Random Numbers and Cryptographic Hashes

Generating random numbers and computing cryptographic hashes (digests).

applied informatics

Overview

- Random number generation and random streams
- Cryptographic Hashes/Digests

Random Numbers

- POCO includes a pseudo random number generator (PRNG), using a nonlinear additive feedback algorithm with 256 bits of state information and a period of up to 2^{69}.
- The PRNG can generate 31 bit pseudo random numbers.
- It can generate UInt32, char, bool, float and double random values.
- In addition, there is a stream class providing a stream of random bytes (using /dev/random or the Windows crypto APIs).

The Random Class

- POCO::Random implements a PRNG.
- #include "Poco/Random.h"
- void seed(Poco::UInt32 seed)
  seeds the PRNG using the given seed.
- void seed()
  seeds the PRNG using random data (from RandomInputStream)
- The constructor only seeds the PRNG using the current date and time. For better seeding, explicitely call one of the seed() methods.

The Random Class (cont'd)

- UInt32 next()
  returns pseudo random number in the range [0, 2^{31})
- UInt32 next(UInt32 n)
  returns a pseudo random number in the range [0, n)
- char nextChar()
  returns a pseudo random character
- bool nextBool()
  return a pseudo random boolean
- float nextFloat(), double nextDouble()
  return a pseudo random floating point value in the range [0, 1]

The RandomInputStream Class

- Poco::RandomInputStream is an istream that produces an endless sequence of random bytes.
- #include "Poco/RandomStream.h"
- The random bytes are taken from /dev/random, or the Windows cryptography API (if neither is available, RandomInputStream creates its own random data).

```
#include "Poco/Random.h"
#include "Poco/RandomStream.h"
#include <iostream>

using Poco::Random;
using Poco::RandomInputStream;

int main(int argc, char** argv)
{
	Random rnd;
	rnd.seed();

	std::cout << "Random integer: " << rnd.next() << std::endl;
	std::cout << "Random digit: "   << rnd.next(10) << std::endl;
	std::cout << "Random char: " << rnd.nextChar() << std::endl;
	std::cout << "Random bool: " << rnd.nextBool() << std::endl;
	std::cout << "Random double: " << rnd.nextDouble() << std::endl;

	RandomInputStream ri;
	std::string rs;
	ri >> rs;

	return 0;
}
```

Cryptographic Hashes

[...] A cryptographic hash function is a hash function with certain additional security properties to make it suitable for use as a primitive in various information security applications, such as authentication and message integrity. A hash function takes a long string (or message) of any length as input and produces a fixed length string as output, sometimes termed a message digest or a digital fingerprint.

Wikipedia

Cryptographic Hashes (cont'd)

- POCO provides implementations of some widely used cryptographic hash functions:
  MD2, MD4, MD5 and SHA1
- An implementation of the HMAC message authentication code algorithm (RFC 2104) is available as well.
- The implementations of all hash functions, and HMAC, are subclasses of the DigestEngine class.
- If you want to implement your own hash functions, it's a good idea to derive them from DigestEngine also.

The DigestEngine Class

- Poco::DigestEngine defines the common interface for all message digest algorithm implementations.
- #include "Poco/DigestEngine.h"
- The length of a message digest depends on the actual algorithm.
- So in POCO, a Digest is just a std::vector<unsigned char>.
- To compute a digest, you repeatedly call on of the DigestEngine's update() methods with your data.
- When all data has been passed to the DigestEngine, you call the digest() method to obtain the Digest for your data.

The DigestEngine Class (cont'd)

- void update(const void* data, unsigned length)
  updates the digest with a block of data
- void update(char data)
  updates the digest with a byte of data
- void update(const std::string& data)
  updates the digest with a string of data
- const Digest& digest()
  finishes digest computation and returns a reference to the digest
- For the other methods, please see the reference documentation.

Hash Algorithm Implementations

- The following implementations of cryptographic hash algorithms are available in POCO:
  - Poco::MD2Engine (#include "Poco/MD2Engine.h)
  - Poco::MD4Engine (#include "Poco/MD4Engine.h)
  - Poco::MD5Engine (#include "Poco/MD5Engine.h)
  - Poco::SHA1Engine (#include "Poco/SHA1Engine.h)
  - Poco::HMACEngine (#include "Poco/HMACEngine.h)
    This is actually a class template that must be instantiated with a DigestEngine subclass.

```
#include "Poco/HMACEngine.h"
#include "Poco/SHA1Engine.h"

using Poco::DigestEngine;
using Poco::HMACEngine;
using Poco::SHA1Engine;

int main(int argc, char** argv)
{
    std::string message1("This is a top-secret message.");
    std::string message2("Don't tell anyone!");
    std::string passphrase("s3cr3t"); // HMAC needs a passphrase

    HMACEngine<SHA1Engine> hmac(passphrase); // we'll compute a HMAC-SHA1
    hmac.update(message1);
    hmac.update(message2);

    const DigestEngine::Digest& digest = hmac.digest();
        // finish HMAC computation and obtain digest

    std::string digestString(DigestEngine::digestToHex(digest));
        // convert to a string of hexadecimal numbers

    return 0;
}
```

DigestInputStream and DigestOutputStream

- Poco::DigestInputStream and Poco::DigestOutputStream allow for digest computation for all data written to an output stream, or read from an input stream.
- #include "Poco/DigestStream.h"
- A DigestEngine must be passed to the constructor of the stream. The streams then pass all data going through them on to the DigestEngine for digest computation.
- After writing to a DigestOutputStream, always flush the stream to ensure all data is being passed to the digest engine.

```
#include "Poco/DigestStream.h"
#include "Poco/MD5Engine.h"

using Poco::DigestOutputStream;
using Poco::DigestEngine;
using Poco::MD5Engine;

int main(int argc, char** argv)
{
    MD5Engine md5;
    DigestOutputStream ostr(md5);

    ostr << "This is some text";
    ostr.flush(); // Ensure everything gets passed to the digest engine

    const DigestEngine::Digest& digest = md5.digest(); // obtain result

    std::string result = DigestEngine::digestToHex(digest);

    return 0;
}
```

applied informatics

www.appinf.com | info@appinf.com
T +43 4253 32596 | F +43 4253 32096