ptr.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

namespace System
{
    public class UniquePtr<T>
    {
        private typedef UniquePtr<T> Self;

        public nothrow UniquePtr(): ptr(null)
        {
        }
        public nothrow UniquePtr(T* ptr_): ptr(ptr_)
        {
        }
        suppress UniquePtr(const Self&);
        public nothrow UniquePtr(Self&& that): ptr(that.Release())
        {
        }
        public nothrow void operator=(T* ptr_)
        {
            if (ptr != null)
            {
                delete ptr;
            }
            ptr = ptr_;
        }
        suppress void operator=(const Self&);
        public nothrow void operator=(Self&& that)
        {
            if (ptr != null)
            {
                delete ptr;
            }
            ptr = that.Release();
```

```
        }
        public nothrow ~UniquePtr()
        {
            if (ptr != null)
            {
                delete ptr;
            }
        }
        public nothrow void Reset()
        {
            if (ptr != null)
            {
                delete ptr;
                ptr = null;
            }
        }
        public nothrow void Reset(T* ptr_)
        {
            if (ptr != null)
            {
                delete ptr;
            }
            ptr = ptr_;
        }
        public inline nothrow T* Release()
        {
            T* ptr_ = ptr;
            ptr = null;
            return ptr_;
        }
        public inline nothrow T* GetPtr() const
        {
            return ptr;
        }
        public inline nothrow bool IsNull() const
        {
            return ptr == null;
        }
        public inline nothrow T* operator->() const
        {
            #assert(ptr != null);
            return ptr;
        }
        public inline nothrow T& operator*() const
        {
            #assert(ptr != null);
            return *ptr;
        }
        public nothrow void Swap(Self& that)
        {
            Swap(ptr, that.ptr);
        }
        private T* ptr;
```

```
}

public nothrow bool operator==<T>(const UniquePtr<T>& left , const
    UniquePtr<T>& right )
{
    return  left . GetPtr ( ) == right . GetPtr ( ) ;
}

public nothrow bool operator<<T>(const UniquePtr<T>& left , const
    UniquePtr<T>& right )
{
    return  left . GetPtr ( ) < right . GetPtr ( ) ;
}

public abstract class CounterBase
{
    public nothrow CounterBase ( ) :  useCount ( 1 ) , weakCount ( 1 )
    {
    }
    suppress CounterBase ( const CounterBase&) ;
    suppress void operator=(const CounterBase&) ;
    suppress CounterBase ( CounterBase&&) ;
    suppress void operator=(CounterBase&&) ;
    public nothrow virtual  ~CounterBase ( )
    {
    }
    public abstract nothrow void Dispose ( ) ;
    public virtual nothrow void Destruct ( )
    {
        delete this ;
    }
    public inline nothrow void AddReference ( )
    {
        #assert ( ! ( useCount == 0 && weakCount != 0 ) ) ;
        ++useCount ;
        ++weakCount ;
    }
    public inline nothrow void Release ( )
    {
        --useCount ;
        if ( useCount != 0 )
        {
            --weakCount ;
            return ;
        }
        Dispose ( ) ;
        WeakRelease ( ) ;
    }
    public nothrow inline void WeakAddReference ( )
    {
        ++weakCount ;
    }
    public nothrow void WeakRelease ( )
```

```
        {
            −−weakCount;
            if (weakCount == 0)
            {
                Destruct();
            }
        }
        public nothrow inline int GetUseCount() const
        {
            return useCount;
        }
        private int useCount;
        private int weakCount;
    }

    public class Counter<T> : CounterBase
    {
        private typedef Counter<T> Self;

        public nothrow Counter(T* ptr_): ptr(ptr_)
        {
        }
        suppress Counter(const Self&);
        suppress void operator=(const Self&);
        suppress Counter(Self&&);
        suppress void operator=(Self&&);
        public override nothrow void Dispose()
        {
            delete ptr;
            ptr = null;
        }
        private T* ptr;
    }

    public class SharedCount<T>
    {
        private typedef Counter<T>* CounterPtrType;
        private typedef SharedCount<T> Self;

        public nothrow SharedCount(): counter(null)
        {
        }
        public nothrow SharedCount(T* ptr_): counter(new Counter<T>(ptr_)
            )
        {
        }
        public nothrow SharedCount(Counter<T>* counter_): counter(
            counter_) // to support SharedCountCast
        {
            if (counter != null)
            {
                counter−>AddReference();
            }
```

```
}
public nothrow SharedCount(const Self& that): counter(that.
    counter)
{
    if (counter != null)
    {
        counter->AddReference();
    }
}
public nothrow SharedCount(Self&& that) : counter(that.counter)
{
    that.counter = null;
}
public nothrow SharedCount(const WeakCount<T>& that): counter(
    that.GetCounter())
{
    #assert(counter != null);
    counter->AddReference();
}
public nothrow void operator=(const Self& that)
{
    CounterPtrType otherCounter = that.counter;
    if (otherCounter != null)
    {
        otherCounter->AddReference();
    }
    if (counter != null)
    {
        counter->Release();
    }
    counter = otherCounter;
}
public default void operator=(Self&&);
public nothrow ~SharedCount()
{
    if (counter != null)
    {
        counter->Release();
    }
}
public nothrow void Swap(Self& that)
{
    CounterPtrType otherCounter = that.counter;
    that.counter = counter;
    counter = otherCounter;
}
public nothrow int GetUseCount() const
{
    if (counter != null)
    {
        return counter->GetUseCount();
    }
    return 0;
```

```
        }
        public nothrow bool IsUnique() const
        {
            return GetUseCount() == 1;
        }
        public nothrow CounterPtrType GetCounter() const
        {
            return counter;
        }
        private CounterPtrType counter;
    }

    public nothrow bool operator==<T>(const SharedCount<T>& left, const
        SharedCount<T>& right)
    {
        return left.GetCounter() == right.GetCounter();
    }

    public nothrow bool operator<<T>(const SharedCount<T>& left, const
        SharedCount<T>& right)
    {
        return left.GetCounter() < right.GetCounter();
    }

    internal nothrow SharedCount<U> SharedCountCast<U, T>(const
        SharedCount<T>& from)
    {
        return SharedCount<U>(cast<Counter<U>*>(from.GetCounter()));
    }

    public class WeakCount<T>
    {
        private typedef Counter<T>* CounterPtrType;
        private typedef WeakCount<T> Self;

        public nothrow WeakCount(): counter(null)
        {
        }
        public nothrow WeakCount(const Self& that): counter(that.counter)
        {
            if (counter != null)
            {
                counter->WeakAddReference();
            }
        }
        public nothrow WeakCount(Self&& that) : counter(that.counter)
        {
            that.counter = null;
        }
        public nothrow WeakCount(const SharedCount<T>& that): counter(
            that.GetCounter())
        {
            if (counter != null)
```

```
                {
                    counter->WeakAddReference();
                }
            }
            public nothrow ~WeakCount()
            {
                if (counter != null)
                {
                    counter->WeakRelease();
                }
            }
            public nothrow void operator=(const SharedCount<T>& that)
            {
                CounterPtrType otherCounter = that.GetCounter();
                if (otherCounter != null)
                {
                    otherCounter->WeakAddReference();
                }
                if (counter != null)
                {
                    counter->WeakRelease();
                }
                counter = otherCounter;
            }
            public nothrow void operator=(const Self& that)
            {
                CounterPtrType otherCounter = that.counter;
                if (otherCounter != null)
                {
                    otherCounter->WeakAddReference();
                }
                if (counter != null)
                {
                    counter->WeakRelease();
                }
                counter = otherCounter;
            }
            default nothrow void operator=(Self&&);
            public nothrow void Swap(Self& that)
            {
                CounterPtrType otherCounter = that.counter;
                that.counter = counter;
                counter = otherCounter;
            }
            public nothrow int GetUseCount() const
            {
                if (counter != null)
                {
                    return counter->GetUseCount();
                }
                return 0;
            }
            public nothrow CounterPtrType GetCounter() const
```

```
        {
            return counter;
        }
        private CounterPtrType counter;
    }

    public nothrow bool operator==<T>(const WeakCount<T>& left, const
        WeakCount<T>& right)
    {
        return left.GetCounter() == right.GetCounter();
    }

    public nothrow bool operator<<T>(const WeakCount<T>& left, const
        WeakCount<T>& right)
    {
        return left.GetCounter() < right.GetCounter();
    }

    public class SharedPtr<T>
    {
        private typedef SharedPtr<T> Self;
        private typedef SharedCount<T> CountType;

        public nothrow SharedPtr(): ptr(null), count()
        {
        }
        public nothrow explicit SharedPtr(T* ptr_): ptr(ptr_), count(ptr)
        {
            EnableSharedFromThis(ptr_, ptr_, count);
        }
        public nothrow SharedPtr(T* ptr_, const CountType& count_): ptr(
            ptr_), count(count_) // to support PtrCast
        {
        }
        public nothrow SharedPtr(const Self& that): ptr(that.ptr), count(
            that.count)
        {
        }
        public nothrow SharedPtr(Self&& that) : ptr(that.ptr), count(
            Rvalue(that.count))
        {
            that.ptr = null;
        }
        public nothrow SharedPtr(const WeakPtr<T>& that): ptr(), count(
            that.GetCount())
        {
            ptr = that.GetPtr();
        }
        public nothrow void Reset()
        {
            Self().Swap(*this);
        }
        public nothrow void Reset(T* ptr_)
```

```
        {
            Self(ptr_).Swap(*this);
        }
        public nothrow void operator=(const Self& that)
        {
            ptr = that.ptr;
            count = that.count;
        }
        public default nothrow void operator=(Self&&);
        public nothrow inline T* operator->() const
        {
            #assert(ptr != null);
            return ptr;
        }
        public nothrow inline T& operator*() const
        {
            #assert(ptr != null);
            return *ptr;
        }
        public nothrow inline T* GetPtr() const
        {
            return ptr;
        }
        public nothrow inline const CountType& GetCount() const
        {
            return count;
        }
        public nothrow inline bool IsNull() const
        {
            return ptr == null;
        }
        public nothrow void Swap(Self& that)
        {
            Swap(ptr, that.ptr);
            count.Swap(that.count);
        }
        public nothrow bool IsUnique() const
        {
            return count.IsUnique();
        }
        public nothrow int GetUseCount() const
        {
            return count.GetUseCount();
        }
        private T* ptr;
        private CountType count;
    }

    public nothrow bool operator==<T>(const SharedPtr<T>& left, const
        SharedPtr<T>& right)
    {
        return left.GetPtr() == right.GetPtr();
    }
```

```
public nothrow bool operator<<T>(const SharedPtr<T>& left , const
    SharedPtr<T>& right )
{
    #assert ( left . GetPtr () != null && right . GetPtr () != null || left .
        GetPtr () == null && right . GetPtr () == null );
    return left . GetPtr () < right . GetPtr ();
}

public nothrow SharedPtr<U> PtrCast<U, T>(const SharedPtr<T>& from )
{
    return SharedPtr<U>(cast<U∗>(from . GetPtr ()), SharedCountCast<U>(
        from . GetCount ()));
}

public class WeakPtr<T>
{
    private typedef WeakPtr<T> Self ;
    private typedef WeakCount<T> CountType ;

    public nothrow WeakPtr (): ptr (null), count ()
    {
    }
    public nothrow WeakPtr(const SharedPtr<T>& that ): ptr (that . GetPtr
        ()), count(that . GetCount ())
    {
    }
    public nothrow WeakPtr(const Self& that ): ptr (), count(that . count
        )
    {
        ptr = that . Lock (). GetPtr ();
    }
    public default ~WeakPtr ();
    public nothrow void operator=(const Self& that )
    {
        ptr = that . Lock (). GetPtr ();
        count = that . count ;
    }
    public nothrow void operator=(const SharedPtr<T>& that )
    {
        ptr = that . GetPtr ();
        count = that . GetCount ();
    }
    public nothrow int GetUseCount () const
    {
        return count . GetUseCount ();
    }
    public nothrow bool IsExpired () const
    {
        return count . GetUseCount () == 0;
    }
    public nothrow SharedPtr<T> Lock () const
    {
```

```
            if (IsExpired())
            {
                return SharedPtr<T>();
            }
            return SharedPtr<T>(*this);
        }
        public nothrow void Reset()
        {
            Self().Swap(*this);
        }
        public nothrow void Swap(Self& that)
        {
            Swap(ptr, that.ptr);
            count.Swap(that.count);
        }
        public inline nothrow const CountType& GetCount() const
        {
            return count;
        }
        public inline nothrow T* GetPtr() const
        {
            return ptr;
        }
        public nothrow void Assign(T* ptr_, const SharedCount<T>& count_)
                // to support EnableSharedFromThis
        {
            ptr = ptr_;
            count = count_;
        }
        private T* ptr;
        private CountType count;
    }

    public class ShareableFromThis<T>
    {
        public nothrow SharedPtr<T> GetSharedFromThis() const
        {
            SharedPtr<T> p(weakThis);
            #assert(p.GetPtr() == this);
            return p;
        }
        public nothrow WeakPtr<T>& GetWeakThis()
        {
            return weakThis;
        }
        private WeakPtr<T> weakThis;
    }

    public nothrow inline void EnableSharedFromThis<T>(void*, void*,
        const SharedCount<T>&)
    {
    }
```

```
    public nothrow void EnableSharedFromThis<T, U>(ShareableFromThis<T>*
        left, U* right, const SharedCount<U>& count)
    {
        if (left != null)
        {
            left->GetWeakThis().Assign(cast<T*>(right), SharedCountCast<T
                >(count));
        }
    }
}
```