LinksPlatform's Platform.Memory Class Library

1.1 ./Platform.Memory/ArrayMemory.cs

```csharp
using System.Runtime.CompilerServices;

namespace Platform.Memory
{
    /// <summary>
    /// <para>Represents a memory block with access via indexer.</para>
    /// <para>Представляет блок памяти с доступом через индексатор.</para>
    /// </summary>
    /// <typeparam name="TElement"><para>Element type.</para><para>Тип элемента.</para></typeparam>
    public class ArrayMemory<TElement> : IArrayMemory<TElement>
    {
        #region Fields

        private readonly TElement[] _array;

        #endregion

        #region Properties

        /// <inheritdoc/>
        /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml' path='doc/members/member[@name="P:Platform.Memory.IMemory.Size"]/*'/>
        public long Size
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _array.Length;
        }

        /// <inheritdoc/>
        /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml' path='doc/members/member[@name="P:Platform.Memory.IArrayMemory`1.Item(System.Int64)"]/*'/>
        public TElement this[long index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _array[index];
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set => _array[index] = value;
        }

        #endregion

        #region Constuctors

        /// <summary>
        /// <para>Initializes a new instance of the <see cref="ArrayMemory{TElement}"/> class.</para>
        /// <para>Инициализирует новый экземпляр класса <see cref="ArrayMemory{TElement}"/>.</para>
        /// </summary>
        /// <param name="size"><para>Size in bytes.</para><para>Размер в байтах.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayMemory(long size) => _array = new TElement[size];

        #endregion
    }
}
```

1.2 ./Platform.Memory/DirectMemoryAsArrayMemoryAdapter.cs

```csharp
using System;
using System.Runtime.CompilerServices;
using Platform.Disposables;
using Platform.Exceptions;
using Platform.Unsafe;

namespace Platform.Memory
{
    /// <summary>
    /// <para>Represents adapter to a memory block with access via indexer.</para>
    /// <para>Представляет адаптер к блоку памяти с доступом через индексатор.</para>
    /// </summary>
    /// <typeparam name="TElement"><para>Element type.</para><para>Тип элемента.</para></typeparam>
    public class DirectMemoryAsArrayMemoryAdapter<TElement> : DisposableBase, IArrayMemory<TElement>, IDirectMemory
        where TElement : struct
    {
        #region Fields

        private readonly IDirectMemory _memory;

        #endregion

        #region Properties

        /// <inheritdoc/>
        /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml'
         ↪  path='doc/members/member[@name="P:Platform.Memory.IMemory.Size"]/*'/>
        public long Size
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _memory.Size;
        }

        /// <inheritdoc/>
        /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml'
         ↪  path='doc/members/member[@name="P:Platform.Memory.IDirectMemory.Pointer"]/*'/>
        public IntPtr Pointer
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _memory.Pointer;
        }

        /// <inheritdoc/>
        /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml' path='doc/members/mem⌋
         ↪  ber[@name="P:Platform.Memory.IArrayMemory`1.Item(System.Int64)"]/*'/>
        public TElement this[long index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => Pointer.ReadElementValue<TElement>(index);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set => Pointer.WriteElementValue(index, value);
        }

        #endregion

        #region DisposableBase Properties

        /// <inheritdoc/>
        protected override string ObjectName
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => $"Array as memory block at '{Pointer}' address.";
        }

        #endregion

        #region Constructors

        /// <summary>
        /// <para>Initializes a new instance of the <see
         ↪  cref="DirectMemoryAsArrayMemoryAdapter{TElement}"/> class.</para>
        /// <para>Инициализирует новый экземпляр класса <see
         ↪  cref="DirectMemoryAsArrayMemoryAdapter{TElement}"/>.</para>
        /// </summary>
        /// <param name="memory"><para>An object implementing <see cref="IDirectMemory"/>
         ↪  interface.</para><para>Объект, реализующий интерфейс <see
         ↪  cref="IDirectMemory"/>.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DirectMemoryAsArrayMemoryAdapter(IDirectMemory memory)
        {
            Ensure.Always.ArgumentNotNull(memory, nameof(memory));
            Ensure.Always.ArgumentMeetsCriteria(memory, m => (m.Size % Structure<TElement>.Size)
             ↪  == 0, nameof(memory), "Memory is not aligned to element size.");
            _memory = memory;
        }

        #endregion

        #region DisposableBase Methods

        /// <inheritdoc/>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                _memory.DisposeIfPossible();
            }
        }

        #endregion
    }
}
```

1.3 ./Platform.Memory/FileArrayMemory.cs

```
using System.IO;
using System.Runtime.CompilerServices;
using Platform.Disposables;
using Platform.Unsafe;
using Platform.IO;

namespace Platform.Memory
{
    /// <summary>
    /// <para>Represents a memory block with access via indexer and stored as file on
    ↪  disk.</para>
    /// <para>Представляет блок памяти с доступом через индексатор и хранящийся в виде файла на
    ↪  диске.</para>
    /// </summary>
    /// <typeparam name="TElement"><para>Element type.</para><para>Тип
    ↪  элемента.</para></typeparam>
    public class FileArrayMemory<TElement> : DisposableBase, IArrayMemory<TElement> //-V3073
        where TElement : struct
    {
        #region Fields

        private readonly FileStream _file;

        #endregion

        #region Properties

        /// <inheritdoc/>
        /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml'
        ↪  path='doc/members/member[@name="P:Platform.Memory.IMemory.Size"]/*'/>
        public long Size
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _file.Length;
        }

        /// <inheritdoc/>
        /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml' path='doc/members/mem
        ↪  ber[@name="P:Platform.Memory.IArrayMemory`1.Item(System.Int64)"]/*'/>
        public TElement this[long index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                _file.Seek(Structure<TElement>.Size * index, SeekOrigin.Begin);
                return _file.ReadOrDefault<TElement>();
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set
            {
                _file.Seek(Structure<TElement>.Size * index, SeekOrigin.Begin);
                _file.Write(value);
            }
        }

        #endregion

        #region DisposableBase Properties

        /// <inheritdoc/>
        protected override string ObjectName
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => $"File stored memory block at '{_file.Name}' path.";
        }

        #endregion

        #region Contructors

        /// <summary>
        /// <para>Initializes a new instance of the <see cref="FileArrayMemory{TElement}"/>
         ↪  class.</para>
        /// <para>Инициализирует новый экземпляр класса <see
         ↪  cref="FileArrayMemory{TElement}"/>.</para>
        /// </summary>
        /// <param name="file"><para>File stream.</para><para>Файловый поток.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public FileArrayMemory(FileStream file) => _file = file;

        /// <summary>
        /// <para>Initializes a new instance of the <see cref="FileArrayMemory{TElement}"/>
         ↪  class.</para>
        /// <para>Инициализирует новый экземпляр класса <see
         ↪  cref="FileArrayMemory{TElement}"/>.</para>
        /// </summary>
        /// <param name="path"><para>An path to file.</para><para>Путь к файлу.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public FileArrayMemory(string path) : this(File.Open(path, FileMode.OpenOrCreate)) { }

        #endregion

        #region DisposableBase Methods

        /// <inheritdoc/>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                _file.DisposeIfPossible();
            }
        }

        #endregion
    }
}
```

1.4 ./Platform.Memory/FileMappedResizableDirectMemory.cs

```
using System;
using System.IO;
using System.IO.MemoryMappedFiles;
using System.Runtime.CompilerServices;
using Platform.Disposables;
using Platform.Exceptions;
using Platform.Collections;
using Platform.IO;

namespace Platform.Memory
{
    /// <summary>
    /// <para>Represents a memory block stored as a file on disk.</para>
    /// <para>Представляет блок памяти, хранящийся в виде файла на диске.</para>
    /// </summary>
    public unsafe class FileMappedResizableDirectMemory : ResizableDirectMemoryBase
    {
        #region Fields

        private MemoryMappedFile _file;
        private MemoryMappedViewAccessor _accessor;

        /// <summary>
        /// <para>Gets path to memory mapped file.</para>
        /// <para>Получает путь к отображенному в памяти файлу.</para>
        /// </summary>
        protected readonly string Path;

        #endregion

        #region DisposableBase Properties

        /// <inheritdoc/>
        protected override string ObjectName
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => $"File stored memory block at '{Path}' path.";
        }

        #endregion

        #region Constructors

        /// <summary>
        /// <para>Initializes a new instance of the <see cref="FileMappedResizableDirectMemory"/> class.</para>
        /// <para>Инициализирует новый экземпляр класса <see cref="FileMappedResizableDirectMemory"/>.</para>
        /// </summary>
        /// <param name="path"><para>An path to file.</para><para>Путь к файлу.</para></param>
        /// <param name="minimumReservedCapacity"><para>Minimum file size in bytes.</para><para>Минимальный размер файла в байтах.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public FileMappedResizableDirectMemory(string path, long minimumReservedCapacity)
        {
            Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(path, nameof(path));
            if (minimumReservedCapacity < MinimumCapacity)
            {
                minimumReservedCapacity = MinimumCapacity;
            }
            Path = path;
            var size = FileHelpers.GetSize(Path);
            ReservedCapacity = size > minimumReservedCapacity ? ((size / minimumReservedCapacity) + 1) * minimumReservedCapacity : minimumReservedCapacity;
            UsedCapacity = size;
        }

        /// <summary>
        /// <para>Initializes a new instance of the <see cref="FileMappedResizableDirectMemory"/> class.</para>
        /// <para>Инициализирует новый экземпляр класса <see cref="FileMappedResizableDirectMemory"/>.</para>
        /// </summary>
        /// <param name="address"><para>An path to file.</para><para>Путь к файлу.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public FileMappedResizableDirectMemory(string address) : this(address, MinimumCapacity) { }

        #endregion

        #region Methods

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void MapFile(long capacity)
        {
            if (Pointer != IntPtr.Zero)
            {
                return;
            }
            _file = MemoryMappedFile.CreateFromFile(Path, FileMode.Open, mapName: null, capacity, MemoryMappedFileAccess.ReadWrite);
            _accessor = _file.CreateViewAccessor();
            byte* pointer = null;
            _accessor.SafeMemoryMappedViewHandle.AcquirePointer(ref pointer);
            Pointer = new IntPtr(pointer);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void UnmapFile()
        {
            if (UnmapFile(Pointer))
            {
                Pointer = IntPtr.Zero;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool UnmapFile(IntPtr pointer)
        {
            if (pointer == IntPtr.Zero)
            {
                return false;
            }
            if (_accessor != null)
            {
                _accessor.SafeMemoryMappedViewHandle.ReleasePointer();
                Disposable.TryDisposeAndResetToDefault(ref _accessor);
            }
```

```
                Disposable.TryDisposeAndResetToDefault(ref _file);
                return true;
            }

            #endregion

            #region ResizableDirectMemoryBase Methods

            /// <inheritdoc/>
            /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml'
            ↪   path='doc/members/member[@name="M:Platform.Memory.ResizableDirectMemoryBase.OnReserv ⌋
            ↪   edCapacityChanged(System.Int64,System.Int64)"]/*'/>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override void OnReservedCapacityChanged(long oldReservedCapacity, long
            ↪   newReservedCapacity)
            {
                UnmapFile();
                FileHelpers.SetSize(Path, newReservedCapacity);
                MapFile(newReservedCapacity);
            }

            /// <inheritdoc/>
            /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml'
            ↪   path='doc/members/member[@name="M:Platform.Memory.ResizableDirectMemoryBase.DisposeP ⌋
            ↪   ointer(System.IntPtr,System.Int64)"]/*'/>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override void DisposePointer(IntPtr pointer, long usedCapacity)
            {
                if (UnmapFile(pointer))
                {
                    FileHelpers.SetSize(Path, usedCapacity);
                }
            }

            #endregion
        }
    }
```

1.5 ./Platform.Memory/HeapResizableDirectMemory.cs

```
using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using Platform.Unsafe;

namespace Platform.Memory
{
    /// <summary>
    /// <para>Represents a memory block allocated in Heap.</para>
    /// <para>Представляет блок памяти, выделенный в "куче".</para>
    /// </summary>
    public unsafe class HeapResizableDirectMemory : ResizableDirectMemoryBase
    {
        #region DisposableBase Properties

        /// <inheritdoc/>
        protected override string ObjectName
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => $"Heap stored memory block at {Pointer} address.";
        }

        #endregion

        #region Constructors

        /// <summary>
        /// <para>Initializes a new instance of the <see cref="HeapResizableDirectMemory"/>
        ↪   class.</para>
        /// <para>Инициализирует новый экземпляр класса <see
        ↪   cref="HeapResizableDirectMemory"/>.</para>
        /// </summary>
        /// <param name="minimumReservedCapacity"><para>Minimum file size in
        ↪   bytes.</para><para>Минимальный размер файла в байтах.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HeapResizableDirectMemory(long minimumReservedCapacity)
        {
            if (minimumReservedCapacity < MinimumCapacity)
            {
                minimumReservedCapacity = MinimumCapacity;
            }
            ReservedCapacity = minimumReservedCapacity;
            UsedCapacity = 0;
        }

        /// <summary>
        /// <para>Initializes a new instance of the <see cref="HeapResizableDirectMemory"/> class.</para>
        /// <para>Инициализирует новый экземпляр класса <see cref="HeapResizableDirectMemory"/>.</para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HeapResizableDirectMemory() : this(MinimumCapacity) { }

        #endregion

        #region ResizableDirectMemoryBase Methods

        /// <inheritdoc/>
        /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml' path='doc/members/member[@name="M:Platform.Memory.ResizableDirectMemoryBase.DisposePointer(System.IntPtr,System.Int64)"]/*'/>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void DisposePointer(IntPtr pointer, long usedCapacity) => Marshal.FreeHGlobal(pointer);

        /// <inheritdoc/>
        /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml' path='doc/members/member[@name="M:Platform.Memory.ResizableDirectMemoryBase.OnReservedCapacityChanged(System.Int64,System.Int64)"]/*'/>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void OnReservedCapacityChanged(long oldReservedCapacity, long newReservedCapacity)
        {
            if (Pointer == IntPtr.Zero)
            {
                Pointer = Marshal.AllocHGlobal(new IntPtr(newReservedCapacity));
                MemoryBlock.Zero((void*)Pointer, newReservedCapacity);
            }
            else
            {
                Pointer = Marshal.ReAllocHGlobal(Pointer, new IntPtr(newReservedCapacity));
                var pointer = (byte*)Pointer + oldReservedCapacity;
                MemoryBlock.Zero(pointer, newReservedCapacity - oldReservedCapacity);
            }
        }

        #endregion
    }
}
```

1.6 ./Platform.Memory/IArrayMemory.cs

```
using System.Runtime.CompilerServices;

namespace Platform.Memory
{
    /// <summary>
    /// <para>Represents a memory block interface with access via indexer.</para>
    /// <para>Представляет интерфейс блока памяти с доступом через индексатор.</para>
    /// </summary>
    /// <typeparam name="TElement"><para>Element type.</para><para>Тип элемента.</para></typeparam>
    public interface IArrayMemory<TElement> : IMemory
    {
        /// <summary>
        /// <para>Gets or sets the element at the specified index.</para>
        /// <para>Возвращает или устанавливает элемент по указанному индексу.</para>
        /// </summary>
        /// <param name="index"><para>The index of the element to get or set.</para><para>Индекс элемента, который нужно получить или установить.</para></param>
        TElement this[long index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }
    }
}
```

1.7 ./Platform.Memory/IDirectMemory.cs

```
using System;
using System.Runtime.CompilerServices;

namespace Platform.Memory
{
    /// <summary>
    /// <para>Represents a memory block interface with direct access (via unmanaged
    ↪  pointers).</para>
    /// <para>Представляет интерфейс блока памяти с прямым доступом (через неуправляемые
    ↪  указатели).</para>
    /// </summary>
    public interface IDirectMemory : IMemory, IDisposable
    {
        /// <summary>
        /// <para>Gets the pointer to the beginning of this memory block.</para>
        /// <para>Возвращает указатель на начало блока памяти.</para>
        /// </summary>
        IntPtr Pointer
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }
    }
}
```

1.8 ./Platform.Memory/IMemory.cs

```
using System.Runtime.CompilerServices;

namespace Platform.Memory
{
    /// <summary>
    /// <para>Represents a memory block interface with size in bytes.</para>
    /// <para>Представляет интерфейс блока памяти с размером в байтах.</para>
    /// </summary>
    public interface IMemory
    {
        /// <summary>
        /// <para>Gets the size in bytes of this memory block.</para>
        /// <para>Возвращает размер блока памяти в байтах.</para>
        /// </summary>
        long Size
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }
    }
}
```

1.9 ./Platform.Memory/IResizableDirectMemory.cs

```
using System.Runtime.CompilerServices;

namespace Platform.Memory
{
    /// <summary>
    /// <para>Represents a resizable memory block interface with direct access (via unmanaged
    ↪  pointers).</para>
    /// <para>Представляет интерфейс блока памяти с изменяемым размером и прямым доступом (через
    ↪  неуправляемые указатели).</para>
    /// </summary>
    public interface IResizableDirectMemory : IDirectMemory
    {
        /// <summary>
        /// <para>Gets or sets the reserved capacity in bytes of this memory block.</para>
        /// <para>Возвращает или устаналивает зарезервированный размер блока памяти в
        ↪  байтах.</para>
        /// </summary>
        /// <remarks>
        /// <para>
        /// If less then zero the value is replaced with zero.
        /// Cannot be less than the used capacity of this memory block.
        /// </para>
        /// <para>
        /// Если меньше нуля, значение заменяется на ноль.
        /// Не может быть меньше используемой емкости блока памяти.
        /// </para>
        /// </remarks>
        long ReservedCapacity
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>Gets or sets the used capacity in bytes of this memory block.</para>
        /// <para>Возвращает или устанавливает используемый размер в блоке памяти (в
        ↪  байтах).</para>
        /// </summary>
        /// <remarks>
        /// <para>
        /// If less then zero the value is replaced with zero.
        /// Cannot be greater than the reserved capacity of this memory block.
        /// </para>
        /// <para>
        /// It is recommended to reduce the reserved capacity of the memory block to the used
        ↪  capacity (specified in this property) after the completion of the use of the memory
        ↪  block.
        /// </para>
        /// <para>
        /// Если меньше нуля, значение заменяется на ноль.
        /// Не может быть больше, чем зарезервированная емкость этого блока памяти.
        /// </para>
        /// <para>
        /// Рекомендуется уменьшать фактический размер блока памяти до используемого размера
        ↪  (указанного в этом свойстве) после завершения использования блока памяти.
        /// </para>
        /// </remarks>
        long UsedCapacity
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }
    }
}
```

1.10 ./Platform.Memory/ResizableDirectMemoryBase.cs

```
using System;
using System.Threading;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Disposables;
using Platform.Ranges;

namespace Platform.Memory
{
    /// <summary>
    /// <para>Provides a base implementation for the resizable memory block with direct access
    ↪  (via unmanaged pointers).</para>
    /// <para>Предоставляет базовую реализацию для блока памяти с изменяемым размером и прямым
    ↪  доступом (через неуправляемые указатели).</para>
    /// </summary>
    public abstract class ResizableDirectMemoryBase : DisposableBase, IResizableDirectMemory
    {
        #region Constants

        /// <summary>
        /// <para>Gets minimum capacity in bytes.</para>
        /// <para>Возвращает минимальную емкость в байтах.</para>
        /// </summary>
        public static readonly long MinimumCapacity = 4096;

        #endregion

        #region Fields

        private IntPtr _pointer;
        private long _reservedCapacity;
        private long _usedCapacity;

        #endregion

        #region Properties
```

```csharp

        /// <inheritdoc/>
        /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml' path='doc/members/member[@name="P:Platform.Memory.IMemory.Size"]/*'/>
        /// <exception cref="ObjectDisposedException"><para>The memory block is disposed.</para><para>Блок памяти уже высвобожден.</para></exception>
        public long Size
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                Ensure.Always.NotDisposed(this);
                return UsedCapacity;
            }
        }

        /// <inheritdoc/>
        /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml' path='doc/members/member[@name="P:Platform.Memory.IDirectMemory.Pointer"]/*'/>
        /// <exception cref="ObjectDisposedException"><para>The memory block is disposed.</para><para>Блок памяти уже высвобожден.</para></exception>
        public IntPtr Pointer
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                Ensure.Always.NotDisposed(this);
                return _pointer;
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected set
            {
                Ensure.Always.NotDisposed(this);
                _pointer = value;
            }
        }

        /// <inheritdoc/>
        /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml' path='doc/members/member[@name="P:Platform.Memory.IResizableDirectMemory.ReservedCapacity"]/*'/>
        /// <exception cref="ObjectDisposedException"><para>The memory block is disposed.</para><para>Блок памяти уже высвобожден.</para></exception>
        /// <exception cref="ArgumentOutOfRangeException"><para>Attempted to set the reserved capacity to a value that is less than the used capacity.</para><para>Была выполнена попытка установить зарезервированную емкость на значение, которое меньше используемой емкости.</para></exception>
        public long ReservedCapacity
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                Ensure.Always.NotDisposed(this);
                return _reservedCapacity;
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set
            {
                Ensure.Always.NotDisposed(this);
                if (value != _reservedCapacity)
                {
                    Ensure.Always.ArgumentInRange(value, new Range<long>(_usedCapacity, long.MaxValue));
                    OnReservedCapacityChanged(_reservedCapacity, value);
                    _reservedCapacity = value;
                }
            }
        }

        /// <inheritdoc/>
        /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml' path='doc/members/member[@name="P:Platform.Memory.IResizableDirectMemory.UsedCapacity"]/*'/>
        /// <exception cref="ObjectDisposedException"><para>The memory block is disposed.</para><para>Блок памяти уже высвобожден.</para></exception>
        /// <exception cref="ArgumentOutOfRangeException"><para>Attempted to set the used capacity to a value that is greater than the reserved capacity or less than zero.</para><para>Была выполнена попытка установить используемую емкость на значение, которое больше, чем зарезервированная емкость или меньше нуля.</para></exception>
        public long UsedCapacity
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                Ensure.Always.NotDisposed(this);
                return _usedCapacity;
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set
            {
                Ensure.Always.NotDisposed(this);
                if (value != _usedCapacity)
                {
                    Ensure.Always.ArgumentInRange(value, new Range<long>(0, _reservedCapacity));
                    _usedCapacity = value;
                }
            }
        }

        #endregion

        #region DisposableBase Properties

        /// <inheritdoc/>
        protected override bool AllowMultipleDisposeCalls
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => true;
        }

        #endregion

        #region Methods

        /// <summary>
        /// <para>Executed on the event of change for <see cref="ReservedCapacity"/>
         ↪  property.</para>
        /// <para>Выполняется в случае изменения свойства <see cref="ReservedCapacity"/>.</para>
        /// </summary>
        /// <param name="oldReservedCapacity"><para>The old reserved capacity of the memory
         ↪  block in bytes.</para><para>Старая зарезервированная емкость блока памяти в
         ↪  байтах.</para></param>
        /// <param name="newReservedCapacity"><para>The new reserved capacity of the memory
         ↪  block in bytes.</para><para>Новая зарезервированная емкость блока памяти в
         ↪  байтах.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void OnReservedCapacityChanged(long oldReservedCapacity, long
         ↪  newReservedCapacity);

        /// <summary>
        /// <para>Executed when it is time to dispose <see cref="Pointer"/>.</para>
        /// <para>Выполняется, когда пришло время высвободить <see cref="Pointer"/>.</para>
        /// </summary>
        /// <param name="pointer"><para>The pointer to a memory block.</para><para>Указатель на
         ↪  блок памяти.</para></param>
        /// <param name="usedCapacity"><para>The used capacity of the memory block in
         ↪  bytes.</para><para>Используемая емкость блока памяти в байтах.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void DisposePointer(IntPtr pointer, long usedCapacity);

        #endregion

        #region DisposableBase Methods

        /// <inheritdoc/>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                var pointer = Interlocked.Exchange(ref _pointer, IntPtr.Zero);
                if (pointer != IntPtr.Zero)
                {
                    DisposePointer(pointer, _usedCapacity);
                }
            }
        }

        #endregion
    }
}
```

1.11 ./Platform.Memory/TemporaryFileMappedResizableDirectMemory.cs

```
using System.IO;
using System.Runtime.CompilerServices;

namespace Platform.Memory
{
    /// <summary>
    /// <para>Represents a memory block stored as a temporary file on disk.</para>
    /// <para>Представляет блок памяти, хранящийся в виде временного файла на диске.</para>
    /// </summary>
    public class TemporaryFileMappedResizableDirectMemory : FileMappedResizableDirectMemory
    {
        #region DisposableBase Properties

        /// <inheritdoc/>
        protected override string ObjectName
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => $"Temporary file stored memory block at '{Path}' path.";
        }

        #endregion

        #region Constructors

        /// <summary>
        /// <para>Initializes a new instance of the <see
        ↪   cref="TemporaryFileMappedResizableDirectMemory"/> class.</para>
        /// <para>Инициализирует новый экземпляр класса <see
        ↪   cref="TemporaryFileMappedResizableDirectMemory"/>.</para>
        /// </summary>
        /// <param name="minimumReservedCapacity"><para>Minimum file size in
        ↪   bytes.</para><para>Минимальный размер файла в байтах.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TemporaryFileMappedResizableDirectMemory(long minimumReservedCapacity) :
        ↪   base(System.IO.Path.GetTempFileName(), minimumReservedCapacity) { }

        /// <summary>
        /// <para>Initializes a new instance of the <see
        ↪   cref="TemporaryFileMappedResizableDirectMemory"/> class.</para>
        /// <para>Инициализирует новый экземпляр класса <see
        ↪   cref="TemporaryFileMappedResizableDirectMemory"/>.</para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TemporaryFileMappedResizableDirectMemory() : this(MinimumCapacity) { }

        #endregion

        #region DisposableBase Methods

        /// <inheritdoc/>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void Dispose(bool manual, bool wasDisposed)
        {
            base.Dispose(manual, wasDisposed);
            if (!wasDisposed)
            {
                File.Delete(Path);
            }
        }

        #endregion
    }
}
```

1.12 ./Platform.Memory.Tests/HeapResizableDirectMemoryTests.cs

```
using Xunit;

namespace Platform.Memory.Tests
{
    public unsafe class HeapResizableDirectMemoryTests
    {
        [Fact]
        public void CorrectMemoryReallocationTest()
        {
            using var heapMemory = new HeapResizableDirectMemory();
            var value1 = GetLastByte(heapMemory);
            heapMemory.ReservedCapacity *= 2;
            var value2 = GetLastByte(heapMemory);
            Assert.Equal(value1, value2);
            Assert.Equal(0, value1);
        }

        private static byte GetLastByte(HeapResizableDirectMemory heapMemory)
        {
            var pointer1 = (void*)heapMemory.Pointer;
            return *((byte*)pointer1 + heapMemory.ReservedCapacity - 1);
        }
    }
}
```

Index