Simple encryption with SimpleCrypt

From Qt Wiki

Sometimes, you have to store some information that you may want to protect against casual observation. Think about passwords for remote services, for instance. Strong cryptography is obviously the best solution, but it can be hard to use those in the right way, they tend to pull in more libraries, and frankly, it may just be overkill for the situation.

- Note: additional details are available in SimpleCrypt_algorithm_details
- Note: a usage example can be found here Simple_Crypt_IO_Device

Contents

A word of warning

The class presented in this article does *not* provide strong encryption!

It will protect your data from casual observers, but it will not stand up to dedicated hackers trying to break your secrets. I am not a cryptography expert. Use the code here at your own risk. I am not to be held responsible for any damages that may, directly or indirectly, happen because of your use of this code. Don't say you were not warned.

Introducing SimpleCrypt

The idea of ***SimpleCrypt*** is to provide some basic cryptographic capabilities for simple encryption. Only symmetric encryption is supported (= same key for encryption and decryption required), and there is no API for streaming data in and out like you often see with more advanced cyphers. If you need stronger cryptography, streaming cyphers or asymmetric keys, take a look at QCA (http://delta.affinix.com/qca/) instead.

The ***SimpleCrypt*** class takes a 64 bits key in the form of a quint64. If you use a fixed key build into your program, you can do something like this to initialize your ***SimpleCrypt*** object:

```
SimpleCrypt crypto(Q_UINT64_C(0x0c2ad4a4acb9f023)); //some random number
```

But you can also use the **SimpleCrypt::setKey()** method to set the key.

```
SimpleCrypt crypto();
crypto.setKey(0x0c2ad4a4acb9f023)
```

Selecting a suitable key

To get such random numbers, you might use this service (https://www.random.org/integers/?num=10&min=0&max=65535&col=5&base=16&format=html&rnd=new) and pick four of the 2 byte hexadecimals that you concatenate together as done above. That works better than trying to make up something semi-random yourself. Of course, you can also use other means to get to a quint64 key, such as using some hash of a password and reducing that to 64 bits.

SimpleCrypt in use

Once setup, you can use ***SimpleCrypt*** to actually encrypt or decrypt data. For the encrypt functions, both the *plaintext* that you input as the *cyphertext* that is being outputted can be either a QString (http://doc.qt.io/qt-5/qstring.html#) or a QByteArray (http://doc.qt.io/qt-5/qbytearray.html#). The reverse applies to the decrypt functions. That results in four encrypt methods, and four corresponding decrypt methods:

Encryption methods

```
QString encryptToString(const QString& plaintext);
QString encryptToString(QByteArray plaintext);
QByteArray encryptToByteArray(const QString& plaintext);
QByteArray encryptToByteArray(QByteArray plaintext);
```

Decryption methods

```
QString decryptToString(const QString& plaintext);
QString decryptToString(QByteArray plaintext);
QByteArray decryptToByteArray(const QString& plaintext);
QByteArray decryptToByteArray(QByteArray plaintext);
```

Use with QStrings

```
SimpleCrypt crypto(Q_UINT64_C(0x0c2ad4a4acb9f023)); //some random number
QString testString("Lorem ipsum dolor sit amet, consectetur adipiscing elit.");

//Encryption
QString result = crypto.encryptToString(testString);

//Decryption
QString decrypted = crypto.decryptToString(result);
qDebug() << testString << endl << result << endl << decrypted;
```

The code above encrypts a **QString** to a **QString**, and then decrypts that string again to a new **QString**.

Use with binary data

If you want to encrypt your own binary data, you might do something like this:

```
//setup our objects
 SimpleCrypt crypto(Q_UINT64_C(0x0c2ad4a4acb9f023)); //some random number
 crypto.setCompressionMode(SimpleCrypt::CompressionAlways); //always compress the data, see section belov
 crypto.setIntegrityProtectionMode(SimpleCrypt::ProtectionHash); //properly protect the integrity of the
 QBuffer buffer;
 buffer.open(QIODevice::WriteOnly);
 QDataStream s(&buffer);
 //stream the data into our buffer
 s.setVersion(QDataStream::Qt_4_7);
 s << myString; //stream in a string
 s << myUint16; //stream in an unsigned integer
 s << myImage; //stream in an image
 s << someMoreData;
 // … etc.

 QByteArray myCypherText = crypto.encryptToByteArray(buffer.data());
 if (crypto.lastError() == SimpleCrypt::ErrorNoError) {
   // do something relevant with the cyphertext, such as storing it or sending it over a socket to anothe
 }
 buffer.close();
```

Note the use of the data protection feature. I would recommend that you use at least the ProtectionChecksum level of protection (enabled by default) if you work with binary data, as streaming invalid binary data into your program can lead to big problems.

On the other end, you can now decrypt with this snippet:

```
 SimpleCrypt crypto(Q_UINT64_C(0x0c2ad4a4acb9f023)); //same random number: key should match encryption ke
 QByteArray plaintext = crypto.decryptToByteArray(myCypherText);
 if (!crypto.lastError() == SimpleCrypt::ErrorNoError) {
   // check why we have an error, use the error code from crypto.lastError() for that
 return;
 }

 QBuffer buffer(&plaintext);
 buffer.open(QIODevice::ReadOnly);
 QDataStream s(&buffer);
 s.setVersion(QDataStream::Qt_4_7);
 s >> myString; //stream in a string
 s >> myUint16; //stream in an unsigned integer
 s >> myImage; //stream in an image
 s >> someMoreData;
 //etc.
 buffer.close();
```

The code above might crash if the integrity of the data is not guaranteed, for instance by the use of a wrong password. Using the highest level of protection is probably OK here, as we have quite a big blob of data already, so adding the overhead of adding a 20 byte SHA1 cryptographic hash is justifiable.

Compression

For long strings, it can be beneficial to use compression before encryption. Not only does the length of the string decrease, the input will be a bit more random-looking too, resulting in a harder to break cyphertext (at least, in principle, see the warning at the top of this page.) However, for short strings, applying compression can lead to an *increase* in the string length.

SimpleCrypt supports three modes for compression before encryption. You can force to always use compression, to never use compression, or to automatically choose the shortest version (either the compressed one or the uncompressed one). The last option is the default option. Note that setting this only affects *encryption*. For decryption, SimpleCrypt automatically decompresses the data if compression was used for the encryption.

The code

The source code of SimpleCrypt consists of two files: the header (.h) and the source (.cpp).

simplecrypt.h

```
/*
Copyright (c) 2011, Andre Somers
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.
    * Neither the name of the Rathenau Instituut, Andre Somers nor the
      names of its contributors may be used to endorse or promote products
      derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL ANDRE SOMERS BE LIABLE FOR ANY
DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR #######; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

#ifndef SIMPLECRYPT_H
#define SIMPLECRYPT_H
#include <QString>
#include <QVector>
#include <QFlags>

/**
  @short Simple encryption and decryption of strings and byte arrays

  This class provides a simple implementation of encryption and decryption
  of strings and byte arrays.

  @warning The encryption provided by this class is NOT strong encryption. It may
  help to shield things from curious eyes, but it will NOT stand up to someone
  determined to break the encryption. Don't say you were not warned.

  The class uses a 64 bit key. Simply create an instance of the class, set the key,
  and use the encryptToString() method to calculate an encrypted version of the input string.
  To decrypt that string again, use an instance of SimpleCrypt initialized with
  the same key, and call the decryptToString() method with the encrypted string. If the key
  matches, the decrypted version of the string will be returned again.

  If you do not provide a key, or if something else is wrong, the encryption and
  decryption function will return an empty string or will return a string containing nonsense.
  lastError() will return a value indicating if the method was succesful, and if not, why not.

  SimpleCrypt is prepared for the case that the encryption and decryption
  algorithm is changed in a later version, by prepending a version identifier to the cypertext.
  */
```

```
class SimpleCrypt
{
public:
    /**
      CompressionMode describes if compression will be applied to the data to be
      encrypted.
      */
    enum CompressionMode {
        CompressionAuto,    /*!< Only apply compression if that results in a shorter plaintext. */
        CompressionAlways,  /*!< Always apply compression. Note that for short inputs, a compression may 
        CompressionNever    /*!< Never apply compression. */
    };
    /**
      IntegrityProtectionMode describes measures taken to make it possible to detect problems with the da
      or wrong decryption keys.

      Measures involve adding a checksum or a cryptograhpic hash to the data to be encrypted. This
      increases the length of the resulting cypertext, but makes it possible to check if the plaintext
      appears to be valid after decryption.
    */
    enum IntegrityProtectionMode {
        ProtectionNone,    /*!< The integerity of the encrypted data is not protected. It is not really p
        ProtectionChecksum,/*!< A simple checksum is used to verify that the data is in order. If not, an
        ProtectionHash     /*!< A cryptographic hash is used to verify the integrity of the data. This me
    };
    /**
      Error describes the type of error that occured.
      */
    enum Error {
        ErrorNoError,         /*!< No error occurred. */
        ErrorNoKeySet,        /*!< No key was set. You can not encrypt or decrypt without a valid key. */
        ErrorUnknownVersion,  /*!< The version of this data is unknown, or the data is otherwise not vali
        ErrorIntegrityFailed, /*!< The integrity check of the data failed. Perhaps the wrong key was used
    };

    /**
      Constructor.

      Constructs a SimpleCrypt instance without a valid key set on it.
     */
    SimpleCrypt();
    /**
      Constructor.

      Constructs a SimpleCrypt instance and initializes it with the given @arg key.
     */
    explicit SimpleCrypt(quint64 key);

    /**
      (Re-) initializes the key with the given @arg key.
      */
    void setKey(quint64 key);
    /**
      Returns true if SimpleCrypt has been initialized with a key.
      */
    bool hasKey() const {return !m_keyParts.isEmpty();}

    /**
      Sets the compression mode to use when encrypting data. The default mode is Auto.

      Note that decryption is not influenced by this mode, as the decryption recognizes
      what mode was used when encrypting.
      */
    void setCompressionMode(CompressionMode mode) {m_compressionMode = mode;}
    /**
      Returns the CompressionMode that is currently in use.
      */
    CompressionMode compressionMode() const {return m_compressionMode;}

    /**
      Sets the integrity mode to use when encrypting data. The default mode is Checksum.

      Note that decryption is not influenced by this mode, as the decryption recognizes
      what mode was used when encrypting.
      */
    void setIntegrityProtectionMode(IntegrityProtectionMode mode) {m_protectionMode = mode;}
    /**
      Returns the IntegrityProtectionMode that is currently in use.
      */
    IntegrityProtectionMode integrityProtectionMode() const {return m_protectionMode;}

    /**
      Returns the last error that occurred.
      */
    Error lastError() const {return m_lastError;}

    /**
      Encrypts the @arg plaintext string with the key the class was initialized with, and returns
      a cyphertext the result. The result is a base64 encoded version of the binary array that is the
      actual result of the string, so it can be stored easily in a text format.
      */
    QString encryptToString(const QString& plaintext) ;
    /**
      Encrypts the @arg plaintext QByteArray with the key the class was initialized with, and returns
      a cyphertext the result. The result is a base64 encoded version of the binary array that is the
      actual result of the encryption, so it can be stored easily in a text format.
      */
    QString encryptToString(QByteArray plaintext) ;
    /**
      Encrypts the @arg plaintext string with the key the class was initialized with, and returns
      a binary cyphertext in a QByteArray the result.

      This method returns a byte array, that is useable for storing a binary format. If you need
      a string you can store in a text file, use encryptToString() instead.
      */
    QByteArray encryptToByteArray(const QString& plaintext) ;
    /**
      Encrypts the @arg plaintext QByteArray with the key the class was initialized with, and returns
      a binary cyphertext in a QByteArray the result.

      This method returns a byte array, that is useable for storing a binary format. If you need
      a string you can store in a text file, use encryptToString() instead.
      */
    QByteArray encryptToByteArray(QByteArray plaintext) ;

    /**
      Decrypts a cyphertext string encrypted with this class with the set key back to the
      plain text version.

      If an error occured, such as non-matching keys between encryption and decryption,
      an empty string or a string containing nonsense may be returned.
      */
    QString decryptToString(const QString& cyphertext) ;
    /**
      Decrypts a cyphertext string encrypted with this class with the set key back to the
      plain text version.

      If an error occured, such as non-matching keys between encryption and decryption,
      an empty string or a string containing nonsense may be returned.
      */
    QByteArray decryptToByteArray(const QString& cyphertext) ;
    /**
      Decrypts a cyphertext binary encrypted with this class with the set key back to the
      plain text version.

      If an error occured, such as non-matching keys between encryption and decryption,
      an empty string or a string containing nonsense may be returned.
      */
    QString decryptToString(QByteArray cypher) ;
    /**
      Decrypts a cyphertext binary encrypted with this class with the set key back to the
      plain text version.

      If an error occured, such as non-matching keys between encryption and decryption,
      an empty string or a string containing nonsense may be returned.
      */
    QByteArray decryptToByteArray(QByteArray cypher) ;

    //enum to describe options that have been used for the encryption. Currently only one, but
    //that only leaves room for future extensions like adding a cryptographic hash...
    enum CryptoFlag{CryptoFlagNone = 0,
                    CryptoFlagCompression = 0x01,
                    CryptoFlagChecksum = 0x02,
                    CryptoFlagHash = 0x04
                   };
    Q_DECLARE_FLAGS(CryptoFlags, CryptoFlag);
private:

    void splitKey();

    quint64 m_key;
    QVector<char> m_keyParts;
    CompressionMode m_compressionMode;
    IntegrityProtectionMode m_protectionMode;
    Error m_lastError;
};
Q_DECLARE_OPERATORS_FOR_FLAGS(SimpleCrypt::CryptoFlags)

#endif // SimpleCrypt_H
```

simplecrypt.cpp

```
/*
Copyright (c) 2011, Andre Somers
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.
    * Neither the name of the Rathenau Instituut, Andre Somers nor the
      names of its contributors may be used to endorse or promote products
      derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL ANDRE SOMERS BE LIABLE FOR ANY
DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR #######; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/
#include "simplecrypt.h"
#include <QByteArray>
#include <QtDebug>
#include <QtGlobal>
#include <QDateTime>
#include <QCryptographicHash>
#include <QDataStream>

SimpleCrypt::SimpleCrypt():
    m_key(0),
    m_compressionMode(CompressionAuto),
    m_protectionMode(ProtectionChecksum),
    m_lastError(ErrorNoError)
{
    qsrand(uint(QDateTime::currentMSecsSinceEpoch() & 0xFFFF));
}

SimpleCrypt::SimpleCrypt(quint64 key):
    m_key(key),
    m_compressionMode(CompressionAuto),
    m_protectionMode(ProtectionChecksum),
    m_lastError(ErrorNoError)
{
    qsrand(uint(QDateTime::currentMSecsSinceEpoch() & 0xFFFF));
    splitKey();
}

void SimpleCrypt::setKey(quint64 key)
{
    m_key = key;
    splitKey();
}

void SimpleCrypt::splitKey()
{
    m_keyParts.clear();
    m_keyParts.resize(8);
    for (int i=0;i<8;i++) {
        quint64 part = m_key;
        for (int j=i; j>0; j--)
            part = part >> 8;
        part = part & 0xff;
        m_keyParts[i] = static_cast<char>(part);
    }
}

QByteArray SimpleCrypt::encryptToByteArray(const QString& plaintext)
{
    QByteArray plaintextArray = plaintext.toUtf8();
    return encryptToByteArray(plaintextArray);
}

QByteArray SimpleCrypt::encryptToByteArray(QByteArray plaintext)
{
    if (m_keyParts.isEmpty()) {
        qWarning() << "No key set.";
        m_lastError = ErrorNoKeySet;
        return QByteArray();
    }


    QByteArray ba = plaintext;

    CryptoFlags flags = CryptoFlagNone;
    if (m_compressionMode == CompressionAlways) {
        ba = qCompress(ba, 9); //maximum compression
        flags |= CryptoFlagCompression;
    } else if (m_compressionMode == CompressionAuto) {
        QByteArray compressed = qCompress(ba, 9);
        if (compressed.count() < ba.count()) {
            ba = compressed;
            flags |= CryptoFlagCompression;
        }
    }

    QByteArray integrityProtection;
    if (m_protectionMode == ProtectionChecksum) {
        flags |= CryptoFlagChecksum;
        QDataStream s(&integrityProtection, QIODevice::WriteOnly);
        s << qChecksum(ba.constData(), ba.length());
    } else if (m_protectionMode == ProtectionHash) {
        flags |= CryptoFlagHash;
        QCryptographicHash hash(QCryptographicHash::Sha1);
        hash.addData(ba);

        integrityProtection += hash.result();
    }

    //prepend a random char to the string
    char randomChar = char(qrand() & 0xFF);
    ba = randomChar + integrityProtection + ba;

    int pos(0);
    char lastChar(0);

    int cnt = ba.count();

    while (pos < cnt) {
        ba[pos] = ba.at(pos) ^ m_keyParts.at(pos % 8) ^ lastChar;
        lastChar = ba.at(pos);
        ++pos;
    }

    QByteArray resultArray;
    resultArray.append(char(0x03));  //version for future updates to algorithm
    resultArray.append(char(flags)); //encryption flags
    resultArray.append(ba);

    m_lastError = ErrorNoError;
    return resultArray;
}

QString SimpleCrypt::encryptToString(const QString& plaintext)
{
    QByteArray plaintextArray = plaintext.toUtf8();
    QByteArray cypher = encryptToByteArray(plaintextArray);
    QString cypherString = QString::fromLatin1(cypher.toBase64());
    return cypherString;
}

QString SimpleCrypt::encryptToString(QByteArray plaintext)
{
    QByteArray cypher = encryptToByteArray(plaintext);
    QString cypherString = QString::fromLatin1(cypher.toBase64());
    return cypherString;
}

QString SimpleCrypt::decryptToString(const QString &cyphertext)
{
    QByteArray cyphertextArray = QByteArray::fromBase64(cyphertext.toLatin1());
    QByteArray plaintextArray = decryptToByteArray(cyphertextArray);
    QString plaintext = QString::fromUtf8(plaintextArray, plaintextArray.size());

    return plaintext;
}

QString SimpleCrypt::decryptToString(QByteArray cypher)
{
    QByteArray ba = decryptToByteArray(cypher);
    QString plaintext = QString::fromUtf8(ba, ba.size());

    return plaintext;
}

QByteArray SimpleCrypt::decryptToByteArray(const QString& cyphertext)
{
    QByteArray cyphertextArray = QByteArray::fromBase64(cyphertext.toLatin1());
    QByteArray ba = decryptToByteArray(cyphertextArray);

    return ba;
}

QByteArray SimpleCrypt::decryptToByteArray(QByteArray cypher)
{
    if (m_keyParts.isEmpty()) {
        qWarning() << "No key set.";
        m_lastError = ErrorNoKeySet;
        return QByteArray();
    }

    QByteArray ba = cypher;

    if( cypher.count() < 3 )
        return QByteArray();

    char version = ba.at(0);

    if (version !=3) {  //we only work with version 3
        m_lastError = ErrorUnknownVersion;
        qWarning() << "Invalid version or not a cyphertext.";
        return QByteArray();
    }

    CryptoFlags flags = CryptoFlags(ba.at(1));

    ba = ba.mid(2);
    int pos(0);
    int cnt(ba.count());
    char lastChar = 0;

    while (pos < cnt) {
        char currentChar = ba[pos];
        ba[pos] = ba.at(pos) ^ lastChar ^ m_keyParts.at(pos % 8);
        lastChar = currentChar;
        ++pos;
    }

    ba = ba.mid(1); //chop off the random number at the start

    bool integrityOk(true);
    if (flags.testFlag(CryptoFlagChecksum)) {
        if (ba.length() < 2) {
            m_lastError = ErrorIntegrityFailed;
            return QByteArray();
        }
        quint16 storedChecksum;
        {
            QDataStream s(&ba, QIODevice::ReadOnly);
            s >> storedChecksum;
        }
        ba = ba.mid(2);
        quint16 checksum = qChecksum(ba.constData(), ba.length());
        integrityOk = (checksum == storedChecksum);
    } else if (flags.testFlag(CryptoFlagHash)) {
        if (ba.length() < 20) {
            m_lastError = ErrorIntegrityFailed;
            return QByteArray();
        }
        QByteArray storedHash = ba.left(20);
        ba = ba.mid(20);
        QCryptographicHash hash(QCryptographicHash::Sha1);
        hash.addData(ba);
        integrityOk = (hash.result() == storedHash);
    }

    if (!integrityOk) {
        m_lastError = ErrorIntegrityFailed;
        return QByteArray();
    }

    if (flags.testFlag(CryptoFlagCompression))
        ba = qUncompress(ba);

    m_lastError = ErrorNoError;
    return ba;
}
```

You can copy/paste the code above into your own **simplecrypt.h** and **simplecrypt.cpp** files to use it.

Details on the algorithm

The algorithm and data format used in this class, have been detailed on a separate page. On that page, you can find byte-for-byte descriptions of how the cyphertext is constructed from the plain text.

Future extensions

I, or somebody else, may choose to extend SimpleCrypt in the future, or fix a weakness of some kind. To make that possible without losing the option to read your older encrypted data, the generated cyphertext contains a version number. The idea is that future versions of this code will have a higher version number, but will keep supporting the decryption of cyphertexts with a lower version number. The current version number is 2. Note that these version numbers need not coincide with code version numbers, as a change in the code does not always result in a change in the actual encryption.

Versions

The latest version of the code is 3.1

- Version 3.1
  - The code is now compatible with Qt5 and higher
- Version 3
  - Fixed embarrassing mistake with only using 4 out of 8 of the key bytes. Thanks to Gerolf for noticing!
- Version 2
  - Renamed flags to avoid clashes
  - Upped version number to 2, but did not retain backwards compatibility
  - Added optional data integrity protection using either a checksum or a cryptographic hash
  - Added error reporting (making is necessary to make the encryption and decryption functions non-const)
- Version 1
  - This marks the first public release.

Retrieved from "http://wiki.qt.io/index.php?title=Simple_encryption_with_SimpleCrypt&oldid=18008"
Category: Snippets

- This page was last modified on 16 June 2015, at 10:30.
- This page has been accessed 11,570 times.