
             }
      }
      PQclear(result);
      return balance;
}
```

The **AccountExists()** is a Medium Level function that returns a Boolean value determining whether the specified account number (as a parameter) is exists. It queries the database for the specified number, if the query result returns the number of rows more than 0, then the account exists, otherwise if the query returns 0 rows, the account number does not exist. This function is commonly throughout the ATM software example to prompt inform the user that the account doesn't exist or that the wrong account number has been entered.

```
bool AccountExists(string accountNumber)
{
      string createCommand;
      bool exists;
      createCommand = "SELECT accountnumber FROM accountdatabase WHERE accountnumber = ";
      createCommand += accountNumber;
      createCommand += " LIMIT 1";
      PGresult *result;
      result = PQexec(accountDB, createCommand.c_str());
      int nrows = PQntuples(result);
      string mystring;
      if(nrows > 0)
      {
            if(!PQgetisnull(result, 0, 0))
            {
                  switch(PQresultStatus(result))
                  {
                  case PGRES_TUPLES_OK:
                        /***************************************************************
                         * To get single value from result that is readable, must use *
                         * PQgetvalue(result, 0, 0) parameters!!!                     *
                         ***************************************************************/
                         mystring = string(PQgetvalue(result,0,0)); /* return PQgetvalue in
string type */
                        /* mynumber = atoi(mystring.c_str());                        */
                         exists = true;
                         break;
                  case PGRES_EMPTY_QUERY:
                         exists = false;
                         break;
                  case PGRES_NONFATAL_ERROR:
                         exists = false;
                         break;
                  }
            }
      }
      else
      {
            exists = false;
      }
```

```
    PQclear(result);
    return exists;
}
```

The **ShowAllTransactions()** is a Medium Level function that prints the last 20 transactions made by the specified account number (as the parameter of this function). The **ShowAllTransactions()** function stores the transaction number, transaction account, transaction amount, transaction success and transaction type into arrays each containing 20 elements.

```
void ShowAllTransactions(string accountNumber)
{
      string createCommand;
      createCommand = "SELECT * FROM transactiondatabase WHERE accountnumber = ";
      createCommand += accountNumber;
      createCommand += "ORDER BY transactionnumber DESC";
      PGresult *result;
      result = PQexec(accountDB, createCommand.c_str());
      int nrows = PQntuples(result);
      string mystring;
      string transactionNumbers[20];
      string transactionAccount[20];
      string transactionAmounts[20];
      string transactionSuccess[20];
      string transactionType[20];
      switch(PQresultStatus(result))
      {
      case PGRES_TUPLES_OK:
            {
                  if(nrows > 20)
                  {
                        nrows = 20;
                  }
                  int r;
                  int nfields = PQnfields(result);
                  printf("Number of rows returned = %d\n", nrows);
                  printf("Number of fields returned = %d\n", nfields);
                  for(r = 0; r < nrows; r++)
                  {
                        transactionNumbers[r] = string(PQgetvalue(result,r, 0));
                        transactionAccount[r] = string(PQgetvalue(result,r, 3));
                        transactionAmounts[r] = string(PQgetvalue(result,r, 2));
                        transactionSuccess[r] = string(PQgetvalue(result,r, 6));
                        transactionType[r]    = string(PQgetvalue(result,r, 1));
                  }
            }
      }
      cout << endl;

      cout << "     Transact"
           << "\t"
           << "Account"
           << "\t"
           << "Amount"
           << "\t\t"
           << "Type"
           << "\t\t"
           << "Success"
           << endl;

      for(int index = 0; index < nrows; index++)
      {
            cout << index << ".\t"
            << transactionNumbers[index]
```

```
            << "\t"
            << transactionAccount[index]
            << "\t$"
            << transactionAmounts[index]
            << "\t\t";
            if(transactionType[index] == "1")
            {
                  cout << "DEPOSIT ";
            }
            else
            {
                  cout << "WITHDRAW";
            }
            cout << "\t";
            if(transactionSuccess[index] == "1")
            {
                  cout << "SUCCESS ";
            }
            else
            {
                  cout << "FAIL    ";
            }
            //<< transactionSuccess[index]
            cout << endl;
      }
      cout << endl;
      PQclear(result);
}
```

The **GetAccountFirstName()** function returns the first name of the account holder as a string type. The data is retrieved based on the specified account number.

```
string GetAccountFirstName(string accountNumber)
{
      string createCommand;
      createCommand = "SELECT accountfirstname FROM accountdatabase WHERE accountnumber = ";
      createCommand += accountNumber;
      PGresult *result;
      result = PQexec(accountDB, createCommand.c_str());
      int nrows = PQntuples(result);
      string firstName;
      if(nrows > 0)
      {
            if(!PQgetisnull(result, 0, 0))
            {
                  switch(PQresultStatus(result))
                  {
                  case PGRES_TUPLES_OK:
                        /************************************************************
                        * To get single value from result that is readable, must use *
                        * PQgetvalue(result, 0, 0) parameters!!!                    *
                        ************************************************************/
                        firstName = string(PQgetvalue(result,0,0)); /* return PQgetvalue in
string type */
                        break;
                  }
            }
      }
      PQclear(result);
      return firstName;
}
```

The **GetAccountLastName()** Medium Level function returns the last name of the account holder as a string object. Its parameter is the account number from which the last name is returned. Upon being called, the function assembles an SQL SELECT Query which returns the last name, from the 'accountlastname' table column. The **GetAccountLastName()** function is commonly used for purposes of providing a more convenient way to retrieve up-to-date information of the account.

```
string GetAccountLastName(string accountNumber)
{
	string createCommand;
	createCommand = "SELECT accountlastname FROM accountdatabase WHERE accountnumber = ";
	createCommand += accountNumber;
	PGresult *result;
	result = PQexec(accountDB, createCommand.c_str());
	int nrows = PQntuples(result);
	string lastName;
	if(nrows > 0)
	{
		if(!PQgetisnull(result, 0, 0))
		{
			switch(PQresultStatus(result))
			{
			case PGRES_TUPLES_OK:
				/************************************************************
				* To get single value from result that is readable, must use *
				* PQgetvalue(result, 0, 0) parameters!!!                     *
				************************************************************/
				lastName = string(PQgetvalue(result,0,0)); /* return PQgetvalue in
/                                                                 string type */
				break;
			}
		}
	}
	PQclear(result);
	return lastName;
}
```

The **GetLastTransactionNumberString()** is a Medium Level function used to retrieve the last transaction number in the transaction database. The last transaction number is the highest number in the entire database, thus making it the last number as the primary key ID of the recorded transactions. This function returns the number value as a string type, rather than an integer. Its primary purpose serves for the assisting in the storing of a new transaction record in the database, while providing a new ID number for the new transaction which is incremented with the retrieved last transaction number.

```
string GetLastTransactionNumberString()
{
	string createCommand;
	createCommand = "SELECT * FROM transactiondatabase ORDER BY transactionnumber DESC LIMIT 
1";
	PGresult *result;
	result = PQexec(accountDB, createCommand.c_str());
	int nrows = PQntuples(result);
	string mystring;
	string strLastTransactionNumber;
	int lastTransactionNumber;
	switch(PQresultStatus(result))
	{
	case PGRES_TUPLES_OK:
		{
```

```
                        int r;
                        int nrows = PQntuples(result);
                        int nfields = PQnfields(result);
                        for(r = 0; r < nrows; r++)
                        {
                              strLastTransactionNumber = string(PQgetvalue(result,r, 0));
                        }
                  }
      }
      lastTransactionNumber = atoi(strLastTransactionNumber.c_str());
      return strLastTransactionNumber;
}
```

The **GetLastTransactionNumber()** is a Medium Level function used to retrieve the last transaction number in the transaction database. The last transaction number is the highest number in the entire database, thus making it the last number as the primary key ID of the recorded transactions. Unlike the **GetLastTransactionNumberString()** function, this function returns the number value as an integer type, rather than an integer. Its primary purpose serves for the assisting in the storing of a new transaction record in the database, while providing a new ID number for the new transaction which is incremented with the retrieved last transaction number.

```
int  GetLastTransactionNumber()
{
      string createCommand;
  createCommand = "SELECT * FROM transactiondatabase ORDER BY transactionnumber DESC LIMIT 1";
      PGresult *result;
      result = PQexec(accountDB, createCommand.c_str());
      int nrows = PQntuples(result);
      string mystring;
      string strLastTransactionNumber;
      int lastTransactionNumber;
      switch(PQresultStatus(result))
      {
      case PGRES_TUPLES_OK:
            {
                  int r;
                  int nrows = PQntuples(result);
                  int nfields = PQnfields(result);
                  for(r = 0; r < nrows; r++)
                  {
                        strLastTransactionNumber = string(PQgetvalue(result,r, 0));
                  }
            }
      }
      lastTransactionNumber = atoi(strLastTransactionNumber.c_str());
      return lastTransactionNumber;
}
```

The **StoreWithdrawalTransaction()** is a Medium Level function which facilitates the storing a withdrawal type transaction into the transaction database. It assembles an INSERT SQL query, which inserts the specified function parameter values into the transaction database.

```
void StoreWithdrawalTransaction(string transactionNumber, string accountNumber,
                                int transactionSuccess, double transactionAmount,
                                int transactionTime, int transactionDate)
{
      string createCommand;
      createCommand = "INSERT INTO transactiondatabase VALUES(";
      createCommand += transactionNumber;
```

```
        createCommand += ",'";
        createCommand += "0',";
        createCommand += doubleToString(transactionAmount);
        createCommand += ", ";
        createCommand += accountNumber;
        createCommand += ", ";
        createCommand += doubleToString(transactionTime);
        createCommand += ", ";
        createCommand += doubleToString(transactionDate);
        createCommand += ", '";
        createCommand += doubleToString(transactionSuccess);
        createCommand += "')";
        commandSQL(accountDB, createCommand.c_str());
}
```

The **StoreDepositTransaction()** is a Medium Level function which facilitates the storing of a deposit type transaction into the transaction database. This function is literarily the same as the **StoreWithdrawalTransaction()** function, except that the 'transactiontype' column in the transaction database is marked with a boolean value of '1' or 'TRUE' to mark the transaction as a deposit; as opposed to the **StoreWithdrawalFunction()** which stores '0' or 'FALSE' to mark the transaction as a withdrawal type.

```
void StoreDepositTransaction(string transactionNumber, string accountNumber,
int transactionSuccess, double transactionAmount,
int transactionTime, int transactionDate)
{
        string createCommand;
        createCommand = "INSERT INTO transactiondatabase VALUES(";
        createCommand += transactionNumber;
        createCommand += ",'";
        createCommand += "1',";
        createCommand += doubleToString(transactionAmount);
        createCommand += ", ";
        createCommand += accountNumber;
        createCommand += ", ";
        createCommand += doubleToString(transactionTime);
        createCommand += ", ";
        createCommand += doubleToString(transactionDate);
        createCommand += ", '";
        createCommand += doubleToString(transactionSuccess);
        createCommand += "')";
        commandSQL(accountDB, createCommand.c_str());
}
```

The **commandSQL()** function is a Medium Level function facilitating the execution of the SQL queries, mainly the INSERT, DELETE and UPDATE queries. It provides an abstract implementation for interfacing to the PostgreSQL® database in order to allow for possibilities of interfacing the ATM software towards other databases such as MySQL®.

```
void commandSQL(PGconn *conn, string command)
{
        cout << "SQL command run..." << endl;
        PGresult *result;
        result = PQexec(conn, command.c_str());
        switch(PQresultStatus(result))
        {
        case PGRES_TUPLES_OK:
                {
                        /***************************************************************
                        * To get single value from result that is readable, must use *
                        * PQgetvalue(result, 0, 0) parameters!!!                     *
```

```
                ************************************************************/
                string mystring;
                int mynumber;
                mystring = string(PQgetvalue(result,0,0)); //return PQgetvalue in string
type
                mynumber = atoi(mystring.c_str());         //return integer
                PQclear(result);
            }
      }
}
```

The **showMenu()** function displays a set of menu items on the screen.

```
void showMenu()
{
      system("cls");
      printf("1. Create Bank Account.           \n");
      printf("2. Check existing Bank Account.   \n");
      printf("3. Activate Bank Account.         \n");
      printf("4. Deactivate Bank Account.       \n");
      printf("5. Check Account Balance          \n");
      printf("6. Show All Account Transactions \n");
      printf("7. Withdraw Money from Account    \n");
      printf("8. Deposit Money to Account       \n");
      printf("9. Exit.\n\n");
      printf("0. Customer Menu.\n\n");
}
```

The **showMaintenanceMenu()** function displays a set of database administrative/maintenance items on the screen.

```
void showMaintenanceMenu()
{
      system("cls");
      printf("1. Create database tables.\n");
      printf("2. Output database tables.\n");
      printf("3. Perform test query.\n\n");
      printf("0. Back to main menu.\n");
}
```

The **maintenanceMode()** function processes the call in order to switch to the maintenance mode **processMaintenanceMenu()** function which displays the maintenance menu.

```
void maintenanceMode()
{
      printf("maintenanceMode()...\n");
      processMaintenanceMenu();
}
```

The **processMainMenu()** function processes the input for the main menu options.

```
void processMainMenu()
{
      char ch = '1';
      do{
            switch(getch())
            {
```

```
                    case '1':
                          MenuCreateAccount();
                          break;
                    case '2':
                          MenuCheckExistingAccount();
                          break;
                    case '3':
                          MenuActivateBankAccount();
                          break;
                    case '4':
                          MenuDeactivateBankAccount();
                          break;
                    case '5':
                          MenuCheckAccountBalance();
                          break;
                    case '6':
                          MenuShowAllAccountTransactions();
                          break;
                    case '7':
                          MenuWithdrawFromAccount();
                          break;
                    case '8':
                          MenuDepositToAccount();
                          break;
                    case '0':
                          system("cls");
                          showMaintenanceMenu();
                          maintenanceMode();
                          break;
                    case 'x':
                    case 'X':
                          ch = 0;
                          break;
                    default:
                          system("cls");
                          showMenu();
                          break;
              }
      }while(ch == '1');
}
```

The **processMaintenanceMenu()** function listens for specific keyboard entries, according to the choices provided in the menu display.

```
void processMaintenanceMenu()
{
      printf("\n");
      char ch = '1';
      do{
            switch(getch())
            {
                  case '1':
                        createTables(accountDB);
                        break;
                  case '2':
                        outputDatabaseData();
                        break;
                  case '3':
                        querySQL(accountDB, "SELECT * FROM accountdatabase");
                        break;
                  case '0':
                        ch = 0;
                        system("cls");
```

```
                        showMenu();
                        cout << endl;
                        break;
                  default:
                        continue;
            }
      }while(ch == '1');
}
```

The **doubleToString()** function converts the double/integer type to string object type. This function proves highly convenient in use with assembling SQL queries containing dynamic values of numeric types, such as seen in **StoreWithdrawalTransaction()** function.

```
string doubleToString(double num)
{
      char buf[80];
      sprintf(buf,"%lf", num);
      string ret = buf;
      ret = ret.substr(0,ret.find_last_not_of('0')+1);
      if(ret[ret.length()-1] == '.')
            ret = ret.substr(0,ret.length()-1);
      return ret;
}
```

ATM Software Screenshot:

```
C:\Documents and Settings\Administrator\Desktop\ATM Machine Tutorial\Console Program\...
C:\Documents and Settings\Administrator\Deskto
1. Create Bank Account.
2. Check existing Bank Account.
3. Activate Bank Account.
4. Deactivate Bank Account.
5. Check Account Balance
6. Show All Account Transactions
7. Withdraw Money from Account
8. Deposit Money to Account
9. Exit.

0. Customer Menu.

PRINT ALL ACCOUNT TRANSACTIONS WIZARD
Account Number: 4545

Number of rows returned = 11
Number of fields returned = 7

     Transact   Account  Amount          Type            Success
0.      18      4545     $2300           WITHDRAW        FAIL
1.      17      4545     $2120.14                WITHDRAW        SUCCESS
2.      16      4545     $400            DEPOSIT         SUCCESS
3.      15      4545     $200            WITHDRAW        SUCCESS
4.      14      4545     $80             WITHDRAW        SUCCESS
5.      13      4545     $5000           WITHDRAW        FAIL
6.      12      4545     $30             DEPOSIT         SUCCESS
7.      11      4545     $980            WITHDRAW        SUCCESS
8.      10      4545     $389.39         WITHDRAW        SUCCESS
9.      9       4545     $89.78          WITHDRAW        SUCCESS
10.     8       4545     $34.89          DEPOSIT         SUCCESS
```