

fstream.cm

```
/*  
  
    Copyright (c) 2012–2016 Seppo Laakko  
    http://sourceforge.net/projects/cmajor/  
  
    Distributed under the GNU General Public License, version 3 (GPLv3).  
    (See accompanying LICENSE.txt or http://www.gnu.org/licenses/gpl.html  
    )  
  
*/  
  
// Copyright (c) 1994  
// Hewlett-Packard Company  
// Copyright (c) 1996  
// Silicon Graphics Computer Systems, Inc.  
// Copyright (c) 2009 Alexander Stepanov and Paul McJones  
  
using System;  
using System.Support;  
  
namespace System.IO  
{  
    public class OpenFileException: Exception  
    {  
        publicnothrow OpenFileException(const string& message_): base(  
            message_)  
        {  
        }  
    }  
  
    public class CloseFileException: Exception  
    {  
        publicnothrow CloseFileException(const string& message_): base(  
            message_)  
        {  
        }  
    }  
  
    public class IOException: Exception  
    {  
        publicnothrow IOException(const string& message_): base(message_)  
        {  
        }  
    }  
  
    public class IOBuffer  
    {
```

```

    public nothrow IOBuffer(ulong size_): size(size_), mem(MemAlloc(
        size_))
    {
    }
    suppress IOBuffer(const IOBuffer&);
    suppress void operator=(const IOBuffer&);
    public nothrow IOBuffer(IOBuffer&& that): size(that.size), mem(
        that.mem)
    {
        that.size = 0u;
        that.mem = null;
    }
    public nothrow default void operator=(IOBuffer&& that);
    public nothrow ~IOBuffer()
    {
        MemFree(mem);
    }
    public inline nothrow void* Mem() const
    {
        return mem;
    }
    public inline nothrow ulong Size() const
    {
        return size;
    }
    private ulong size;
    private void* mem;
}

public class InputFileStream: InputStream
{
    public nothrow InputFileStream(int handle_, uint bufferSize):
        fileName(),
        handle(handle_),
        fileIsOpen(false),
        buffer(bufferSize),
        pos(null),
        end(null),
        endOfStream(false)
    {
    }
    public nothrow InputFileStream(): this(stdin, defaultBufferSize)
    {
    }
    public InputFileStream(const string& fileName_): this(-1,
        defaultBufferSize)
    {
        Open(fileName_);
    }
    public InputFileStream(const string& fileName_, uint bufferSize)
        : this(-1, bufferSize)
    {
        Open(fileName_);
    }

```

```

}
suppress InputFileStream(const InputFileStream&);
suppress void operator=(const InputFileStream&);
public nothrow InputFileStream(InputFileStream&& that):
    fileName(Rvalue(that.fileName)), handle(that.handle),
    fileIsOpen(that.fileIsOpen), buffer(Rvalue(that.buffer)),
    pos(that.pos), end(that.end), endOfStream(that.endOfStream)
    )
{
    that.handle = -1;
    that.fileIsOpen = false;
    that.pos = null;
    that.end = null;
    that.endOfStream = false;
}
public nothrow default void operator=(InputFileStream&&);
public void Open(const string& fileName_)
{
    if (fileIsOpen)
    {
        Close();
    }
    fileName = fileName_;
    if (fileName.IsEmpty())
    {
        throw OpenFileException("given file name is empty");
    }
    handle = open_file(fileName.Chars(), cast<OpenFlags>(
        OpenFlags.readOnly | OpenFlags.text), 0);
    if (handle == -1)
    {
        string reason = strerror(get_errno());
        throw OpenFileException("could not open file " +
            fileName + "' for reading: " + reason);
    }
    fileIsOpen = true;
}
public override ~InputFileStream()
{
    if (fileIsOpen)
    {
        try
        {
            Close();
        }
        catch (const Exception&)
        {
        }
    }
}
public void Close()
{
    if (fileIsOpen)

```

```

{
    fileIsOpen = false;
    int result = close(handle);
    handle = -1;
    if (result == -1)
    {
        string reason = strerror(get_errno());
        throw CloseFileException("could not close file " +
            fileName + "': " + reason);
    }
}
else
{
    throw CloseFileException("no file is open");
}
}
public nothrow const string& FileName() const
{
    return fileName;
}
public nothrow int Handle() const
{
    return handle;
}
public override string ReadLine()
{
    string line;
    while (!endOfStream)
    {
        if (pos == end)
        {
            int result = read_64(handle, buffer.Mem(), buffer.
                Size());
            if (result == -1)
            {
                string reason = strerror(get_errno());
                throw IOException("could not read from file " +
                    fileName + "': " + reason);
            }
            else if (result == 0)
            {
                endOfStream = true;
            }
            else
            {
                pos = cast<const char*>(buffer.Mem());
                end = pos + result;
            }
        }
        while (pos != end)
        {
            if (*pos == '\n')
            {

```

```

        ++pos;
        return line;
    }
    line.Append(*pos++);
}
}
return line;
}
public override string ReadToEnd()
{
    string content;
    while (!endOfStream)
    {
        int result = read_64(handle, buffer.Mem(), buffer.Size())
        ;
        if (result == -1)
        {
            string reason = strerror(errno);
            throw IOException("could not read from file '" +
                fileName + "': " + reason);
        }
        else if (result == 0)
        {
            endOfStream = true;
        }
        else
        {
            content.Append(cast<const char*>(buffer.Mem()),
                result);
        }
    }
    return content;
}
public nothrow override bool EndOfStream() const
{
    return endOfStream;
}
private string fileName;
private int handle;
private bool fileIsOpen;
private IOBuffer buffer;
private const char* pos;
private const char* end;
private bool endOfStream;
private const uint defaultBufferSize = 4096u;
}

public string ReadFile(const string& fileName)
{
    InputFileStream s(fileName, cast<uint>(4u) * 1024u * 1024u);
    return s.ReadToEnd();
}

```

```

public class OutputFileStream: OutputStream
{
    static nothrow OutputFileStream(): newline("\n")
    {
    }
    public nothrow OutputFileStream(int handle_): fileName(), handle(
        handle_), fileIsOpen(false)
    {
    }
    public nothrow OutputFileStream(): this(stdout)
    {
    }
    public OutputFileStream(const string& fileName_): fileName(
        fileName_), handle(-1), fileIsOpen(false)
    {
        Open(fileName_);
    }
    public OutputFileStream(const string& fileName_, int pmode):
        fileName(fileName_), handle(-1), fileIsOpen(false)
    {
        Open(fileName_, pmode);
    }
    public OutputFileStream(const string& fileName_, bool append):
        fileName(fileName_), handle(-1), fileIsOpen(false)
    {
        Open(fileName_, append);
    }
    public OutputFileStream(const string& fileName_, int pmode, bool
        append): fileName(fileName_), handle(-1), fileIsOpen(false)
    {
        Open(fileName_, pmode, append);
    }
    suppress OutputFileStream(const OutputFileStream&);
    suppress void operator=(const OutputFileStream&);
    public nothrow OutputFileStream(OutputFileStream&& that):
        fileName(Rvalue(that.fileName)), handle(that.handle),
        fileIsOpen(that.fileIsOpen)
    {
        that.handle = -1;
        that.fileIsOpen = false;
    }
    public nothrow default void operator=(OutputFileStream&&);
    public void Open(const string& fileName_)
    {
        Open(fileName_, get_default_pmode(), false);
    }
    public void Open(const string& fileName_, int pmode)
    {
        Open(fileName_, pmode, false);
    }
    public void Open(const string& fileName_, bool append)
    {
        Open(fileName_, get_default_pmode(), append);
    }
}

```

```

}
public void Open(const string& fileName_, int pmode, bool append)
{
    if (fileIsOpen)
    {
        Close();
    }
    fileName = fileName_;
    if (fileName.IsEmpty())
    {
        throw OpenFileException("given file name is empty");
    }
    OpenFlags openFlags = cast<OpenFlags>(OpenFlags.text |
        OpenFlags.writeOnly);
    if (append)
    {
        openFlags = cast<OpenFlags>(openFlags | OpenFlags.create
            | OpenFlags.append);
    }
    else
    {
        openFlags = cast<OpenFlags>(openFlags | OpenFlags.create
            | OpenFlags.truncate);
    }
    handle = open_file(fileName.Chars(), openFlags, pmode);
    if (handle == -1)
    {
        string reason = strerror(errno);
        throw OpenFileException("could not open file '" +
            fileName + "' for writing: " + reason);
    }
    fileIsOpen = true;
}
public override ~OutputFileStream()
{
    if (fileIsOpen)
    {
        try
        {
            Close();
        }
        catch (const Exception&)
        {
        }
    }
}
public void Close()
{
    if (fileIsOpen)
    {
        fileIsOpen = false;
        int result = close(handle);
        handle = -1;
    }
}

```

```

        if (result == -1)
        {
            string reason = strerror(errno());
            throw CloseFileException("could not close file '" +
                fileName + "': " + reason);
        }
    }
    else
    {
        throw CloseFileException("no file is open");
    }
}

public nothrow const string& FileName() const
{
    return fileName;
}

public nothrow int Handle() const
{
    return handle;
}

public override void Write(const char* s)
{
    int result = write_64(handle, s, cast<ulong>(StrLen(s)));
    if (result == -1)
    {
        string reason = strerror(errno());
        throw IOException("could not write to file '" + fileName
            + "': " + reason);
    }
}

public override void Write(const string& s)
{
    int result = write_64(handle, s.Chars(), cast<ulong>(s.Length
        ()));
    if (result == -1)
    {
        string reason = strerror(errno());
        throw IOException("could not write to file '" + fileName
            + "': " + reason);
    }
}

public override void Write(char c)
{
    Write(ToString(c));
}

public override void Write(byte b)
{
    Write(ToString(b));
}

public override void Write(sbyte s)
{
    Write(ToString(s));
}

```



```

public override void Write(short s)
{
    Write(ToString(s));
}
public override void Write(ushort u)
{
    Write(ToString(u));
}
public override void Write(int i)
{
    Write(ToString(i));
}
public override void Write(uint i)
{
    Write(ToString(i));
}
public override void Write(long l)
{
    Write(ToString(l));
}
public override void Write(ulong u)
{
    Write(ToString(u));
}
public override void Write(bool b)
{
    Write(ToString(b));
}
public override void Write(float f)
{
    Write(ToString(f));
}
public override void Write(double d)
{
    Write(ToString(d));
}
public override void WriteLine()
{
    Write(newline);
}
public override void WriteLine(const char* s)
{
    Write(s);
    WriteLine();
}
public override void WriteLine(const string& s)
{
    Write(s);
    WriteLine();
}
public override void WriteLine(char c)
{
    Write(c);
}

```

```

        WriteLine();
    }
    public override void WriteLine(byte b)
    {
        Write(b);
        WriteLine();
    }
    public override void WriteLine(sbyte s)
    {
        Write(s);
        WriteLine();
    }
    public override void WriteLine(short s)
    {
        Write(s);
        WriteLine();
    }
    public override void WriteLine(ushort u)
    {
        Write(u);
        WriteLine();
    }
    public override void WriteLine(int i)
    {
        Write(i);
        WriteLine();
    }
    public override void WriteLine(uint u)
    {
        Write(u);
        WriteLine();
    }
    public override void WriteLine(long l)
    {
        Write(l);
        WriteLine();
    }
    public override void WriteLine(ulong u)
    {
        Write(u);
        WriteLine();
    }
    public override void WriteLine(bool b)
    {
        Write(b);
        WriteLine();
    }
    public override void WriteLine(float f)
    {
        Write(f);
        WriteLine();
    }
    public override void WriteLine(double d)

```

```

    {
        Write(d);
        WriteLine();
    }
    private string fileName;
    private int handle;
    private bool fileIsOpen;
    private static const char* newline;
}

public enum OpenMode
{
    readOnly, writeOnly, readWrite
}

public class BinaryFileStream
{
    public BinaryFileStream(const string& fileName_, OpenMode mode_):
        fileName(fileName_), handle(-1), fileIsOpen(false)
    {
        Open(fileName_, mode_, get_default_pmode());
    }
    public BinaryFileStream(const string& fileName_, OpenMode mode_,
        int pmode): fileName(fileName_), handle(-1), fileIsOpen(false)
    {
        Open(fileName_, mode_, pmode);
    }
    suppress BinaryFileStream(const BinaryFileStream&);
    suppress void operator=(const BinaryFileStream&);
    public nothrow BinaryFileStream(BinaryFileStream&& that):
        fileName(Rvalue(that.fileName)), handle(that.handle),
        fileIsOpen(that.fileIsOpen)
    {
        that.handle = -1;
        that.fileIsOpen = false;
    }
    public nothrow default void operator=(BinaryFileStream&&);
    public void Open(const string& fileName_, OpenMode mode_, int
        pmode)
    {
        if (fileIsOpen)
        {
            Close();
        }
        fileName = fileName_;
        if (fileName.IsEmpty())
        {
            throw OpenFileException("given file name is empty");
        }
        if (mode_ == OpenMode.readOnly)
        {
            handle = open_file(fileName.Chars(), cast<OpenFlags>(
                OpenFlags.binary | OpenFlags.readOnly), 0);

```

```

    }
    else if (mode_ == OpenMode.writeOnly)
    {
        handle = open_file(fileName.Chars(), cast<OpenFlags>(
            OpenFlags.binary | OpenFlags.writeOnly | OpenFlags.
            create | OpenFlags.truncate), pmode);
    }
    else if (mode_ == OpenMode.readWrite)
    {
        handle = open_file(fileName.Chars(), cast<OpenFlags>(
            OpenFlags.binary | OpenFlags.readWrite), 0);
    }
    if (handle == -1)
    {
        string reason = strerror(get_errno());
        throw OpenFileException("could not open file '" +
            fileName + "': " + reason);
    }
    fileIsOpen = true;
}
public ~BinaryFileStream()
{
    if (fileIsOpen)
    {
        try
        {
            Close();
        }
        catch (const Exception&)
        {
        }
    }
}
public void Close()
{
    if (fileIsOpen)
    {
        fileIsOpen = false;
        int result = close(handle);
        handle = -1;
        if (result == -1)
        {
            string reason = strerror(get_errno());
            throw CloseFileException("could not close file '" +
                fileName + "': " + reason);
        }
    }
    else
    {
        throw CloseFileException("no file is open");
    }
}
public void Write(void* buffer, ulong size)

```

```

{
    int result = write_64(handle, buffer, size);
    if (result != size)
    {
        string reason = strerror(errno);
        throw IOException("could not write to file '" + fileName
            + "': " + reason);
    }
}
public int Read(void* buffer, ulong size)
{
    int result = read_64(handle, buffer, size);
    if (result == -1)
    {
        string reason = strerror(errno);
        throw IOException("could not read from file '" + fileName
            + "': " + reason);
    }
    return result;
}
public void ReadSize(void* buffer, ulong size)
{
    int bytesRead = Read(buffer, size);
    if (bytesRead != size)
    {
        throw IOException("unexpected end of file '" + fileName +
            "'");
    }
}
public long Seek(long offset, int origin)
{
    long result = lseek(handle, offset, origin);
    if (result == -1)
    {
        string reason = strerror(errno);
        throw IOException("could not seek file '" + fileName +
            "': " + reason);
    }
    return result;
}
public long Tell()
{
    return Seek(0, SEEK_CUR);
}
public void Write(const char* s)
{
    int len = StrLen(s);
    Write(len);
    Write(s, cast<ulong>(len));
}
public void Write(const string& s)
{
    int len = s.Length();

```

```

        Write(len);
        Write(s.Chars(), cast<ulong>(len));
    }
    public string ReadString()
    {
        int len = 0;
        ReadSize(&len, sizeof(len));
        ulong size = cast<ulong>(len);
        IOBuffer buffer(size);
        ReadSize(buffer.Mem(), size);
        return string(cast<const char*>(buffer.Mem()), len);
    }
    public void Write(char c)
    {
        Write(&c, sizeof(c));
    }
    public char ReadChar()
    {
        char c = '\0';
        ReadSize(&c, sizeof(c));
        return c;
    }
    public void Write(byte b)
    {
        Write(&b, sizeof(b));
    }
    public byte ReadByte()
    {
        byte b = 0u;
        ReadSize(&b, sizeof(b));
        return b;
    }
    public void Write(sbyte s)
    {
        Write(&s, sizeof(s));
    }
    public sbyte ReadSByte()
    {
        sbyte s = 0;
        ReadSize(&s, sizeof(s));
        return s;
    }
    public void Write(short s)
    {
        Write(&s, sizeof(s));
    }
    public short ReadShort()
    {
        short s = 0;
        ReadSize(&s, sizeof(s));
        return s;
    }
    public void Write(ushort u)

```

```

{
    Write(&u, sizeof(u));
}
public ushort ReadUShort()
{
    ushort u = 0u;
    ReadSize(&u, sizeof(u));
    return u;
}
public void Write(int i)
{
    Write(&i, sizeof(i));
}
public int ReadInt()
{
    int i = 0;
    ReadSize(&i, sizeof(i));
    return i;
}
public void Write(uint u)
{
    Write(&u, sizeof(u));
}
public uint ReadUInt()
{
    uint u = 0u;
    ReadSize(&u, sizeof(u));
    return u;
}
public void Write(long l)
{
    Write(&l, sizeof(l));
}
public long ReadLong()
{
    long l = 0;
    ReadSize(&l, sizeof(l));
    return l;
}
public void Write(ulong u)
{
    Write(&u, sizeof(u));
}
public ulong ReadULong()
{
    ulong u = 0u;
    ReadSize(&u, sizeof(u));
    return u;
}
public void Write(bool b)
{
    Write(&b, sizeof(b));
}

```

```

public bool ReadBool()
{
    bool b = false;
    ReadSize(&b, sizeof(b));
    return b;
}
public void Write(float f)
{
    Write(&f, sizeof(f));
}
public float ReadFloat()
{
    float f = 0;
    ReadSize(&f, sizeof(f));
    return f;
}
public void Write(double d)
{
    Write(&d, sizeof(d));
}
public double ReadDouble()
{
    double d = 0;
    ReadSize(&d, sizeof(d));
    return d;
}
public long GetFileSize()
{
    long pos = Tell();
    long end = Seek(0, SEEK_END);
    Seek(pos, SEEK_SET);
    return end;
}
private string fileName;
private int handle;
private bool fileIsOpen;
}

public bool FileContentsEqual(const string& fileName1, const string&
fileName2)
{
    BinaryFileStream file1(fileName1, OpenMode.readOnly);
    BinaryFileStream file2(fileName2, OpenMode.readOnly);
    long size1 = file1.GetFileSize();
    long size2 = file2.GetFileSize();
    if (size1 != size2) return false;
    while (size1 > 0)
    {
        int size = 4096;
        if (size1 < size)
        {
            size = cast<int>(size1);
            size1 = 0;

```



```

    }
    else
    {
        size1 = size1 - 4096;
    }
    IOBuffer buffer1(cast<ulong>(size));
    file1.Read(buffer1.Mem(), buffer1.Size());
    IOBuffer buffer2(cast<ulong>(size));
    file2.Read(buffer2.Mem(), buffer2.Size());
    const char* p1 = cast<const char*>(buffer1.Mem());
    const char* p2 = cast<const char*>(buffer2.Mem());
    for (int i = 0; i < size; ++i)
    {
        if (p1[i] != p2[i]) return false;
    }
}
return true;
}

public class FileMappingFailure : Exception
{
    public FileMappingFailure(const string& message_) : base(message_)
    {
    }
}

public void ThrowFileMappingFailure(const string& message)
{
    throw FileMappingFailure(message);
}

public class FileMapping
{
    public FileMapping(const string& filePath_): filePath(filePath_),
        handle(-1), begin(null), end(null)
    {
        if (!FileExists(filePath))
        {
            ThrowFileMappingFailure("could not create file mapping
            for file '" + filePath + "': no such file");
        }
        handle = create_file_mapping(filePath.Chars(), &begin, &end);
        if (handle < 0)
        {
            switch (handle)
            {
                case -1: ThrowFileMappingFailure("could not create
                file mapping for file '" + filePath + "': too many
                file mappings"); break;
                case -2: ThrowFileMappingFailure("could not create
                file mapping for file '" + filePath + "': could
                not open file for reading"); break;
            }
        }
    }
}

```

```

        case -3: ThrowFileMappingFailure("could not create
            file mapping for file '" + filePath + "': could
            not get size of file"); break;
        case -4: ThrowFileMappingFailure("could not create
            file mapping for file '" + filePath + "': failed
            creating mapping"); break;
        case -5: ThrowFileMappingFailure("could not create
            file mapping for file '" + filePath + "': failed
            mapping view"); break;
        default: ThrowFileMappingFailure("could not create
            file mapping for file '" + filePath + "': unknown
            error"); break;
    }
}
}
public nothrow ~FileMapping()
{
    if (handle >= 0)
    {
        close_file_mapping(handle);
    }
}
suppress FileMapping(const FileMapping&);
suppress void operator=(const FileMapping&);
public nothrow FileMapping(FileMapping&& that) : handle(that.
    handle), filePath(Rvalue(that.filePath)), begin(that.begin),
    end(that.end)
{
    that.handle = -1;
}
public default nothrow void operator=(FileMapping&&);
public inline nothrow const string& FilePath() const
{
    return filePath;
}
public inline nothrow const char* Begin() const
{
    const char* start = begin;
    if (end - begin >= 3)
    {
        if (cast<byte>(start[0]) == 0xEFu &&
            cast<byte>(start[1]) == 0xBBu &&
            cast<byte>(start[2]) == 0xBFu)
        {
            start = start + 3;
        }
    }
    return start;
}
public inline nothrow const char* End() const
{
    return end;
}

```

```
private int handle;  
private string filePath;  
private const char* begin;  
private const char* end;  
}  
}
```