# ПРИЛОЖЕНИЕ А

(*обязательное*)

Листинг программы с комментариями

// ChangeKeyWindow.h

#ifndef CHANGEKEYWINDOW\_H

#define CHANGEKEYWINDOW\_H

#include <QDialog>

namespace Ui {

class ChangeKeyWindow;

}

class ChangeKeyWindow : public QDialog

{

Q\_OBJECT

public:

ChangeKeyWindow(QString, QWidget \*parent = nullptr);

~ChangeKeyWindow();

void ConfigureLineEdits();

QString \*GetStringValue() const;

void SetStringValue(QString \*newStringValue);

uint \*GetIntValue() const;

void SetIntValue(uint \*newIntValue);

qulonglong \*GetLongValue() const;

void SetLongValue(qulonglong \*newLongValue);

QString GetKeyName() const;

private slots:

void on\_okButton\_clicked();

private:

void ConfigureWindow();

QString\* stringValue;

uint\* intValue;

qulonglong\* longValue;

QString keyName;

Ui::ChangeKeyWindow \*ui;

};

#endif // CHANGEKEYWINDOW\_H

// ChangeKeyWindow.cpp

#include "../include/ChangeKeyWindow.h"

#include "ui\_ChangeKeyWindow.h"

#include <QMessageBox>

ChangeKeyWindow::ChangeKeyWindow(QString keyName, QWidget \*parent): QDialog(parent), stringValue(nullptr), intValue(nullptr), longValue(nullptr), keyName(keyName),

ui(new Ui::ChangeKeyWindow)

{

ui->setupUi(this);

ConfigureWindow();

}

ChangeKeyWindow::~ChangeKeyWindow()

{

delete ui;

}

void ChangeKeyWindow::ConfigureWindow()

{

setFixedSize(400, 150);

setWindowModality(Qt::WindowModal);

}

QString ChangeKeyWindow::GetKeyName() const

{

return keyName;

}

void ChangeKeyWindow::ConfigureLineEdits()

{

ui->nameLineEdit->setText(keyName); // set selected key name

if (stringValue) // set selected key value according to it type

{

ui->valueLineEdit->setText(\*stringValue);

}

else if (intValue)

{

ui->valueLineEdit->setText(QString::number(\*intValue));

}

else

{

ui->valueLineEdit->setText(QString::number(\*longValue));

}

}

qulonglong \*ChangeKeyWindow::GetLongValue() const

{

return longValue;

}

void ChangeKeyWindow::SetLongValue(qulonglong \*newLongValue)

{

longValue = newLongValue;

}

uint \*ChangeKeyWindow::GetIntValue() const

{

return intValue;

}

void ChangeKeyWindow::SetIntValue(uint \*newIntValue)

{

intValue = newIntValue;

}

QString \*ChangeKeyWindow::GetStringValue() const

{

return stringValue;

}

void ChangeKeyWindow::SetStringValue(QString \*newStringValue)

{

stringValue = newStringValue;

}

void ChangeKeyWindow::on\_okButton\_clicked()

{

auto newValue = ui->valueLineEdit->text(); // new value for selected key

auto newName = ui->nameLineEdit->text(); // new name for selected key

if (newValue.isEmpty() || newName.isEmpty())

{

QMessageBox::critical(this, "Error!", "Error! New name or value can't be empty!");

return;

}

keyName = newName; // set new name

// set new value for the key according to it type

if (stringValue)

{

\*stringValue = newValue;

}

else if (intValue)

{

bool isOk;

\*intValue = newValue.toUInt(&isOk);

if (!isOk)

{

QMessageBox::warning(this, "Warning!", "DWORD key can't has a string value!"); // if string was entered for dword key

return;

}

}

else

{

bool isOk;

\*longValue = newValue.toULongLong(&isOk);

if (!isOk)

{

QMessageBox::warning(this, "Warning!", "QWORD key can't has a string value!"); // if string was entered for qword key

return;

}

}

close();

}

// CreateGroupWindow.h

#ifndef CREATEGROUPWINDOW\_H

#define CREATEGROUPWINDOW\_H

#include <QDialog>

namespace Ui {

class CreateGroupWindow;

}

class CreateGroupWindow : public QDialog

{

Q\_OBJECT

public:

CreateGroupWindow(QString, QWidget \*parent = nullptr);

~CreateGroupWindow();

QString GetGroupName() const;

private slots:

void on\_okButton\_clicked();

private:

void ConfigureWindow();

QString currentPath;

QString groupName;

Ui::CreateGroupWindow \*ui;

};

#endif

// CreateGroupWindow.cpp

#include "../include/CreateGroupWindow.h"

#include "ui\_CreateGroupWindow.h"

#include "../include/Registry.h"

#include <QMessageBox>

CreateGroupWindow::CreateGroupWindow(QString currentPath, QWidget \*parent): QDialog(parent), currentPath(currentPath), ui(new Ui::CreateGroupWindow)

{

ui->setupUi(this);

ConfigureWindow();

}

CreateGroupWindow::~CreateGroupWindow()

{

delete ui;

}

void CreateGroupWindow::ConfigureWindow()

{

setWindowModality(Qt::ApplicationModal);

setWindowTitle("Create group");

setFixedSize(300, 100);

}

QString CreateGroupWindow::GetGroupName() const

{

return groupName;

}

void CreateGroupWindow::on\_okButton\_clicked()

{

groupName = ui->inputNameLineEdit->text();

if (groupName.isEmpty())

{

QMessageBox::critical(this, "Error!", "Group name can't be empty!");

}

auto pathElements = currentPath.split("\\"); // get all elements of the current path

pathElements.push\_back(groupName); // add new group name for the current path

HKEY baseGroup = Registry().GetGroupHkeyByName(pathElements[0]); // curent group descriptor

pathElements.pop\_front(); // remove "Computer" from the current path

HKEY hKey;

auto pathToNewGroup = pathElements.join("\\");

if (RegCreateKeyEx(baseGroup, (wchar\_t\*)pathToNewGroup.utf16(), 0, NULL, REG\_OPTION\_NON\_VOLATILE, KEY\_WRITE, NULL, &hKey, NULL) != ERROR\_SUCCESS) // attempt to create new group

{

QMessageBox::warning(this, "Warning!", "Couldn't create new group!");

return;

}

close();

}

// CreateKeyWindow.h

#ifndef CREATEKEYWINDOW\_H

#define CREATEKEYWINDOW\_H

#include <QDialog>

#include <QVariant>

namespace Ui {

class CreateKeyWindow;

}

class CreateKeyWindow : public QDialog

{

Q\_OBJECT

public:

CreateKeyWindow(QWidget \*parent = nullptr);

~CreateKeyWindow();

QString GetKeyName() const;

QVariant GetKeyValue() const;

private slots:

void on\_okButton\_clicked();

private:

void ConfigureWindow();

QString keyName;

QVariant keyValue;

Ui::CreateKeyWindow \*ui;

};

#endif

// CreateKeyWindow.cpp

#include "../include/CreateKeyWindow.h"

#include "ui\_CreateKeyWindow.h"

#include <QMessageBox>

#include <QSettings>

CreateKeyWindow::CreateKeyWindow(QWidget \*parent): QDialog(parent), ui(new Ui::CreateKeyWindow)

{

ui->setupUi(this);

ConfigureWindow();

}

CreateKeyWindow::~CreateKeyWindow()

{

delete ui;

}

QString CreateKeyWindow::GetKeyName() const

{

return keyName;

}

QVariant CreateKeyWindow::GetKeyValue() const

{

return keyValue;

}

void CreateKeyWindow::ConfigureWindow()

{

setWindowModality(Qt::ApplicationModal);

setFixedSize(400, 230);

setWindowTitle("Create key");

}

void CreateKeyWindow::on\_okButton\_clicked()

{

keyName = ui->inputNameLineEdit->text(); // name for the new key

keyValue = ui->inputValueLineEdit->text(); // value for the new key

if (!ui->stringKey->isChecked() && !ui->intKey->isChecked() && !ui->longKey->isChecked()) // if type for the new key wasn't selected

{

QMessageBox::critical(this, "Error!", "You should choose key type!");

return;

}

else if (keyName.isEmpty() || keyValue.toString().isEmpty()) // if name of value wasn't entered

{

QMessageBox::critical(this, "Error!", "Error! Name or value can't be empty!");

return;

}

bool isNumber;

if (ui->intKey->isChecked())

{

uint number = keyValue.toUInt(&isNumber);

if (!isNumber) // if string was entered for the dword key

{

QMessageBox::warning(this, "Warning!", "DWORD key can't has a string value!");

return;

}

keyValue = QVariant(number);

}

else if (ui->longKey->isChecked())

{

qulonglong number = keyValue.toULongLong(&isNumber);

if (!isNumber) // if string was entered for the qword key

{

QMessageBox::warning(this, "Warning!", "QWORD key can't be a string!");

return;

}

keyValue = QVariant(number);

}

close();

}

// main.cpp

#include "../include/MainWindow.h"

#include <QApplication>

#include <QSettings>

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

{

QApplication a(argc, argv);

MainWindow w;

w.show();

return a.exec();

}

// MainWindow.h

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

#include <QTreeWidget>

#include "Registry.h"

namespace Ui { class MainWindow; }

class MainWindow : public QMainWindow

{

Q\_OBJECT

public:

MainWindow(QWidget \*parent = nullptr);

~MainWindow();

private slots:

void on\_registryTree\_itemExpanded(QTreeWidgetItem \*item);

void on\_registryTree\_itemClicked(QTreeWidgetItem \*item, int column);

void on\_currentPath\_returnPressed();

void on\_createKeyAction\_triggered();

void on\_groupKeys\_itemClicked(QTableWidgetItem \*item);

void on\_removeKeyAction\_triggered();

void on\_createGroupAction\_triggered();

void on\_removeGroupAction\_triggered();

void on\_groupKeys\_cellDoubleClicked(int row, int column);

private:

void ConfigureMainWIndow();

void ConfigureRegistryTreeWidget();

void ConfigureGroupKeysWidget();

void ConfigureCurrentPathWidget(QString path = "");

void ConfigureCreateButtons(QString path = "");

void ConfigureRemoveKeyButton(QTableWidgetItem\* currentItem = nullptr);

QTreeWidgetItem\* CreateRegistryBranch(QString, QTreeWidgetItem\*);

Ui::MainWindow \*ui;

Registry registry;

};

#endif

// MainWindow.cpp

#include "../include/MainWindow.h"

#include "../include/Registry.h"

#include "../include/CreateGroupWindow.h"

#include "../include/CreateKeyWindow.h"

#include "../include/ChangeKeyWindow.h"

#include "ui\_mainwindow.h"

#include <QMessageBox>

MainWindow::MainWindow(QWidget \*parent): QMainWindow(parent), ui(new Ui::MainWindow)

{

ui->setupUi(this);

ConfigureMainWIndow();

ConfigureRegistryTreeWidget();

ConfigureGroupKeysWidget();

ConfigureCurrentPathWidget();

ConfigureCreateButtons();

ConfigureRemoveKeyButton();

}

MainWindow::~MainWindow()

{

delete ui;

}

void MainWindow::ConfigureMainWIndow()

{

setWindowIcon(QIcon(":/img/window\_icon.png"));

setWindowTitle("Registry Editor");

}

void MainWindow::ConfigureRegistryTreeWidget()

{

showMaximized();

QTreeWidgetItem\* root = new QTreeWidgetItem(QStringList() << "Computer");

root->setIcon(0, QIcon(":/img/computer\_icon.png"));

ui->registryTree->addTopLevelItem(root); // add top level item ("Computer") for the registry tree

QStringList baseGroups =

{

"HKEY\_CLASSES\_ROOT",

"HKEY\_CURRENT\_USER",

"HKEY\_LOCAL\_MACHINE",

"HKEY\_USERS"

};

foreach (auto groupName, baseGroups) // create subgroups for every base group

{

CreateRegistryBranch(groupName, ui->registryTree->topLevelItem(0));

}

}

void MainWindow::ConfigureGroupKeysWidget()

{ // creation table for key name, type and value

ui->groupKeys->setColumnCount(3);

ui->groupKeys->setHorizontalHeaderLabels(QStringList() << "Key" << "Type" << "Value");

ui->groupKeys->verticalHeader()->setVisible(false);

ui->groupKeys->horizontalHeader()->setSectionsClickable(false);

ui->groupKeys->horizontalHeader()->resizeSection(0, 250);

ui->groupKeys->horizontalHeader()->resizeSection(1, 250);

ui->groupKeys->horizontalHeader()->resizeSection(2, 350);

}

void MainWindow::ConfigureCurrentPathWidget(QString path)

{ // line edit which contains path for selected group

ui->currentPath->setText("Computer\\" + path);

ui->currentPath->setReadOnly(true);

}

void MainWindow::ConfigureCreateButtons(QString path)

{

if (path.isEmpty()) // create buttons can be shown only when current path isn't empty

{

ui->createGroupAction->setEnabled(false);

ui->createKeyAction->setEnabled(false);

}

else

{

ui->createGroupAction->setEnabled(true);

ui->createKeyAction->setEnabled(true);

}

}

void MainWindow::ConfigureRemoveKeyButton(QTableWidgetItem \*currentItem)

{

if (!currentItem) // remove key button can be shown only if there is selected table item

{

ui->removeKeyAction->setEnabled(false);

}

else

{

ui->removeKeyAction->setEnabled(true);

}

}

QTreeWidgetItem\* MainWindow::CreateRegistryBranch(QString groupName, QTreeWidgetItem\* parentItem)

{

QTreeWidgetItem\* item = new QTreeWidgetItem(QStringList() << groupName);

QIcon icon(":/img/group\_icon.ico");

item->setIcon(0, icon);

parentItem->addChild(item);

QSettings settings(groupName, QSettings::NativeFormat);

registry.ParseRegistryBranch(item, &settings, &icon);

return item;

}

void MainWindow::on\_registryTree\_itemExpanded(QTreeWidgetItem \*group)

{

Q\_UNUSED(group);

ui->registryTree->resizeColumnToContents(0);

QSettings settings(registry.FindPathForGroup(group), QSettings::NativeFormat);

QIcon icon(":/img/group\_icon.ico");

registry.ParseRegistryBranch(group, &settings, &icon);

}

void MainWindow::on\_registryTree\_itemClicked(QTreeWidgetItem \*item, int column)

{

Q\_UNUSED(column);

auto path = registry.FindPathForGroup(item);

ConfigureCurrentPathWidget(path); // set new path in the line edit

ConfigureCreateButtons(path); // enable create buttons enable

ConfigureRemoveKeyButton(); // disable remove key button

QList<QTableWidgetItem\*> valuesTypes;

auto keysAndValues = registry.CreateGroupKeys(item, valuesTypes); // get map which contains key names and values; get list of table items with key types

QMap<QString, QString>::const\_iterator iterator = keysAndValues.constBegin();

ui->groupKeys->setRowCount(keysAndValues.size()); // set row count for the table

int rowIndex = 0;

while (iterator != keysAndValues.constEnd())

{

// create table item for key name; make it immutable

auto\* keyItem = new QTableWidgetItem(iterator.key());

keyItem->setFlags(keyItem->flags() & ~Qt::ItemIsEditable);

ui->groupKeys->setItem(rowIndex, 0, keyItem);

// create table item for key type; make it immutable

valuesTypes[rowIndex]->setFlags(valuesTypes[rowIndex]->flags() & ~Qt::ItemIsEditable);

ui->groupKeys->setItem(rowIndex, 1, valuesTypes[rowIndex]);

// create table item for key value; make it immutable

auto\* valueItem = new QTableWidgetItem(iterator.value());

valueItem->setFlags(valueItem->flags() & ~Qt::ItemIsEditable);

ui->groupKeys->setItem(rowIndex, 2, valueItem);

// increase row number and iterator

iterator++;

rowIndex++;

}

}

void MainWindow::on\_currentPath\_returnPressed()

{

auto group = registry.FindGroupByPath(ui->currentPath->text(), ui->registryTree->topLevelItem(0)); // get registry tree item with current path

ui->registryTree->setCurrentItem(group); // make it current item

emit ui->registryTree->itemClicked(group, 0); // open it

}

void MainWindow::on\_createKeyAction\_triggered()

{

ConfigureRemoveKeyButton(); // disable remove key button

CreateKeyWindow createKeyWindow(this);

createKeyWindow.show();

createKeyWindow.exec();

auto currentItem = ui->registryTree->currentItem();

auto path = registry.FindPathForGroup(currentItem);

QSettings settings(path, QSettings::NativeFormat);

settings.setValue(createKeyWindow.GetKeyName(), createKeyWindow.GetKeyValue()); // create new key

on\_registryTree\_itemClicked(currentItem, 0); // display new key

}

void MainWindow::on\_groupKeys\_itemClicked(QTableWidgetItem \*item)

{

ConfigureRemoveKeyButton(item); // enable remove key button

}

void MainWindow::on\_removeKeyAction\_triggered()

{

auto keyName = ui->groupKeys->selectedItems()[0]->text(); // get name of selected key

auto currentGroup = ui->registryTree->currentItem();

auto groupName = registry.FindPathForGroup(currentGroup);

QSettings settings(groupName, QSettings::NativeFormat);

settings.remove(keyName); // remove selected key

on\_registryTree\_itemClicked(currentGroup, 0);

}

void MainWindow::on\_createGroupAction\_triggered()

{

CreateGroupWindow createGroupWindow(registry.FindPathForGroup(ui->registryTree->currentItem()), this);

createGroupWindow.show();

createGroupWindow.exec();

auto createdGroupName = createGroupWindow.GetGroupName(); // get name for the new group

if (createdGroupName.isEmpty())

{

return;

}

CreateRegistryBranch(createdGroupName, ui->registryTree->currentItem()); // create subbranch for the new item including it

ConfigureCurrentPathWidget(registry.FindPathForGroup(ui->registryTree->currentItem()) + "\\" + createdGroupName); // set new path

on\_currentPath\_returnPressed(); // open new group

}

void MainWindow::on\_removeGroupAction\_triggered()

{

auto currentPath = registry.FindPathForGroup(ui->registryTree->currentItem());

auto currentPathElements = currentPath.split("\\");

QString baseGroupName = currentPathElements[0]; // get base group name

HKEY baseGroup = registry.GetGroupHkeyByName(baseGroupName); // get base group descriptor

currentPathElements.pop\_front(); // remobe base group name

currentPath = currentPathElements.join("\\"); // create path for the selected group

if (RegDeleteKeyEx(baseGroup, (wchar\_t\*)currentPath.utf16(), KEY\_WOW64\_64KEY, 0) != ERROR\_SUCCESS) // if group wasn't removed

{

QMessageBox::warning(this, "Warning!", "Couldn't delete this group!");

return;

}

int indexToRemove;

auto\* parentOfCurrentItem = ui->registryTree->currentItem()->parent();

for (indexToRemove = 0; indexToRemove < parentOfCurrentItem->childCount(); indexToRemove++)

{

if (ui->registryTree->currentItem()->text(0) == parentOfCurrentItem->child(indexToRemove)->text(0)) // get index of the removed group

{

break;

}

}

ui->registryTree->currentItem()->parent()->takeChild(indexToRemove); // remove group from the tree

}

void MainWindow::on\_groupKeys\_cellDoubleClicked(int row, int column)

{

Q\_UNUSED(row);

Q\_UNUSED(column);

auto currentKeyName = ui->groupKeys->selectedItems()[0]->text(); // get name of selected key

ChangeKeyWindow window(currentKeyName, this);

QString valueType = ui->groupKeys->selectedItems()[1]->text(); // get type of the selected type

// get value of the key and set it in the window

if (valueType == "REG\_SZ")

{

auto keyValue = ui->groupKeys->selectedItems()[2]->text();

window.SetStringValue(&keyValue);

}

else if (valueType == "REG\_DWORD")

{

auto keyValue = ui->groupKeys->selectedItems()[2]->text().toUInt();

window.SetIntValue(&keyValue);

}

else if (valueType == "REG\_QWORD")

{

auto keyValue = ui->groupKeys->selectedItems()[2]->text().toULongLong();

window.SetLongValue(&keyValue);

}

else

{

QMessageBox::warning(this, "Warning!", "Unable to change this key!");

return;

}

window.ConfigureLineEdits();

window.show();

window.exec();

// get key name and path for this key

auto currentItem = ui->registryTree->currentItem();

auto path = registry.FindPathForGroup(currentItem);

QSettings settings(path, QSettings::NativeFormat);

auto keyName = window.GetKeyName();

if (currentKeyName != keyName)

{

on\_removeKeyAction\_triggered();

}

// set new value in registry

if (valueType == "REG\_SZ")

{

settings.setValue(keyName, \*window.GetStringValue());

}

else if (valueType == "REG\_DWORD")

{

settings.setValue(keyName, \*window.GetIntValue());

}

else if (valueType == "REG\_QWORD")

{

settings.setValue(keyName, \*window.GetLongValue());

}

on\_registryTree\_itemClicked(currentItem, 0);

}

// Registry.h

#ifndef REGISTRY\_H

#define REGISTRY\_H

#include <QTreeWidget>

#include <QSettings>

#include <QIcon>

#include <QTableWidgetItem>

#include <QMap>

#include <windows.h>

#include <winreg.h>

class Registry

{

public:

void ParseRegistryBranch(QTreeWidgetItem\*, QSettings\*, QIcon\*);

QString FindPathForGroup(QTreeWidgetItem\*);

QTreeWidgetItem\* FindGroupByPath(QString, QTreeWidgetItem\*);

QMap<QString, QString> CreateGroupKeys(QTreeWidgetItem\*, QList<QTableWidgetItem\*>&);

HKEY GetGroupHkeyByName(QString);

private:

const QString PATH\_SEPARATOR = "\\";

};

#endif // REGISTRY\_H

// Registry.cpp

#include "../include/Registry.h"

void Registry::ParseRegistryBranch(QTreeWidgetItem \*root, QSettings \*settings, QIcon \*icon)

{ // get all group on 2 levels

static int depth = 0;

static const int MAX\_DEPTH = 2;

depth++;

if (root->childCount() > 0)

{

for (int i = 0; i < root->childCount(); i++)

{

auto child = root->child(i);

if (child->childCount() > 0)

{

break;

}

settings->beginGroup(child->text(0));

ParseRegistryBranch(child, settings, icon);

settings->endGroup();

}

}

else if (depth <= MAX\_DEPTH)

{

foreach (const auto& group, settings->childGroups())

{

auto\* item = new QTreeWidgetItem(QStringList() << group);

item->setIcon(0, \*icon);

root->addChild(item);

settings->beginGroup(group);

ParseRegistryBranch(item, settings, icon);

settings->endGroup();

}

}

depth--;

}

QString Registry::FindPathForGroup(QTreeWidgetItem \*group)

{

QStringList reversedPath;

for (; group->text(0) != "Computer"; group = group->parent())

{

reversedPath.append(group->text(0));

}

std::reverse(reversedPath.begin(), reversedPath.end());

return reversedPath.join(PATH\_SEPARATOR);

}

QTreeWidgetItem \*Registry::FindGroupByPath(QString pathForGroup, QTreeWidgetItem\* registryTreeRoot)

{

auto pathElements = pathForGroup.split(PATH\_SEPARATOR);

if (pathElements.last() == "")

{

pathElements.pop\_back();

}

if (registryTreeRoot->text(0) != pathElements[0])

{

return nullptr;

}

pathElements.pop\_front();

foreach (auto pathElement, pathElements)

{

for (int i = 0; i < registryTreeRoot->childCount(); i++)

{

if (pathElement == registryTreeRoot->child(i)->text(0))

{

registryTreeRoot = registryTreeRoot->child(i);

pathElements.pop\_front();

break;

}

}

}

if (pathElements.size() > 0)

{

return nullptr;

}

return registryTreeRoot;

}

QMap<QString, QString> Registry::CreateGroupKeys(QTreeWidgetItem \*selectedGroup, QList<QTableWidgetItem\*>& valuesTypes)

{

QString pathToGroup = FindPathForGroup(selectedGroup); // get path for selected group

QSettings selectedGroupSettings(pathToGroup, QSettings::NativeFormat);

auto childKeys = selectedGroupSettings.childKeys(); // get child keys

QMap<QString, QString> keysAndValues;

foreach (auto key, childKeys)

{

auto keyValue = selectedGroupSettings.value(key);

auto typeName = QString::fromUtf8(keyValue.typeName()).toLower();

if (typeName == "qstringlist")

{ // get string list key

keysAndValues[key] = keyValue.toStringList().join(" ");

valuesTypes.append(new QTableWidgetItem("REG\_MULTI\_SZ"));

}

else if (typeName == "int")

{ // get dword key

keysAndValues[key] = QString::number(keyValue.toUInt());

valuesTypes.append(new QTableWidgetItem("REG\_DWORD"));

}

else if (typeName == "qlonglong")

{ // get qword key

keysAndValues[key] = QString::number(keyValue.toULongLong());

valuesTypes.append(new QTableWidgetItem("REG\_QWORD"));

}

else if (typeName == "qstring")

{ // get string value

keysAndValues[key] = keyValue.toString();

valuesTypes.append(new QTableWidgetItem("REG\_SZ"));

}

else

{ // unable to read this key

keysAndValues[key] = "Can not display this key's value!";

valuesTypes.append(new QTableWidgetItem("Unknown type of registry key!"));

}

}

return keysAndValues;

}

HKEY Registry::GetGroupHkeyByName(QString baseGroupName)

{ // get base group desciptor according to it name

if (baseGroupName == "HKEY\_CLASSES\_ROOT")

{

return HKEY\_CLASSES\_ROOT;

}

else if (baseGroupName == "HKEY\_CURRENT\_USER")

{

return HKEY\_CURRENT\_USER;

}

else if (baseGroupName == "HKEY\_LOCAL\_MACHINE")

{

return HKEY\_LOCAL\_MACHINE;

}

else

{

return HKEY\_USERS;

}

}