string.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.Support;
using System.Collections;

namespace System
{
    public nothrow const char* EmptyString(char x)
    {
        return "";
    }

    public nothrow const wchar* EmptyString(wchar x)
    {
        return w"";
    }

    public nothrow const uchar* EmptyString(uchar x)
    {
        return u"";
    }

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

        public nothrow String(): chars(null), len(0), res(0)
        {
        }
        public nothrow String(const CharT* chars_): len(StrLen(chars_)),
            res(0), chars(null)
        {
```

```
        if (len > 0)
        {
            Reserve(len);
            StrCopy(chars, chars_);
        }
    }
    public nothrow String(const CharT* chars_, int length_): len(0),
        res(0), chars(null)
    {
        if (length_ > 0)
        {
            Reserve(length_);
            len = StrCopy(chars, chars_, length_);
        }
    }
    public nothrow String(const CharT* begin, const CharT* end) :
        this(begin, end - begin)
    {
    }
    public nothrow String(const String<CharT>& that): len(that.len),
        res(0), chars(null)
    {
        if (len > 0)
        {
            Reserve(len);
            StrCopy(chars, that.chars);
        }
    }
    public nothrow String(String<CharT>&& that): len(that.len), res(
        that.res), chars(that.chars)
    {
        that.len = 0;
        that.res = 0;
        that.chars = null;
    }
    public nothrow String(CharT c): len(1), res(0), chars(null)
    {
        Reserve(1);
        chars[0] = c;
        chars[1] = '\0';
    }
    public nothrow String(CharT 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<CharT>& that)
    {
        Deallocate();
```

```
        Reserve(that.len);
        len = that.len;
        if (len > 0)
        {
            StrCopy(chars, that.chars);
        }
    }
    public default nothrow void operator=(String<CharT>&& 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 CharT* Chars() const
    {
        if (chars != null)
        {
            return chars;
        }
        return EmptyString(CharT());
    }
    public nothrow CharT operator[](int index) const
    {
        #assert(index >= 0 && index < len);
        return chars[index];
    }
    public nothrow CharT& 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<CharT>& Append(CharT c)
    {
        Reserve(len + 1);
        chars[len] = c;
        chars[++len] = '\0';
        return *this;
    }
    public nothrow String<CharT>& Append(CharT c, int count)
    {
        #assert(count >= 0);
        if (count > 0)
        {
            Reserve(len + count);
            for (int i = 0; i < count; ++i)
            {
                chars[len++] = c;
            }
            chars[len] = '\0';
        }
        return *this;
    }
    public nothrow String<CharT>& Append(const CharT* that)
    {
        AppendFrom(that, StrLen(that));
        return *this;
    }
    public nothrow String<CharT>& Append(const CharT* that, int count
        )
    {
        AppendFrom(that, count);
        return *this;
    }
    public nothrow String<CharT>& Append(const String<CharT>& that)
    {
        AppendFrom(that.chars, that.len);
        return *this;
    }
    public nothrow void Replace(CharT oldChar, CharT newChar)
    {
        int n = len;
        for (int i = 0; i < n; ++i)
        {
            if (chars[i] == oldChar)
            {
                chars[i] = newChar;
            }
        }
    }
    public nothrow String<CharT> Substring(int start) const
```

```
{
    if (start >= 0 && start < len)
    {
        return String<CharT>(chars + start);
    }
    return String<CharT>();
}
public nothrow String<CharT> Substring(int start, int length)
    const
{
    if (start >= 0 && start < len)
    {
        return String<CharT>(chars + start, length);
    }
    return String<CharT>();
}
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<CharT>& 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<CharT>& 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)
            {
                CharT left = chars[i];
                CharT 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<CharT>& 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<CharT>& 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<CharT>> Split(CharT c)
        {
            List<String<CharT>> 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(CharT x) const
    {
        return Find(x, 0);
    }
    public nothrow int Find(CharT 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(CharT x) const
    {
        return RFind(x, len - 1);
    }
    public nothrow int RFind(CharT 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<CharT>& s) const
    {
        return Find(s, 0);
    }
    public nothrow int Find(const String<CharT>& s, int start) const
    {
        #assert(start >= 0);
        if (s.IsEmpty()) return start;
        int n = s.Length();
```

```
        CharT 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<CharT>& s) const
    {
        return RFind(s, len - 1);
    }
    public nothrow int RFind(const String<CharT>& s, int start) const
    {
        #assert(start < len);
        if (s.IsEmpty()) return start;
        int n = s.Length();
        CharT 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 CharT* 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)));
            CharT* newChars = cast<CharT*>(MemAlloc(sizeof(CharT) * 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 CharT* chars;
    }

    public typedef String<char> string;
    public typedef String<wchar> wstring;
    public typedef String<uchar> ustring;

    public nothrow String<CharT> operator+<CharT>(const String<CharT>&
        first, const String<CharT>& second)
    {
        String<CharT> temp(first);
        temp.Append(second);
        return temp;
    }
```

```
public nothrow String<CharT> operator+<CharT>(const String<CharT>&
    first, const CharT* second)
{
    String<CharT> temp(first);
    temp.Append(second);
    return temp;
}

public nothrow String<CharT> operator+<CharT>(const CharT* first,
    const String<CharT>& second)
{
    String<CharT> temp(first);
    temp.Append(second);
    return temp;
}

public string ToLower(const string& s)
{
    ustring result;
    ustring utf32 = System.Unicode.ToUtf32(s);
    for (uchar c : utf32)
    {
        result.Append(System.Unicode.ToLower(c));
    }
    return System.Unicode.ToUtf8(result);
}

public string ToUpper(const string& s)
{
    ustring result;
    ustring utf32 = System.Unicode.ToUtf32(s);
    for (uchar c : utf32)
    {
        result.Append(System.Unicode.ToUpper(c));
    }
    return System.Unicode.ToUtf8(result);
}

public wstring ToLower(const wstring& s)
{
    ustring result;
    ustring utf32 = System.Unicode.ToUtf32(s);
    for (uchar c : utf32)
    {
        result.Append(System.Unicode.ToLower(c));
    }
    return System.Unicode.ToUtf16(result);
}

public wstring ToUpper(const wstring& s)
{
    ustring result;
    ustring utf32 = System.Unicode.ToUtf32(s);
```

```
        for (uchar c : utf32)
        {
            result.Append(System.Unicode.ToUpper(c));
        }
        return System.Unicode.ToUtf16(result);
    }

    public ustring ToLower(const ustring& s)
    {
        ustring result;
        for (uchar c : s)
        {
            result.Append(System.Unicode.ToLower(c));
        }
        return result;
    }

    public ustring ToUpper(const ustring& s)
    {
        ustring result;
        for (uchar c : s)
        {
            result.Append(System.Unicode.ToUpper(c));
        }
        return result;
    }

    public bool LastComponentsEqual<CharT>(const String<CharT>& s0, const
        String<CharT>& s1, CharT componentSeparator)
    {
        List<String<CharT>> c0 = s0.Split(componentSeparator);
        List<String<CharT>> 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;
    }
}
```