Memory<T> API documentation and samples

memory_docs_samples.md

Memory<T> API documentation and samples

This document describes the APIs of `Memory<T>`, `IMemoryOwner<T>`, and `MemoryManager<T>` and their relationships to each other.

See also the `Memory<T>` usage guidelines document for background information.

First, a brief summary of the basic types

- `Memory<T>` is the basic type that represents a contiguous buffer. This type is a *struct*, which means that developers cannot subclass it and override the implementation. The basic implementation of the type is aware of contigious memory buffers backed by `T[]` and `System.String` (in the case of `ReadOnlyMemory<char>`).
- `IMemoryOwner<T>` is the interface that controls the *lifetime* of `Memory<T>` instances. The interface implementation how to *create* and *destroy* these instances. Instances of this interface are returned by `MemoryPool<T>.Rent`.
- `MemoryPool<T>` is the abstract base class that represents a memory pool, where `IMemoryOwner<T>` instances can be rented from the pool (via `MemoryPool<T>.Rent`) and released back to the pool (via `IMemoryOwner<T>.Dispose`).
- `MemoryManager<T>` is the abstract base class used to *replace the implementation of `Memory<T>`*. This class is used to extend the knowledge of types that `Memory<T>` knows about; e.g., to allow `Memory<T>` to be backed by a `SafeHandle`. **This type is intended for advanced scenarios; most developers will not need to use it.**

Some simple examples using built-in types

Say you have a method that needs to use a temporary buffer, but you don't want to incur the cost of an allocation. One way to write this code is to use the existing `ArrayPool<T>` type.

```
// workhorse method
static void DoSomething(Memory<byte> scratchBuffer);

public void UserCode()
{
    byte[] rentedArray = ArrayPool<byte>.Shared.Rent(...);
    try {
        DoSomething(rentedArray); // T[] implicitly converted to Memory<T>
    } finally {
        if (rentedArray != null) {
            ArrayPool<byte>.Shared.Return(rentedArray);
        }
    }
}
```

The above method is almost always allocation-free. When the `DoSomething` method is finished with the buffer, the buffer is returned back to the shared `ArrayPool<byte>`.

This method can be replaced with the following simpler code, using `MemoryPool<T>` instead of `ArrayPool<T>`.

```
// workhorse method
static void DoSomething(Memory<byte> scratchBuffer);

public void UserCode()
{
    using (var rental = MemoryPool<byte>.Shared.Rent(...))
    {
        DoSomething(rental.Memory);
    }
}
```

In fact, you could even abstract away the particular `MemoryPool<T>` used by this method, allowing the application to use a custom memory pool that is backed by native memory or some other custom implementation.

```
// workhorse method
static void DoSomething(Memory<byte> scratchBuffer);

public void UserClass : IDisposable
{
    private readonly MemoryPool<T> _pool;

    public UserClass(MemoryPool<T> pool)
    {
        _pool = pool;
    }

    public void UserCode()
    {
        using (var rental = pool.Rent(...))
        {
            DoSomething(rental.Memory);
        }
    }

    void IDisposable.Dispose()
    {
        _pool.Dispose();
    }
}
```

Creating custom pools

Say that as a security measure, you want to implement a custom `MemoryPool<T>` that has the following semantics.

1. You use a different pool than the one used by `ArrayPool<T>.Shared` to limit secret exposure to the rest of the application.
2. Any buffers returned to the pool should be cleared before they're next rented out.

The implementation of this custom memory pool might look as follows.

```
public class CustomMemoryPool<T> : MemoryPool<T>
{
    private readonly ArrayPool<T> _arrayPool = ArrayPool<T>.Create();

    public override int MaxBufferSize => Int32.MaxValue;

    protected override void Dispose(bool disposing)
```

```
    {
        if (disposing)
        {
            _arrayPool.Dispose();
        }
    }

    public override IMemoryOwner<T> Rent(int minBufferSize)
    {
        return new CustomMemoryOwner(_arrayPool, minBufferSize);
    }

    private sealed class CustomMemoryOwner : IMemoryOwner<T>
    {
        private readonly ArrayPool<T> _pool;
        private T[] _array;

        public CustomMemoryOwner(ArrayPool<T> pool, int minBufferSize)
        {
            _pool = pool;
            _array = pool.Rent(minBufferSize);
        }

        public Memory<T> Memory
        {
            get
            {
                if (_array == null)
                    throw new ObjectDisposedException(...);

                return new Memory<T>(_array);
            }
        }

        public void Dispose()
        {
            if (_array != null)
            {
                ((Span<T>)_array).Clear();
                _pool.Return(_array);
                _array = null;
            }
        }
    }
}

// workhorse method
static void DoSomething(Memory<byte> scratchBuffer);
```

```
static readonly MemoryPool<byte> _customPool = new CustomMemoryPool<byte>();

public void UserCode()
{
    // use custom pool instead of default pool
    using (var rental = _customPool.Rent(...))
    {
        DoSomething(rental.Memory);
    }
}
```

As a second example, consider the behavior of the existing `MemoryPool<T>.Shared.Rent` method. This method necessarily always allocates when called. Even though it's not allocating the full buffer during each call to `Rent`, it *is* allocating a new lightweight `IMemoryOwner<T>` wrapper on each call.

Let's say we want to make a custom memory pool which is equivalent to the built-in shared pool but which *doesn't* allocate on each call to `Rent`. An implementation of such a pool might look as follows.

```
public abstract class NonAllocatingPool<T> : MemoryPool<T>
{
    public override int MaxBufferSize => Int32.MaxValue;

    public static new NonAllocatingPool<T>.Impl Shared { get; } = new NonAllocatingP

    protected override void Dispose(bool disposing) { }

    public override IMemoryOwner<T> Rent(int minBufferSize) => RentCore(minBufferSiz

    private Rental RentCore(int minBufferSize) => new MemoryOwner(minBufferSize);

    public sealed class Impl : NonAllocatingPool<T>
    {
        // Typed to return the actual type rather than the
        // interface to avoid boxing, like how List<T>.GetEnumerator()
        // returns List<T>.Enumerator instead of IEnumerator<T>.
        public new Rental Rent(int minBufferSize) => RentCore(minBufferSize);
    }

    // Struct implements the interface so it can be boxed if necessary.
    public struct Rental : IMemoryOwner<T>
    {
        private T[] _array;

        public Rental(int minBufferSize)
        {
```

```
            _array = ArrayPool<T>.Shared.Rent(minBufferSize);
        }

        public Memory<T> Memory
        {
            get
            {
                if (_array == null)
                    throw new ObjectDisposedException(...);

                return new Memory<T>(_array);
            }
        }

        public void Dispose()
        {
            if (_array != null)
            {
                ArrayPool<T>.Shared.Return(_array);
                _array = null;
            }
        }
    }
}

// workhorse method
static void DoSomething(Memory<byte> scratchBuffer);

public void UserCode()
{
    // Use non-allocating pool instead of default pool.
    // n.b. it's best to use a 'using' block to avoid inadvertently
    // making copies of the rental struct and exposing the application
    // to double-releasing the rental.
    using (var rental = NonAllocatingPool<byte>.Shared.Rent(...))
    {
        DoSomething(rental.Memory);
    }
}
```

As shown in these examples, creating a custom pool involves subclassing the `MemoryPool<T>` type and implementing the `IMemoryOwner<T>` interface. In particular, these scenarios **do not** involve subclassing the `MemoryManager<T>` type.

Replacing the `Memory<T>` implementation

Rarely, the developer may need to replace the entire implementation of `Memory<T>` in order to get it to recognize new categories of contiguous buffers. One example would be the scenario where you want to back `Memory<T>` by an existing `SafeHandle`, which is not natively supported out-of-box.

To enable these scenarios, subclass the `MemoryManager<T>` type. This allows replacing the implementation of `Memory<T>`'s `Span` property getter and its `Pin` method. **These are advanced scenarios and should only be performed by developers comfortable with manual memory management techniques.**

Your `SafeHandle`-backed `MemoryManager<T>` type might look like the below.

```
public unsafe class SafeHandleMemoryManager<T> : MemoryManager<T>
{
    private int _disposeCalled = 0;
    private readonly int _elementCount;
    private readonly SafeHandle _handle;

    public SafeHandleMemoryManager(SafeHandle handle, int elementCount)
    {
        // Important: Add a reference to the SafeHandle so that it's not
        // released without a corresponding call to our Dispose method.
        // It's better for us to leak (due to the owner forgetting to
        // call Dispose) than it is for the GC to reclaim the handle while
        // it might still be in use, potentially leading to heap corruption.

        bool unused = false;
        handle.DangerousAddRef(ref unused);

        _handle = handle;
        _elementCount = elementCount;
    }

    public override int Length => _elementCount;

    public override bool Dispose(bool disposing)
    {
        if (disposing && (Interlocked.Exchange(ref _disposeCalled, 1) == 0))
        {
            _handle.DangerousRelease(); // undo DangerousAddRef from ctor
            _handle.Dispose();
        }
    }
```

```
    public override Span<T> GetSpan()
    {
        // n.b. The check below *cannot* detect all use-after-free cases
        // where an incorrectly written consumer attempts to utilize this
        // Memory<T> instance after its lease has ended. The safety of a
        // Memory<T> instance backed by an unmanaged pointer is ultimately
        // dependent on the owner and the consumers following the provided
        // guidance correctly.

        if (_handle.IsClosed)
            throw new ObjectDisposedException(...);

        return new Span<T>((void*)_handle.DangerousGetHandle(), _elementCount);
    }

    public override MemoryHandle Pin(int elementIndex)
    {
        // Check for negative elementIndex and elementIndex greater
        // than our element count. By convention we allow an element
        // index equal to the element count, which returns a pointer
        // to the end of the buffer but which the caller should not
        // attempt to read from or write to.

        if ((uint)elementIndex > (uint)_elementCount)
            throw new ArgumentException("Invalid index.");

        bool unused = false;
        _handle.DangerousAddRef(ref unused); // will be released in Unpin

        // n.b. The Unsafe.Add method below does the correct computation
        // automatically based on the bitness of the current process. If
        // you need to perform the calculation manually by mutiplying
        // sizeof(T) by the element index, be sure to extend the arguments
        // to 64 bits *before* the multiplication if you're running in a
        // 64-bit process, otherwise you could be subject to integer overflow.

        byte* pbData = Unsafe.Add<T>((void*)_handle.DangerousGetHandle(), elementInd
        return new MemoryHandle(pbData, owner: this);
    }

    public override void Unpin()
    {
        _handle.DangerousRelease();
    }
}

// factory method
static SafeHandle HeapAlloc(int cb);
```

```
// workhorse method
static void DoSomething(Memory<byte> scratchBuffer);

public void UserCode()
{
    using (var manager = new SafeHandleMemoryManager<byte>(HeapAlloc(512), 512))
    {
        // n.b. the SafeHandle will be disposed at the end of this
        // 'using' block.
        DoSomething(manager.Memory);
    }
}
```

Implementation notes when subclassing `MemoryManager<T>`

Due to the inherent danger in manual memory management, below are some tips to keep in mind if you have the need to subclass `MemoryManager<T>`.

Never give your `MemoryManger<T>`-derived type a finalizer. If an incorrectly-written component loses its reference to the `IMemoryOwner<T>` and forgets to call `Dispose`, the garbage collector could kick in unexpectedly, *even while a component still has a reference to the `Span<T>` backed by the custom `MemoryManager<T>`*. In this scenario it is better to leak than to risk heap corruption. (Exception: it is acceptable to create a finalizer for debugging or validation purposes as long as the finalizer no-ops.)

The `GetSpan` method corresponds to the `Memory<T>.Span` property getter. Use a constructor on `Span<T>` if your buffer is backed by a `T[]`, or use `MemoryMarshal.CreateSpan` if you need to fall back to more complex, unsafe creation semantics.

Finally, consider the relationship between the `Pin` and `Unpin` methods. A correctly written `Pin` method needs to take into account the following.

- If the underlying buffer is not already pinned, call `GCHandle.Alloc` to pin the buffer. n.b. the call to `GCHandle.Alloc` may throw, and the `Pin` method should be prepared to handle these failures.
- The parameter passed to `Pin` is the *element index* of the requested pointer. When performing any byte offset calculations, the implementation needs to consider the bitness of the current platform so that calculations do not result in integer overflow or underflow. If your underlying buffer is a `T[]`, consider using the `Marshal.UnsafeAddrOfPinnedArrayElement` method to calculate the correct pointer.

- The `Pin` method may cache a single `GCHandle` for the underlying buffer, or it may return a unique `GCHandle` per call. If the `GCHandle` is per-call, then this handle should be given to the `MemoryHandle` constructor, and `MemoryHandle.Dispose` will automatically free the `GCHandle` at the correct time. If the `GCHandle` is cached per instance, then `default(GCHandle)` should be passed to the `MemoryHandle` constructor, and your `MemoryManager<T>` subclass's `Dispose` method should free the `GCHandle`.

- Pass `this` for the *owner* parameter in the `MemoryHandle` constructor. This will cause `MemoryHandle.Dispose` to call your `MemoryManager<T>` subclass's `Unpin` method at the appropriate time.

- Despite the name, the `Unpin` method does not mean "unpin your `GCHandle`." Rather, it's simply a callback to indicate that the `MemoryHandle` returned by your `Pin` method is being destroyed. You generally won't need to put anything into your `Unpin` method implementation aside from ref counting (if required).

Your `MemoryManager<T>` subclass *may* (but is *not required to*) perform ref counting. If your subclass performs ref counting, this should be for diagnostics or validation, not for correctness. Per the usage guidance, the buffers represented by `Memory<T>` instances are not inherently thread-safe without external synchronization, so ref counting for correctness does not necessarily help guard against data corruption.

ektrah commented on Mar 30, 2018

A little bit of feedback. Is there a way to make a pull request for a gist?

```
 public abstract class NonAllocatingPool<T> : MemoryPool<T>
 {
     ...

-    private Impl RentCore(int minBufferSize) => new MemoryOwner(minBufferSize);
+    private MemoryOwner RentCore(int minBufferSize) => new MemoryOwner(minBufferSize);

     public sealed class Impl : NonAllocatingPool<T>
     {
         // Typed to return the actual type rather than the
         // interface to avoid boxing, like how List<T>.GetEnumerator()
         // returns List<T>.Enumerator instead of IEnumerator<T>.
-        public new Impl Rent(int minBufferSize) => RentCore(minBufferSize);
+        public new MemoryOwner Rent(int minBufferSize) => RentCore(minBufferSize);
     }
```

```
public abstract class NonAllocatingPool<T> : MemoryPool<T>
{
    public struct MemoryOwner : IMemoryOwner<T>
    {
        private T[] _rental;

        ...

        public void Dispose()
        {
            if (_rental != null)
            {
                ArrayPool<T>.Shared.Return(_rental);
                _rental = null;
            }
        }
    }
}
```

Mutable structs are evil. This already bothered me a lot with GCHandle. Double-freeing is only prevented if the user carefully avoids the defining feature of structs. Solution: Add a language feature that prevents a struct value from being duplicated or don't create mutable structs.

```
 public unsafe class SafeHandleMemoryManager<T> : MemoryManager<T>
 {
     public override Span<T> GetSpan()
     {
-        bool success = false;
-        try
-        {
-            _handle.DangerousAddRef(ref success);
-            return MemoryMarshal.CreateSpan(ref Unsafe.AsRef<T>((void*)_handle.DangerousGetHandle())
-        }
-        finally
-        {
-            if (success)
-            {
-                _handle.DangerousRelease();
-            }
-        }
+        return new Span<T>((void*)_handle.DangerousGetHandle(), _elementCount);
     }
 }
```

- Afaik MemoryMarshal.CreateSpan isn't available outside netcoreapp2.1.

- Not sure why DangerousAddRef/Release is needed here. The reference count is already > 0 because of the constructor and it seems pointless to increase the reference count, get the pointer, decrease the reference count, and then return and use the pointer.

```
public unsafe class SafeHandleMemoryManager<T> : MemoryManager<T>
{
    public override MemoryHandle Pin(int elementIndex)
    {
        // Check for negative elementIndex and elementIndex greater
        // than our element count. By convention we allow an element
        // index equal to the element count, which returns a pointer
        // to the end of the buffer but which the caller should not
        // attempt to read from or write to.

        if ((uint)elementIndex > (uint)_elementCount)
            throw new ArgumentException("Invalid index.");

        bool unused = false;
-       _handle.DangerousAddRef(ref success); // will be released in Unpin
+       _handle.DangerousAddRef(ref unused); // will be released in Unpin

-       byte* pbData = (byte*)_handle.DangerousGetHandle();
-
-       if (IntPtr.Size == 4)
-       {
-           // 32-bit process
-           pbData += elementIndex * Unsafe.SizeOf<T>();
-       }
-       else
-       {
-           // 64-bit process
-           // n.b. extension to 64-bit to avoid integer overflow.
-           pbData += elementIndex * (long)Unsafe.SizeOf<T>();
-       }
+       void* pbData = Unsafe.Add<T>((void*)_handle.DangerousGetHandle(), elementIndex);

        return new MemoryHandle(pbData, owner: this);
    }

    public override void Unpin()
    {
        _handle.DangerousRelease();
    }
}
```

GrabYourPitchforks commented on Mar 31, 2018 Author

@ektrah Thanks for the feedback! This is what I get for writing this in notepad. :)

I agree that mutable structs are not ideal, but they are useful in certain cases as performance optimizations if the developer is disciplined in using them. For instance, `MemoryHandle` (returned by `Memory<T>.Pin`) is a mutable struct. Personally I'd love to see C# get a concept similar to C++'s move-only types, but it's undoubtedly a niche scenario.

antonfirsov commented on May 27, 2018 • edited

@GrabYourPitchforks any plans to update the current "official" docs with this? (+ the guideline)

The API is stable now AFAIK, the old docs are pretty much outdated, but your guidelines rock!

buyaa-n commented on May 15, 2020

Great docs and samples, very helpful thank you **@GrabYourPitchforks**