1.1 ./csharp/Platform.Memory/ArrayMemory.cs

```
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

        /// <summary>
        /// <para>
        /// The array.
        /// </para>
        /// <para></para>
        /// </summary>
        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/mem
        ↪  ber[@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 ./csharp/Platform.Memory/DirectMemoryAsArrayMemoryAdapter.cs

```
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>
```

```
13      /// <typeparam name="TElement"><para>Element type.</para><para>Тип
        ↪  элемента.</para></typeparam>
14      public class DirectMemoryAsArrayMemoryAdapter<TElement> : DisposableBase,
        ↪  IArrayMemory<TElement>, IDirectMemory
15          where TElement : struct
16      {
17          #region Fields
18
19          /// <summary>
20          /// <para>
21          /// The memory.
22          /// </para>
23          /// <para></para>
24          /// </summary>
25          private readonly IDirectMemory _memory;
26
27          #endregion
28
29          #region Properties
30
31          /// <inheritdoc/>
32          /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml'
            ↪  path='doc/members/member[@name="P:Platform.Memory.IMemory.Size"]/*'/>
33          public long Size
34          {
35              [MethodImpl(MethodImplOptions.AggressiveInlining)]
36              get => _memory.Size;
37          }
38
39          /// <inheritdoc/>
40          /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml'
            ↪  path='doc/members/member[@name="P:Platform.Memory.IDirectMemory.Pointer"]/*'/>
41          public IntPtr Pointer
42          {
43              [MethodImpl(MethodImplOptions.AggressiveInlining)]
44              get => _memory.Pointer;
45          }
46
47          /// <inheritdoc/>
48          /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml' path='doc/members/mem
            ↪  ber[@name="P:Platform.Memory.IArrayMemory`1.Item(System.Int64)"]/*'/>
49          public TElement this[long index]
50          {
51              [MethodImpl(MethodImplOptions.AggressiveInlining)]
52              get => Pointer.ReadElementValue<TElement>(index);
53              [MethodImpl(MethodImplOptions.AggressiveInlining)]
54              set => Pointer.WriteElementValue(index, value);
55          }
56
57          #endregion
58
59          #region DisposableBase Properties
60
61          /// <inheritdoc/>
62          protected override string ObjectName
63          {
64              [MethodImpl(MethodImplOptions.AggressiveInlining)]
65              get => $"Array as memory block at '{Pointer}' address.";
66          }
67
68          #endregion
69
70          #region Constructors
71
72          /// <summary>
73          /// <para>Initializes a new instance of the <see
            ↪  cref="DirectMemoryAsArrayMemoryAdapter{TElement}"/> class.</para>
74          /// <para>Инициализирует новый экземпляр класса <see
            ↪  cref="DirectMemoryAsArrayMemoryAdapter{TElement}"/>.</para>
75          /// </summary>
76          /// <param name="memory"><para>An object implementing <see cref="IDirectMemory"/>
            ↪  interface.</para><para>Объект, реализующий интерфейс <see
            ↪  cref="IDirectMemory"/>.</para></param>
77          [MethodImpl(MethodImplOptions.AggressiveInlining)]
78          public DirectMemoryAsArrayMemoryAdapter(IDirectMemory memory)
79          {
80              Ensure.Always.ArgumentNotNull(memory, nameof(memory));
81              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 ./csharp/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

        /// <summary>
        /// <para>
        /// The file.
        /// </para>
        /// <para></para>
        /// </summary>
        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/member[@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 ./csharp/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

        /// <summary>
        /// <para>
        /// The file.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        private MemoryMappedFile _file;
        /// <summary>
        /// <para>
        /// The accessor.
        /// </para>
        /// <para></para>
        /// </summary>
        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="path"><para>An path to file.</para><para>Путь к файлу.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public FileMappedResizableDirectMemory(string path) : this(path, MinimumCapacity) { }

        #endregion

        #region Methods

        /// <summary>
        /// <para>
        /// Maps the file using the specified capacity.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="capacity">
        /// <para>The capacity.</para>
```

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

        /// <summary>
        /// <para>
        /// Unmaps the file.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void UnmapFile()
        {
            if (UnmapFile(Pointer))
            {
                Pointer = IntPtr.Zero;
            }
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance unmap file.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="pointer">
        /// <para>The pointer.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [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 ./csharp/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/>
```

```
55          /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml'
            ↪  path='doc/members/member[@name="M:Platform.Memory.ResizableDirectMemoryBase.DisposeP
            ↪  ointer(System.IntPtr,System.Int64)"]/*'/>
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override void DisposePointer(IntPtr pointer, long usedCapacity) =>
            ↪  Marshal.FreeHGlobal(pointer);
58
59          /// <inheritdoc/>
60          /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml'
            ↪  path='doc/members/member[@name="M:Platform.Memory.ResizableDirectMemoryBase.OnReserv
            ↪  edCapacityChanged(System.Int64,System.Int64)"]/*'/>
61          [MethodImpl(MethodImplOptions.AggressiveInlining)]
62          protected override void OnReservedCapacityChanged(long oldReservedCapacity, long
            ↪  newReservedCapacity)
63          {
64              if (Pointer == IntPtr.Zero)
65              {
66                  Pointer = Marshal.AllocHGlobal(new IntPtr(newReservedCapacity));
67                  MemoryBlock.Zero((void*)Pointer, newReservedCapacity);
68              }
69              else
70              {
71                  Pointer = Marshal.ReAllocHGlobal(Pointer, new IntPtr(newReservedCapacity));
72                  var pointer = (byte*)Pointer + oldReservedCapacity;
73                  MemoryBlock.Zero(pointer, newReservedCapacity - oldReservedCapacity);
74              }
75          }
76
77          #endregion
78      }
79  }
```

1.6 ./csharp/Platform.Memory/IArrayMemory.cs

```
1   using System.Runtime.CompilerServices;
2
3   namespace Platform.Memory
4   {
5       /// <summary>
6       /// <para>Represents a memory block interface with access via indexer.</para>
7       /// <para>Представляет интерфейс блока памяти с доступом через индексатор.</para>
8       /// </summary>
9       /// <typeparam name="TElement"><para>Element type.</para><para>Тип
        ↪  элемента.</para></typeparam>
10      public interface IArrayMemory<TElement> : IMemory
11      {
12          /// <summary>
13          /// <para>Gets or sets the element at the specified index.</para>
14          /// <para>Возвращает или устанавливает элемент по указанному индексу.</para>
15          /// </summary>
16          /// <param name="index"><para>The index of the element to get or set.</para><para>Индекс
            ↪  элемента, который нужно получить или установить.</para></param>
17          TElement this[long index]
18          {
19              [MethodImpl(MethodImplOptions.AggressiveInlining)]
20              get;
21              [MethodImpl(MethodImplOptions.AggressiveInlining)]
22              set;
23          }
24      }
25  }
```

1.7 ./csharp/Platform.Memory/IDirectMemory.cs

```
1   using System;
2   using System.Runtime.CompilerServices;
3
4   namespace Platform.Memory
5   {
6       /// <summary>
7       /// <para>Represents a memory block interface with direct access (via unmanaged
        ↪  pointers).</para>
8       /// <para>Представляет интерфейс блока памяти с прямым доступом (через неуправляемые
        ↪  указатели).</para>
9       /// </summary>
10      public interface IDirectMemory : IMemory, IDisposable
11      {
12          /// <summary>
13          /// <para>Gets the pointer to the beginning of this memory block.</para>
14          /// <para>Возвращает указатель на начало блока памяти.</para>
```

```
        /// </summary>
        IntPtr Pointer
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }
    }
}
```

1.8 ./csharp/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 ./csharp/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 ./csharp/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 = Environment.SystemPageSize;

        #endregion

        #region Fields

        /// <summary>
        /// <para>
        /// The pointer.
        /// </para>
        /// <para></para>
        /// </summary>
        private IntPtr _pointer;
        /// <summary>
        /// <para>
        /// The reserved capacity.
        /// </para>
        /// <para></para>
        /// </summary>
        private long _reservedCapacity;
        /// <summary>
        /// <para>
        /// The used capacity.
        /// </para>
        /// <para></para>
        /// </summary>
        private long _usedCapacity;

        #endregion

        #region Properties

```

```
54          /// <inheritdoc/>
55          /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml'
            ↪  path='doc/members/member[@name="P:Platform.Memory.IMemory.Size"]/*'/>
56          /// <exception cref="ObjectDisposedException"><para>The memory block is
            ↪  disposed.</para><para>Блок памяти уже высвобожден.</para></exception>
57          public long Size
58          {
59              [MethodImpl(MethodImplOptions.AggressiveInlining)]
60              get
61              {
62                  Ensure.Always.NotDisposed(this);
63                  return UsedCapacity;
64              }
65          }
66
67          /// <inheritdoc/>
68          /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml'
            ↪  path='doc/members/member[@name="P:Platform.Memory.IDirectMemory.Pointer"]/*'/>
69          /// <exception cref="ObjectDisposedException"><para>The memory block is
            ↪  disposed.</para><para>Блок памяти уже высвобожден.</para></exception>
70          public IntPtr Pointer
71          {
72              [MethodImpl(MethodImplOptions.AggressiveInlining)]
73              get
74              {
75                  Ensure.Always.NotDisposed(this);
76                  return _pointer;
77              }
78              [MethodImpl(MethodImplOptions.AggressiveInlining)]
79              protected set
80              {
81                  Ensure.Always.NotDisposed(this);
82                  _pointer = value;
83              }
84          }
85
86          /// <inheritdoc/>
87          /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml' path='doc/members/mem
            ↪  ber[@name="P:Platform.Memory.IResizableDirectMemory.ReservedCapacity"]/*'/>
88          /// <exception cref="ObjectDisposedException"><para>The memory block is
            ↪  disposed.</para><para>Блок памяти уже высвобожден.</para></exception>
89          /// <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>
90          public long ReservedCapacity
91          {
92              [MethodImpl(MethodImplOptions.AggressiveInlining)]
93              get
94              {
95                  Ensure.Always.NotDisposed(this);
96                  return _reservedCapacity;
97              }
98              [MethodImpl(MethodImplOptions.AggressiveInlining)]
99              set
100             {
101                 Ensure.Always.NotDisposed(this);
102                 if (value != _reservedCapacity)
103                 {
104                     Ensure.Always.ArgumentInRange(value, new Range<long>(_usedCapacity,
                        ↪  long.MaxValue));
105                     OnReservedCapacityChanged(_reservedCapacity, value);
106                     _reservedCapacity = value;
107                 }
108             }
109         }
110
111         /// <inheritdoc/>
112         /// <include file='bin\Release\netstandard2.0\Platform.Memory.xml' path='doc/members/mem
            ↪  ber[@name="P:Platform.Memory.IResizableDirectMemory.UsedCapacity"]/*'/>
113         /// <exception cref="ObjectDisposedException"><para>The memory block is
            ↪  disposed.</para><para>Блок памяти уже высвобожден.</para></exception>
114         /// <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>
115         public long UsedCapacity
```

```
116         {
117             [MethodImpl(MethodImplOptions.AggressiveInlining)]
118             get
119             {
120                 Ensure.Always.NotDisposed(this);
121                 return _usedCapacity;
122             }
123             [MethodImpl(MethodImplOptions.AggressiveInlining)]
124             set
125             {
126                 Ensure.Always.NotDisposed(this);
127                 if (value != _usedCapacity)
128                 {
129                     Ensure.Always.ArgumentInRange(value, new Range<long>(0, _reservedCapacity));
130                     _usedCapacity = value;
131                 }
132             }
133         }
134 
135         #endregion
136 
137         #region DisposableBase Properties
138 
139         /// <inheritdoc/>
140         protected override bool AllowMultipleDisposeCalls
141         {
142             [MethodImpl(MethodImplOptions.AggressiveInlining)]
143             get => true;
144         }
145 
146         #endregion
147 
148         #region Methods
149 
150         /// <summary>
151         /// <para>Executed on the event of change for <see cref="ReservedCapacity"/>
            ↪ property.</para>
152         /// <para>Выполняется в случае изменения свойства <see cref="ReservedCapacity"/>.</para>
153         /// </summary>
154         /// <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>
155         /// <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>
156         [MethodImpl(MethodImplOptions.AggressiveInlining)]
157         protected abstract void OnReservedCapacityChanged(long oldReservedCapacity, long
            ↪ newReservedCapacity);
158 
159         /// <summary>
160         /// <para>Executed when it is time to dispose <see cref="Pointer"/>.</para>
161         /// <para>Выполняется, когда пришло время высвободить <see cref="Pointer"/>.</para>
162         /// </summary>
163         /// <param name="pointer"><para>The pointer to a memory block.</para><para>Указатель на
            ↪ блок памяти.</para></param>
164         /// <param name="usedCapacity"><para>The used capacity of the memory block in
            ↪ bytes.</para><para>Используемая емкость блока памяти в байтах.</para></param>
165         [MethodImpl(MethodImplOptions.AggressiveInlining)]
166         protected abstract void DisposePointer(IntPtr pointer, long usedCapacity);
167 
168         #endregion
169 
170         #region DisposableBase Methods
171 
172         /// <inheritdoc/>
173         [MethodImpl(MethodImplOptions.AggressiveInlining)]
174         protected override void Dispose(bool manual, bool wasDisposed)
175         {
176             if (!wasDisposed)
177             {
178                 var pointer = Interlocked.Exchange(ref _pointer, IntPtr.Zero);
179                 if (pointer != IntPtr.Zero)
180                 {
181                     DisposePointer(pointer, _usedCapacity);
182                 }
183             }
184         }
185 
186         #endregion
```

```
    }
}
```

1.11 ./csharp/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 ./csharp/Platform.Memory.Tests/HeapResizableDirectMemoryTests.cs

```
using Xunit;

namespace Platform.Memory.Tests
{
    /// <summary>
    /// <para>
    /// Represents the heap resizable direct memory tests.
    /// </para>
    /// <para></para>
    /// </summary>
```

```
11      public unsafe class HeapResizableDirectMemoryTests
12      {
13          /// <summary>
14          /// <para>
15          /// Tests that correct memory reallocation test.
16          /// </para>
17          /// <para></para>
18          /// </summary>
19          [Fact]
20          public void CorrectMemoryReallocationTest()
21          {
22              using var heapMemory = new HeapResizableDirectMemory();
23              var value1 = GetLastByte(heapMemory);
24              heapMemory.ReservedCapacity *= 2;
25              var value2 = GetLastByte(heapMemory);
26              Assert.Equal(value1, value2);
27              Assert.Equal(0, value1);
28          }
29
30          /// <summary>
31          /// <para>
32          /// Gets the last byte using the specified heap memory.
33          /// </para>
34          /// <para></para>
35          /// </summary>
36          /// <param name="heapMemory">
37          /// <para>The heap memory.</para>
38          /// <para></para>
39          /// </param>
40          /// <returns>
41          /// <para>The byte</para>
42          /// <para></para>
43          /// </returns>
44          private static byte GetLastByte(HeapResizableDirectMemory heapMemory)
45          {
46              var pointer1 = (void*)heapMemory.Pointer;
47              return *((byte*)pointer1 + heapMemory.ReservedCapacity - 1);
48          }
49      }
50  }
```

Index