string.cm

```
/*
    ========================================================================

    Copyright (c) 2012-2015 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.Support;
using System.Collections;

namespace System
{
    public static class EmptyStrHolder
    {
        public static nothrow EmptyStrHolder(): emptyStr("")
        {
        }
        public static nothrow const char* GetEmptyStr()
        {
            return emptyStr;
        }
        private static const char* emptyStr;
    }

    public class String
    {
        public typedef RandomAccessIter<char, const char&, const char*>
            ConstIterator;
        public typedef RandomAccessIter<char, char&, char*> Iterator;

        public nothrow String(): chars(null), len(0), res(0)
        {
        }
        public nothrow String(const char* chars_): len(StrLen(chars_)),
            res(0), chars(null)
        {
            if (len > 0)
            {
                Reserve(len);
```

```
            StrCopy(chars, chars_);
        }
    }
    public nothrow String(const char* chars_, int length_): len(0),
        res(0), chars(null)
    {
        if (length_ > 0)
        {
            Reserve(length_);
            len = StrCopy(chars, chars_, length_);
        }
    }
    public nothrow String(const String& that): len(that.len), res(0),
         chars(null)
    {
        if (len > 0)
        {
            Reserve(len);
            StrCopy(chars, that.chars);
        }
    }
    public nothrow String(String&& that): len(that.len), res(that.res
        ), chars(that.chars)
    {
        that.len = 0;
        that.res = 0;
        that.chars = null;
    }
    public nothrow String(char c): len(1), res(0), chars(null)
    {
        Reserve(1);
        chars[0] = c;
        chars[1] = '\0';
    }
    public nothrow String(char c, int n): len(n), res(0), chars(null)
    {
        Reserve(n);
        for (int i = 0; i < n; ++i)
        {
            chars[i] = c;
        }
    }
    public nothrow void operator=(const String& that)
    {
        Deallocate();
        Reserve(that.len);
        len = that.len;
        if (len > 0)
        {
            StrCopy(chars, that.chars);
        }
    }
    public default nothrow void operator=(String&& that);
```

```
public nothrow ~String()
{
    Deallocate();
}
public nothrow inline int Length() const
{
    return len;
}
public nothrow inline int Capacity() const
{
    return res;
}
public nothrow inline bool IsEmpty() const
{
    return len == 0;
}
public nothrow void Clear()
{
    Deallocate();
}
public nothrow const char* Chars() const
{
    if (chars != null)
    {
        return chars;
    }
    return EmptyStrHolder.GetEmptyStr();
}
public nothrow char operator[](int index) const
{
    #assert(index >= 0 && index < len);
    return chars[index];
}
public nothrow char& operator[](int index)
{
    #assert(index >= 0 && index < len);
    return chars[index];
}
public nothrow void Reserve(int minLen)
{
    if (minLen > 0)
    {
        int minRes = minLen + 1;
        if (minRes > res)
        {
            Grow(minRes);
        }
    }
}
public nothrow String& Append(char c)
{
    Reserve(len + 1);
    chars[len] = c;
```

```
        chars[++len] = '\0';
        return *this;
    }
    public nothrow String& Append(const char* that)
    {
        AppendFrom(that, StrLen(that));
        return *this;
    }
    public nothrow String& Append(const char* that, int count)
    {
        AppendFrom(that, count);
        return *this;
    }
    public nothrow String& Append(const String& that)
    {
        AppendFrom(that.chars, that.len);
        return *this;
    }
    public nothrow void Replace(char oldChar, char newChar)
    {
        int n = len;
        for (int i = 0; i < n; ++i)
        {
            if (chars[i] == oldChar)
            {
                chars[i] = newChar;
            }
        }
    }
    public nothrow String Substring(int start) const
    {
        if (start >= 0 && start < len)
        {
            return String(chars + start);
        }
        return String();
    }
    public nothrow String Substring(int start, int length) const
    {
        if (start >= 0 && start < len)
        {
            return String(chars + start, length);
        }
        return String();
    }
    public nothrow Iterator Begin()
    {
        return Iterator(chars);
    }
    public nothrow ConstIterator Begin() const
    {
        return ConstIterator(chars);
    }
```

```
public nothrow ConstIterator CBegin() const
{
    return ConstIterator(chars);
}
public nothrow Iterator End()
{
    if (chars != null)
    {
        return Iterator(chars + len);
    }
    return Iterator(null);
}
public nothrow ConstIterator End() const
{
    if (chars != null)
    {
        return ConstIterator(chars + len);
    }
    return ConstIterator(null);
}
public nothrow ConstIterator CEnd() const
{
    if (chars != null)
    {
        return ConstIterator(chars + len);
    }
    return ConstIterator(null);
}
public nothrow bool operator==(const String& that) const
{
    if (len != that.len) return false;
    for (int i = 0; i < len; ++i)
    {
        if (chars[i] != that.chars[i])
        {
            return false;
        }
    }
    return true;
}
public nothrow bool operator<(const String& that) const
{
    if (len == 0 && that.len > 0) return true;
    if (len > 0 && that.len == 0) return false;
    int n = Min(len, that.len);
    for (int i = 0; i < n; ++i)
    {
        char left = chars[i];
        char right = that.chars[i];
        if (left < right) return true;
        if (left > right) return false;
    }
    if (len < that.len) return true;
```

```
        return false;
    }
    public nothrow bool StartsWith(const String& prefix) const
    {
        int n = prefix.len;
        if (len < n) return false;
        for (int i = 0; i < n; ++i)
        {
            if (chars[i] != prefix[i]) return false;
        }
        return true;
    }
    public nothrow bool EndsWith(const String& suffix) const
    {
        int n = len;
        int m = suffix.len;
        if (n < m) return false;
        for (int i = 0; i < m; ++i)
        {
            if (chars[i + n - m] != suffix[i]) return false;
        }
        return true;
    }
    public List<String> Split(char c)
    {
        List<String> result;
        int start = 0;
        for (int i = 0; i < len; ++i)
        {
            if (chars[i] == c)
            {
                result.Add(Substring(start, i - start));
                start = i + 1;
            }
        }
        if (start < len)
        {
            result.Add(Substring(start));
        }
        return result;
    }
    public nothrow int Find(char x) const
    {
        return Find(x, 0);
    }
    public nothrow int Find(char x, int start) const
    {
        #assert(start >= 0);
        for (int i = start; i < len; ++i)
        {
            if (chars[i] == x)
            {
                return i;
```

```
            }
        }
        return -1;
    }
    public nothrow int RFind(char x) const
    {
        return RFind(x, len - 1);
    }
    public nothrow int RFind(char x, int start) const
    {
        #assert(start < len);
        for (int i = start; i >= 0; --i)
        {
            if (chars[i] == x)
            {
                return i;
            }
        }
        return -1;
    }
    public nothrow int Find(const String& s) const
    {
        return Find(s, 0);
    }
    public nothrow int Find(const String& s, int start) const
    {
        #assert(start >= 0);
        if (s.IsEmpty()) return start;
        int n = s.Length();
        char x = s[0];
        int i = Find(x, start);
        while (i != -1)
        {
            if (len < i + n) return -1;
            bool found = true;
            for (int k = 1; k < n; ++k)
            {
                if (chars[i + k] != s[k])
                {
                    found = false;
                    break;
                }
            }
            if (found)
            {
                return i;
            }
            i = Find(x, i + 1);
        }
        return -1;
    }
    public nothrow int RFind(const String& s) const
    {
```

```
        return RFind(s, len - 1);
    }
    public nothrow int RFind(const String& s, int start) const
    {
        #assert(start < len);
        if (s.IsEmpty()) return start;
        int n = s.Length();
        char x = s[0];
        int i = RFind(x, start);
        while (i != -1)
        {
            if (len >= i + n)
            {
                bool found = true;
                for (int k = 1; k < n; ++k)
                {
                    if (chars[i + k] != s[k])
                    {
                        found = false;
                        break;
                    }
                }
                if (found)
                {
                    return i;
                }
            }
            i = RFind(x, i - 1);
        }
        return -1;
    }
    private nothrow void AppendFrom(const char* that, int thatLen)
    {
        int newLen = len + thatLen;
        if (newLen > 0)
        {
            Reserve(newLen);
            newLen = len + StrCopy(chars + len, that, thatLen);
        }
        len = newLen;
    }
    private nothrow void Grow(int minRes)
    {
        minRes = cast<int>(MemGrow(cast<ulong>(minRes)));
        char* newChars = cast<char*>(MemAlloc(cast<ulong>(minRes)));
        if (chars != null)
        {
            StrCopy(newChars, chars);
            MemFree(chars);
        }
        chars = newChars;
        res = minRes;
    }
```

```
        private nothrow void Deallocate()
        {
            len = 0;
            if (res != 0)
            {
                MemFree(chars);
                res = 0;
            }
            chars = null;
        }
        private int len;
        private int res;
        private char* chars;
    }

    public typedef String string;

    public nothrow string operator+(const string& first, const string&
        second)
    {
        string temp(first);
        temp.Append(second);
        return temp;
    }

    public nothrow string operator+(const string& first, const char*
        second)
    {
        string temp(first);
        temp.Append(second);
        return temp;
    }

    public nothrow string operator+(const char* first, const string&
        second)
    {
        string temp(first);
        temp.Append(second);
        return temp;
    }

    public nothrow string ToLower(const string& s)
    {
        string result;
        int n = s.Length();
        result.Reserve(n);
        for (int i = 0; i < n; ++i)
        {
            char c = s[i];
            if (c >= 'A' && c <= 'Z')
            {
                result.Append(cast<char>(cast<int>(c) + (cast<int>('a') -
                    cast<int>('A'))));
```

```
            }
            else
            {
                result.Append(c);
            }
        }
        return result;
    }

    public nothrow string ToUpper(const string& s)
    {
        string result;
        int n = s.Length();
        result.Reserve(n);
        for (int i = 0; i < n; ++i)
        {
            char c = s[i];
            if (c >= 'a' && c <= 'z')
            {
                result.Append(cast<char>(cast<int>(c) + (cast<int>('A') -
                    cast<int>('a'))));
            }
            else
            {
                result.Append(c);
            }
        }
        return result;
    }

    public bool LastComponentsEqual(const string& s0, const string& s1,
       char componentSeparator)
    {
        List<string> c0 = s0.Split(componentSeparator);
        List<string> c1 = s1.Split(componentSeparator);
        int n0 = c0.Count();
        int n1 = c1.Count();
        int n = Min(n0, n1);
        for (int i = 0; i < n; ++i)
        {
            if (c0[n0 - i - 1] != c1[n1 - i - 1]) return false;
        }
        return true;
    }
}
```