

list.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;  
using System.Support;  
using System.Concepts;  
  
namespace System.Collections  
{  
    public class List<T> where T is Semiregular  
    {  
        public typedef T ValueType;  
        private typedef List<ValueType> Self;  
        public typedef RandomAccessIter<ValueType, const ValueType&,  
            const ValueType*> ConstIterator;  
        public typedef RandomAccessIter<ValueType, ValueType&, ValueType  
            *> Iterator;  
  
        public nothrow List(): items(null), count(0), res(0)  
        {  
        }  
        public List(const Self& that): items(null), count(that.count),  
            res(0) where T is Copyable  
        {  
            if (count > 0)  
            {  
                Reserve(count);  
                ConstructiveCopy(items, that.items, count);  
            }  
        }  
        public nothrow List(Self&& that): items(that.items), count(that.  
            count), res(that.res) where T is Movable  
        {  
            that.items = null;  
            that.count = 0;  
        }  
    }  
}
```

```

        that.res = 0;
    }
    public List(int n, const ValueType& value): items(nullptr), count(0)
        , res(0) where T is Copyable
    {
        #assert(n >= 0);
        count = n;
        Reserve(count);
        for (int i = 0; i < n; ++i)
        {
            construct<ValueType>(items + i, value);
        }
    }
    public void operator=(const Self& that) where T is Copyable
    {
        Destroy();
        count = that.count;
        Reserve(count);
        ConstructiveCopy(items, that.items, count);
    }
    public default nothrow void operator=(Self&& that) where T is
        Movable;
    public nothrow ~List()
    {
        Destroy();
    }
    public void Reserve(int minRes)
    {
        if (minRes > res)
        {
            Grow(minRes);
        }
    }
    public void Resize(int newCount) where T is Copyable
    {
        #assert(newCount >= 0);
        if (newCount != count)
        {
            if (newCount < count)
            {
                for (int i = newCount; i < count; ++i)
                {
                    destroy(items + i);
                }
            }
            else if (newCount > count)
            {
                Reserve(newCount);
                ValueType item;
                for (int i = count; i < newCount; ++i)
                {
                    construct<ValueType>(items + i, item);
                }
            }
        }
    }

```

```

        }
        count = newCount;
    }
}
public nothrow inline int Count() const
{
    return count;
}
public nothrow inline int Capacity() const
{
    return res;
}
public nothrow inline bool IsEmpty() const
{
    return count == 0;
}
public nothrow void Clear()
{
    Destroy();
}
public void Add(const ValueType& item) where T is Copyable
{
    Reserve(count + 1);
    construct<ValueType>(items + count, item);
    ++count;
}
public void Add(ValueType&& item) where T is Movable
{
    Reserve(count + 1);
    construct<ValueType>(items + count, item);
    ++count;
}
public Iterator Insert(Iterator pos, const ValueType& item) where
    T is Copyable
{
    int p = pos - Begin();
    Reserve(count + 1);
    pos = Begin() + p;
    Iterator end = End();
    if (count > 0)
    {
        construct<ValueType>(end.GetPtr(), ValueType());
        MoveBackward(pos, end, end + 1);
        *pos = item;
    }
    else
    {
        construct<ValueType>(end.GetPtr(), item);
        pos = end;
    }
    ++count;
    return pos;
}

```

```

public Iterator Insert(Iterator pos, ValueType&& item)   where T
    is Movable
{
    int p = pos - Begin();
    Reserve(count + 1);
    pos = Begin() + p;
    Iterator end = End();
    if (count > 0)
    {
        construct<ValueType>(end.GetPtr(), ValueType());
        MoveBackward(pos, end, end + 1);
        *pos = item;
    }
    else
    {
        construct<ValueType>(end.GetPtr(), item);
        pos = end;
    }
    ++count;
    return pos;
}
public Iterator InsertFront(const ValueType& item)   where T is
    Copyable
{
    return Insert(Begin(), item);
}
public Iterator InsertFront(ValueType&& item)   where T is
    Movable
{
    return Insert(Begin(), item);
}
public ValueType Remove(Iterator pos)
{
    #assert(pos >= Begin() && pos < End());
    ValueType result = Rvalue(*pos);
    Move(pos + 1, End(), pos);
    --count;
    Iterator end = End();
    destroy(end.GetPtr());
    return result;
}
public ValueType RemoveFirst()
{
    return Remove(Begin());
}
public ValueType RemoveLast()
{
    #assert(!IsEmpty());
    --count;
    Iterator end = End();
    ValueType result = Rvalue(*end);
    destroy(end.GetPtr());
    return result;
}

```

```

}
public nothrow const ValueType& operator [](int index) const
{
    #assert(index >= 0 && index < count);
    return items[index];
}
public nothrow ValueType& operator [](int index)
{
    #assert(index >= 0 && index < count);
    return items[index];
}
public nothrow Iterator Begin()
{
    return Iterator(items);
}
public nothrow ConstIterator Begin() const
{
    return ConstIterator(items);
}
public nothrow ConstIterator CBegin() const
{
    return ConstIterator(items);
}
public nothrow Iterator End()
{
    if (items != null)
    {
        return Iterator(items + count);
    }
    return Iterator(null);
}
public nothrow ConstIterator End() const
{
    if (items != null)
    {
        return ConstIterator(items + count);
    }
    return ConstIterator(null);
}
public nothrow ConstIterator CEnd() const
{
    if (items != null)
    {
        return ConstIterator(items + count);
    }
    return ConstIterator(null);
}
public nothrow const ValueType& Front() const
{
    #assert(!IsEmpty());
    return *Begin();
}
public nothrow ValueType& Front()

```

```

{
    #assert (!IsEmpty());
    return *Begin();
}
public nothrow const ValueType& Back() const
{
    #assert (!IsEmpty());
    return *(End() - 1);
}
public nothrow ValueType& Back()
{
    #assert (!IsEmpty());
    return *(End() - 1);
}
private void Grow(int minRes)
{
    minRes = cast<int>(MemGrow(cast<ulong>(minRes)));
    ValueType* newItems = cast<ValueType*>(MemAlloc(cast<ulong>(
        minRes) * cast<ulong>(sizeof(ValueType))));
    if (items != null)
    {
        ConstructiveMove(newItems, items, count);
        int saveCount = count;
        Destroy();
        count = saveCount;
    }
    items = newItems;
    res = minRes;
}
private nothrow void Destroy()
{
    if (count > 0)
    {
        Destroy(items, count);
        count = 0;
    }
    if (res > 0)
    {
        MemFree(items);
        items = null;
        res = 0;
    }
}
private ValueType* items;
private int count;
private int res;
}

public nothrow bool operator==(T>)(const List<T>& left, const List<T>&
    right) where T is Regular
{
    int n = left.Count();
    if (n != right.Count())

```

```

    {
        return false;
    }
    for (int i = 0; i < n; ++i)
    {
        if (left[i] != right[i])
        {
            return false;
        }
    }
    return true;
}

public nothrow bool operator<<T>(const List<T>& left, const List<T>&
right) where T is TotallyOrdered
{
    return LexicographicalCompare(left.Begin(), left.End(), right.
Begin(), right.End());
}

public void ConstructiveCopy<ValueType>(ValueType* to, ValueType*
from, int count) where ValueType is CopyConstructible
{
    for (int i = 0; i < count; ++i)
    {
        construct<ValueType>(to, *from);
        ++to;
        ++from;
    }
}

public void ConstructiveMove<ValueType>(ValueType* to, ValueType*
from, int count) where ValueType is MoveConstructible
{
    for (int i = 0; i < count; ++i)
    {
        construct<ValueType>(to, Rvalue(*from));
        ++to;
        ++from;
    }
}

public nothrow void Destroy<ValueType>(ValueType* items, int count)
where ValueType is Destructible
{
    for (int i = 0; i < count; ++i)
    {
        destroy(items);
        ++items;
    }
}
}

```