Streams

Working with the various stream classes in POCO.
Overview

- Encoding and Decoding (Base64, HexBinary)
- Data Compression with zlib
- Binary I/O
- Utility Streams (CountingStream, LineEndingConverter, TeeStream, NullStream)
- FileStream
- Creating Your Own Streams
The POCO Stream Classes

- POCO provides a variety of stream classes, compatible with standard C++ IOStreams.

- Most POCO stream classes are implemented as filters, which means that they do not write to or read from a device, but rather from another stream they are connected to.

- A few utility classes in POCO make it easy for you to create your own stream buffer and stream classes.
POCO provides filter stream classes for encoding and decoding data in Base64 and HexBinary format.

Both Base64 and HexBinary can be used to encode arbitrary binary data using only printable ASCII characters.

Base64 uses digits, upper and lowercase characters, as well as '+' and '-' to encode groups of 6 bits. The encoded data takes by a factor 1.33 as much space as the original data.

HexBinary uses digits and the characters 'A' to 'F' to encode groups of 4 bit. The encoded data takes twice the space.

See RFC 4648 for details.
Encoding and Decoding (cont'd)

- Poco::Base64Encoder #include "Poco/Base64Encoder.h"
  Poco::HexBinaryEncoder #include "Poco/HexBinaryEncoder.h"
  are output streams that must be constructed with another output stream, where Base64/HexBinary-encoded data is written to.

- Poco::Base64Decoder #include "Poco/Base64Decoder.h"
  Poco::HexBinaryDecoder #include "Poco/HexBinaryDecoder.h"
  are input streams that must be constructed with another input stream, where Base64/HexBinary-encoded data is read from.
#include "Poco/Base64Encoder.h"
#include <iostream>

using Poco::Base64Encoder;

int main(int argc, char** argv)
{
    Base64Encoder encoder(std::cout);
    encoder << "Hello, world!";
    return 0;
}
POCO provides filter stream wrappers for zlib, supporting "deflate" and "gzip" style compression.

Input and output streams are provided for compression (deflating) and expansion (inflating).

Four stream classes (two input streams and two output streams) are available.
ZLib Stream Classes

raw data

DeflatingOutputStream

DeflatingInputStream

InflatingInputStream

InflatingOutputStream

ostream

istream

compressed data
ZLib Stream Classes (cont'd)

- Deflating Streams
  - `#include "Poco/DeflatingStream.h"
  - `Poco::DeflatingInputStream`
  - `Poco::DeflatingOutputStream`

- Inflating Streams
  - `#include "Poco/InflatingStream.h"
  - `Poco::InflatingInputStream`
  - `Poco::InflatingOutputStream`
ZLib Stream Classes (cont'd)

- Poco::DeflatingInputStream
  Poco::DeflatingOutputStream
  is constructed with another input/output stream and an optional argument specifying the compression type:
  Poco::DeflatingStreamBuf::STREAM_ZLIB (deflate/zlib type)
  Poco::DeflatingStreamBuf::STREAM_GZIP (gzip type)

- Poco::InflatingInputStream
  Poco::InflatingOutputStream
  is constructed with another input/output stream and an optional argument specifying the compression type:
  Poco::InflatingStreamBuf::STREAM_ZLIB (deflate/zlib type)
  Poco::InflatingStreamBuf::STREAM_GZIP (gzip type)
```cpp
#include "Poco/DeflatingStream.h"
#include <fstream>

using Poco::DeflatingOutputStream;
using Poco::DeflatingStreamBuf;

int main(int argc, char** argv)
{
    std::ofstream ostr("test.gz", std::ios::binary);
    DeflatingOutputStream deflater(ostr, DeflatingStreamBuf::STREAM_GZIP);

    deflater << "Hello, world!";

    // ensure buffers get flushed before connected stream is closed
    deflater.close();
    ostr.close();

    return 0;
}
```
Counting Streams

- Poco::CountingInputStream and Poco::CountingOutputStream count the number of characters and lines in a file. They also keep track of the current line number and column position.

- include "Poco/CountingStream.h"
Line Ending Conversion

> Poco::InputLineEndingConverter and Poco::OutputLineEndingConverter converts line endings in text files between Unix (LF), DOS/Windows (CRLF) and Macintosh (CR) format.

> #include "Poco/LineEndingConverter.h"

> Poco::LineEnding defines line ending formats:
NEWLINE_DEFAULT (the default for the current platform)
NEWLINE_CR (Macintosh line endings)
NEWLINE_CRLF (DOS/Windows line endings)
NEWLINE_LF (Unix line endings)
Splitting Streams

- `Poco::TeeInputStream` and `Poco::TeeOutputStream` copy all characters going through them (read or written) to one or more output streams.

- `#include "Poco/TeeStream.h"

- These streams are quite useful for debugging purposes.

- `void addStream(std::ostream& ostr)`
  adds an output stream to a `Poco::TeeInputStream` or `Poco::TeeOutputStream`. 
```cpp
#include "Poco/TeeStream.h"
#include <iostream>
#include <fstream>

using Poco::TeeOutputStream;

int main(int argc, char** argv)
{
    TeeOutputStream tee(std::cout);

    std::ofstream fstr("output.txt");
    tee.addStream(fstr);

    tee << "Hello, world!" << std::endl;

    return 0;
}
```
The Null Stream

- `Poco::NullOutputStream` discards all data written to it.
- `Poco::NullInputStream` signals end-of-file for every read operation.
- `#include "Poco/NullStream.h"`
Poco::BinaryWriter is used to write the value of basic types in binary form to an output stream, using a stream-like interface.

#include "Poco/BinaryWriter.h"

Poco::BinaryReader is used to read basic types in binary form (produced by a Poco::BinaryWriter) from an input stream.

#include "Poco/BinaryReader.h"

Both support big endian and little endian byte order for writing and reading, as well as automatic byte order conversions.

These classes are useful for exchanging binary data between systems with a different architecture.
The **BinaryWriter** Class

- **Poco::BinaryWriter** supports stream insertion operators (<<) for all built-in C++ types, as well as C strings and `std::string`.

- Unsigned integers (32 and 64 bit) can be written in a special compact 7 bit encoded format:
  - The value is written out seven bits at a time, starting with the seven least significant bits.
  - The most significant bit of a byte indicates whether there are more bytes coming.
  - A value that fits into seven bits takes one storage byte.
  - For a 32-bit value, at most five bytes are used.
The BinaryWriter Class (cont'd)

- `void write7BitEncoded(UInt32 value)`
  - `void write7BitEncoded(UInt64 value)`
  - writes an unsigned integer in the compact 7 bit encoded format to the underlying output stream

- `void writeRaw(const std::string& rawData)`
  - writes rawData as is to the underlying stream

- `void writeBOM()`
  - writes a byte order mark (the 16 bit value `0xFEFF` in host byte order) to the stream. A BinaryReader uses the BOM to automatically enable byte order conversion, if required.
BinaryWriter and Byte Order

- A BinaryWriter is constructed with an output stream, and an optional byte order argument.

- The byte order can be one of the following:
  - `NATIVE_BYTE_ORDER` (default)
  - `BIG_ENDIAN_BYTE_ORDER`
  - `NETWORK_BYTE_ORDER`
  - `LITTLE_ENDIAN_BYTE_ORDER`
BinaryWriter Stream State

- **Poco::BinaryWriter** provides convenience functions to determine or change the state of the underlying output stream.

- **void flush()**
  flushes the underlying stream

- **bool good()**
  returns true if the stream is okay

- **bool fail()**
  returns the state of the stream's fail bit

- **bool bad()**
  returns the state of the stream's bad bit
The BinaryReader Class

- Poco::BinaryReader provides stream extraction operators (>>>) for all built-in C++ types, as well as std::string.

- void read7BitEncoded(UInt32& value)
  void read7BitEncoded(UInt64& value)
  read an integer stored in 7 bit compressed format

- void readRaw(int length, std::string& value)
  reads length bytes of raw data into value

- void readBOM()
  reads a byte order mark and enables or disables automatic byte order conversion for all data read in the future
The BinaryReader Class (cont'd)

- **bool good()**
  returns true if the stream is okay

- **bool fail()**
  returns the state of the stream's fail bit

- **bool bad()**
  returns the state of the stream's bad bit

- **bool eof()**
  returns the state of the stream's eof bit
```cpp
#include "Poco/BinaryWriter.h"
#include <fstream>

using Poco::BinaryWriter;

int main(int argc, char** argv)
{
    std::ofstream ostr("binary.dat", std::ios::binary);
    BinaryWriter writer(ostr);

    writer.writeBOM();
    writer << "Hello, world!" << 42;
    writer.write7BitEncoded(123);
    writer << true;

    return 0;
}
```
```cpp
#include "Poco/BinaryReader.h"
#include <fstream>

using Poco::BinaryReader;

int main(int argc, char** argv)
{
    std::ifstream istr("binary.dat", std::ios::binary);
    BinaryReader reader(istr);

    reader.readBOM();

    std::string hello;
    int i;
    bool b;

    reader >> hello >> i;
    reader.read7BitEncoded(i);
    reader >> b;

    return 0;
}
```
Cross-Platform Considerations

- **Poco::BinaryWriter** and **Poco::BinaryReader** can be used to exchange data between systems with different architectures.

- Either write the data in a fixed byte order (e.g., big endian), or use a byte order mark and write in native byte order.

- Be careful with integers. Prefer **Poco::UIntXX** and **Poco::IntXX** to (unsigned) short, (unsigned) int and (unsigned) long.

- For textual data, ensure that a common encoding (e.g., Latin-1 or UTF-8) is used.
File Streams

- POCO provides stream classes for reading and writing files: FileStream, FileInputStream, FileOutputStream

- #include "Poco/FileStream.h"

- On Windows platforms, the path passed to a File Stream is UTF-8 encoded.

- No line ending conversion is performed. File streams are always open in binary mode. Seeking is supported.

- Use InputLineEndingConverter or OutputLineEndingConverter if you need CR-LF conversion.
Writing Your Own Stream Classes

- POCO provides stream buffer class templates that simplify the implementation of custom stream classes.

- Streams are implemented by first creating a stream buffer class (streambuf), and then adding IOS, istream and ostream classes.

- The following stream buffer class templates are available:
  - Poco::BasicUnbufferedStreamBuf
  - Poco::BasicBufferedStreamBuf
  - Poco::BasicBufferedBidirectionalStreamBuf
UnbufferedStreamBuf

- `Poco::BasicUnbufferedStreamBuf` is a class template that must be instantiated for a character type.
- `Poco::UnbufferedStreamBuf` is an instantiation of `Poco::BasicUnbufferedStreamBuf` for `char`.
- `#include "Poco/UnbufferedStreamBuf.h"
- `Poco::UnbufferedStreamBuf` is the simplest way to implement a custom stream. It does not do any buffering.
UnbufferedStreamBuf (cont'd)

> Subclasses must override the following member functions:

> int readFromDevice()
  reads and returns a single (unsigned) byte. Returns char_traits::eof() (-1) if no more data is available.

NOTE: Never return a char value directly, as char might be signed. Always use int charToInt(char c) to convert the character to an integer.

> int writeToDevice(char c)
  writes a single byte. Returns the byte (as integer) if successful, otherwise char_traits::eof() (-1).
```cpp
#include "Poco/UnbufferedStreamBuf.h"
#include <ostream>
#include <cctype>

class UpperStreamBuf: public UnbufferedStreamBuf
{
public:
    UpperStreamBuf(std::ostream& ostr): _ostr(ostr)
    {
    }

protected:
    int writeToDevice(char c)
    {
        _ostr.put(toupper(c));
        return charToInt(c);
    }

private:
    std::ostream& _ostr;
};
```
class UpperIOS: public virtual std::ios
{
    public:
        UpperIOS(std::ostream& ostr): _buf(ostr)
        {
            poco_ios_init(&_buf);
        }
    
    protected:
        UpperStreamBuf _buf;
};

class UpperOutputStream: public UpperIOS, public std::ostream
{
    public:
        UpperOutputStream(std::ostream& ostr):
            UpperIOS(ostr),
            std::ostream(&_buf)
        {
        }
};
int main(int argc, char** argv)
{
    UpperOutputStream upper(std::cout);
    upper << "Hello, world!" << std::endl;
    return 0;
}
Buffered Streams

- **Poco::BasicBufferedStreamBuf** is a class template that must be instantiated for a character type.
- **Poco::BufferedStreamBuf** is an instantiation of **Poco::BasicBufferedStreamBuf** for char.
- `#include "Poco/BufferedStreamBuf.h"`
- An instance of **Poco::BufferedStreamBuf** supports either reading or writing, but not both.
- **Poco::BasicBufferedBidirectionalStreamBuf** supports reading and writing. Internally, it maintains two buffers.
- `#include "Poco/BufferedBidirectionalStreamBuf.h"`
Buffered Streams (cont'd)

> Subclasses of `Buffered[Bidirectional]StreamBuf` must override the following member functions:

> `int readFromDevice(char* buffer, std::streamsize length)`
read up to `length` characters and place them in `buffer`. Return the number of characters read, or -1 if something went wrong.

> `int writeToDevice(const char* buffer, std::streamsize length)`
write `length` bytes starting from `buffer` and return the number of bytes written, or -1 if something went wrong.
Stream Buffers and Exceptions

Exceptions thrown by stream buffers will normally be catched by the stream class and result in the stream's bad bit being set. The exception will not propagate; instead, the stream's bad bit will be set.

This behavior of a stream can be changed by calling the exceptions() member function of a stream with true as argument.