Memory usage guidelines

memory_guidelines.md

Memory<T> usage guidelines

This document describes the relationship between `Memory<T>` and its related classes (`MemoryPool<T>` , `IMemoryOwner<T>` , etc.). It also describes best practices when accepting `Memory<T>` instances in public API surface. Following these guidelines will help developers write clear, bug-free code.

First, a tour of the basic exchange types

- `Span<T>` is the basic exchange type that represents contiguous buffers. These buffers may be backed by managed memory (such as `T[]` or `System.String`). They may also be backed by unmanaged memory (such as via `stackalloc` or a raw `void*`). The `Span<T>` type is not heapable, meaning that it cannot appear as a field in classes, and it cannot be used across `yield` or `await` boundaries.

- `Memory<T>` is a wrapper around an object that can generate a `Span<T>` . For instance, `Memory<T>` instances can be backed by `T[]` , `System.String` (readonly), and even `SafeHandle` instances. `Memory<T>` cannot be backed by "transient" unmanaged memory; e.g., it is forbidden to back a `Memory<T>` with `stackalloc` . The `Memory<T>` type is heapable, meaning that it can appear as a field in a class, and it can be used across `yield` and `await` boundaries.

There are also `ReadOnlySpan<T>` and `ReadOnlyMemory<T>` types that correspond to read-only versions of `Span<T>` and `Memory<T>` , respectively.

Owners, consumers, and lifetime management

Let's stick a pin in `Memory<T>` for now and speak about buffers in more general terms. Since buffers can be passed around between APIs, and since buffers can sometimes be accessed from multiple threads, we need to introduce lifetime semantics. There are three core concepts.

The first concept is **ownership**. The *owner* of a buffer instance is responsible for lifetime management, including *destroying* the buffer when it is no longer in use. All buffers have a *single owner*. Generally the owner is the component which created the buffer or which received the buffer from a factory. Ownership can also be transferred; Component A can relinquish control of the buffer to Component B, at which point Component A may no longer use the buffer, and Component B becomes responsible for destroying the buffer when it is no longer in use.

The second concept is **consumption**. The *consumer* of a buffer instance is allowed to *use* the buffer instance, perhaps writing to or reading from it. Buffers have *one consumer at a time* unless some external synchronization mechanism is provided.

Importantly, *the active consumer of a buffer is not necessarily the buffer's owner*. Consider the following pseudocode, where the `Buffer` type is a stand-in for an arbitrary buffer type.

```
// Writes 'value' as a human-readable string to the output buffer.
void WriteInt32ToBuffer(int value, Buffer buffer);

// Prints the contents of the buffer to the console.
void PrintBufferToConsole(Buffer buffer);

// Application code
void Main()
{
    var buffer = CreateBuffer();
    try {
        int value = Int32.Parse(Console.ReadLine());
        WriteInt32ToBuffer(value, buffer);
        PrintBufferToConsole(buffer);
    } finally {
        buffer.Destroy();
    }
}
```

In this pseudocode, the `Main` method creates the buffer so becomes its *owner*, and `Main` is thus responsible for destroying the buffer when it's no longer in use. The buffer only ever has one *consumer* at a time (first `WriteInt32ToBuffer`, then `PrintBufferToConsole`), and neither of the consumers owns the buffer. Note also that "consumer" in this context does not imply a read-only view of the buffer; consumers can modify buffer contents if given a read+write view of the buffer.

The third concept is that of a **lease**. The lease is the window of time that any given component is allowed to be the consumer of the buffer. In the example above, the `WriteInt32ToBuffer` method has a lease on (can consume) the buffer between the start of the method call and the time the method returns. Similarly, `PrintBufferToConsole` has a lease on the buffer while it is executing, and the lease is released when the method unwinds. (There is no API for lease management; a "lease" is simply a conceptual matter.)

Memory<T> and the owner / consumer model

At this point, let's reintroduce `Memory<T>` into the picture, along with one more type: `IMemoryOwner<T>`.

The type `IMemoryOwner<T>` is, as its name suggests, the **unit of ownership** of the associated `Memory<T>` instance. If a component has an `IMemoryOwner<T>` reference, then that component *owns* the buffer.

`Memory<T>` is itself the **unit of consumption**. If a component has a `Memory<T>` reference, then that component *consumes* the buffer.

To clarify this point, consider once again the earlier pseudocode, but let's now introduce real types into the system.

```
// Writes 'value' as a human-readable string to the output buffer.
void WriteInt32ToBuffer(int value, Memory<char> buffer);

// Prints the contents of the buffer to the console.
void PrintBufferToConsole(Memory<char> buffer);

// Application code
void Main()
{
    IMemoryOwner<char> owner = MemoryPool<char>.Shared.Rent();
    try {
        int value = Int32.Parse(Console.ReadLine());
        WriteInt32ToBuffer(value, owner.Memory);
        PrintBufferToConsole(owner.Memory);
    } finally {
```

```
        owner.Dispose();
    }

    // Alternatively, with 'using' syntax instead of 'try / finally'

    using (var owner = MemoryPool<char>.Shared.Rent())
    {
        int value = Int32.Parse(Console.ReadLine());
        WriteInt32ToBuffer(value, owner.Memory);
        PrintBufferToConsole(owner.Memory);
    }
}
```

Again, in this code, the `Main` method holds the reference to the `IMemoryOwner<char>` instance, so the `Main` method is the *owner* of the buffer. The `WriteInt32ToBuffer` and `PrintBufferToConsole` methods accept `Memory<T>` as a public API, therefore they *consume* the buffer. (And they only consume it one-at-a-time.)

(The observant reader may note that `PrintBufferToConsole` should really accept `ReadOnlyMemory<char>` instead of `Memory<char>` as a method argument. More on this later.)

A quick note on "ownerless" `Memory<T>` instances

It is certainly possible to create a `Memory<T>` without going through `IMemoryOwner<T>`. One way to do this would be to call one of the `Memory<T>` constructors directly, passing in a `T[]`. Or code could call the `String.AsMemory` extension method to produce a `ReadOnlyMemory<char>`.

```
// Writes 'value' as a human-readable string to the output buffer.
void WriteInt32ToBuffer(int value, Memory<char> buffer);

// Prints the contents of the buffer to the console.
void PrintBufferToConsole(Memory<char> buffer);

// Application code
void Main()
{
    Memory<char> memory = new char[64];

    int value = Int32.Parse(Console.ReadLine());
    WriteInt32ToBuffer(value, memory);
    PrintBufferToConsole(memory);
}
```

In this case, the method which initially creates the `Memory<T>` instance is the implicit *owner* of the buffer. Ownership cannot be transferred to any other component because there is no `IMemoryOwner<T>` to facilitate the transfer. (As an alternative, you can also imagine that the runtime's garbage collector *owns* the buffer, and all methods shown here just *consume* of the buffer.)

Usage guidelines

Now that we have the basics down, we can go over the rules necessary for successful usage of `Memory<T>` and related types.

In the rules below, we'll generally refer just to `Memory<T>` and `Span<T>` . The same guidance also applies to `ReadOnlyMemory<T>` and `ReadOnlySpan<T>` unless explicitly called out otherwise.

Rule #1: If writing a synchronous API, accept `Span<T>` instead of `Memory<T>` as a parameter if possible.

`Span<T>` is more versatile than `Memory<T>` and can represent a wider variety of contigious memory buffers. `Span<T>` also has better performance characteristics than `Memory<T>` . Finally, `Memory<T>` is convertible to `Span<T>` , but there is no `Span<T>` -to- `Memory<T>` conversion possible. So if your callers happen to have `Memory<T>` instance, they'll be able to call your `Span<T>` -accepting method anyway.

Accepting `Span<T>` instead of `Memory<T>` also helps you write a correct consuming method implementation, as you'll automatically get compile-time checks to ensure that you're not attempting to access the buffer beyond your method's lease (more on this later).

Sometimes circumstances will necessitate you taking a `Memory<T>` parameter instead of a `Span<T>` parameter, even if you're fully synchronous. Perhaps an API that you depend on has only `Memory<T>` -based overloads, and you need to flow your input parameter down to that method. This is fine, but be aware of the tradeoffs mentioned in the first paragraph in this rule.

Rule #2: Use `ReadOnlySpan<T>` or `ReadOnlyMemory<T>` if the buffer is intended to be immutable.

Consider the `PrintBufferToConsole` method from the earlier sample code.

```
void PrintBufferToConsole(Memory<char> buffer);
```

This method only reads from the buffer; it does not modify the contents of the buffer. The method signature should be changed to the following.

```
void PrintBufferToConsole(ReadOnlyMemory<char> buffer);
```

In fact, combining this rule and Rule #1 above, we can do even better and rewrite it as follows.

```
void PrintBufferToConsole(ReadOnlySpan<char> buffer);
```

The `PrintBufferToConsole` method now works with pretty much every buffer type imagineable: `T[]`, `stackalloc`, and so on. You can even pass a `System.String` directly into it!

Rule #3: If your method accepts `Memory<T>` and returns `void`, you must not use the `Memory<T>` instance after your method returns.

This relates back to the "lease" concept mentioned earlier. A void-returning method's lease on the `Memory<T>` begins when the method is entered, and it ends when the method exits.

Consider the following code sample, which calls `Log` in a loop based on input from the console.

```
// implementation provided by third party
static void Log(ReadOnlyMemory<char> message);

// user code
public void Main()
{
    using (var owner = MemoryPool<char>.Shared.Rent())
    {
        var memory = owner.Memory;
        var span = memory.Span;
        while (true)
        {
            int value = Int.Parse(Console.ReadLine());
            if (value < 0) { return; }

            int numCharsWritten = value.ToBuffer(span);
            Log(memory.Slice(0, numCharsWritten));
        }
```

```
    }
}
```

If `Log` is a fully synchronous method, this code will behave as expected, as there will be only one active consumer of the memory instance at any given time.

Now, imagine instead that `Log` has this implementation.

```
// !!! INCORRECT IMPLEMENTATION !!!
static void Log(ReadOnlyMemory<char> message)
{
    // Run in background so that we don't block the main thread
    // while performing IO.
    Task.Run(() => {
        File.AppendText(message);
    });
}
```

In this scenario, `Log` violates its lease because it's still attempting to use the `Memory<T>` instance in the background *after the original method has returned*. The `Main` method could be mutating the buffer while `Log` is attempting to read from it, which could result in data corruption.

There are a few ways to resolve this. One way could be for the `Log` method to return a `Task` instead of returning `void`. Another way could be for `Log` to instead be implemented as follows.

```
// Acceptable implementation
static void Log(ReadOnlySpan<char> message)
{
    string defensiveCopy = message.ToString();
    // Run in background so that we don't block the main thread
    // while performing IO.
    Task.Run(() => {
        File.AppendText(defensiveCopy);
    });
}
```

Rule #4: If your method accepts `Memory<T>` and returns `Task`, you must not use the `Memory<T>` instance after the `Task` transitions to a terminal state.

This is just the *async* variant of Rule #3. The `Log` method from the earlier example can be written as follows to be compliant with this rule.

```
// Acceptable implementation
static Task LogAsync(ReadOnlyMemory<char> message)
{
    return Task.Run(() => {
        File.AppendText(message);
    });
}
```

Here, "terminal state" means that the `Task` transitions to a successful, faulted, or canceled state. In other words, "terminal state" means "anything that would cause `await` to throw or to continue execution."

This guidance holds for methods which return `Task`, `Task<T>`, `ValueTask<T>`, or any similar type.

Rule #5: If your constructor accepts `Memory<T>` as a parameter, instance methods on the constructed object are assumed to be consumers of the `Memory<T>` instance.

Consider the following sample code.

```
class OddValueExtractor {
    public OddValueExtractor(ReadOnlyMemory<int> input);
    public bool TryReadNextOddValue(out int value);
}

void PrintAllOddValues(ReadOnlyMemory<int> input)
{
    var extractor = new OddValueExtractor(input);
    while (extractor.TryReadNextOddValue(out int value))
    {
      Console.WriteLine(value);
    }
}
```

Here, the `OddValueExtractor` constructor accepts a `Memory<T>` as a constructor parameter, so the constructor itself is a *consumer* of the `Memory<T>` instance, and all instance methods on the returned value are also consumers of the original `Memory<T>` instance.

This means that `TryReadNextOddValue` *consumes* the `Memory<T>` instance, even though the instance isn't passed directly to the `TryReadNextOddValue` method.

Rule #6: If you have a settable `Memory<T>` -typed property (or equivalent instance method) on your type, instance methods on that object are assumed to be consumers of the `Memory<T>` instance.

This is really just a variant of Rule #5. This rule exists because property setters or equivalent methods are assumed to capture and persist their inputs, so instance methods on the same object may utilize the captured state.

A sample class which triggers this rule is provided below.

```
class Person
{
    // settable property
    public Memory<char> FirstName { get; set; }

    // alternatively, equivalent "setter" method
    public SetFirstName(Memory<char> value);

    // alternatively, a public settable field
    public Memory<char> FirstName;
}
```

Rule #7: If you have an `IMemoryOwner<T>` reference, you *must* at some point dispose of it or transfer ownership (but not both).

Since a `Memory<T>` instance may be backed by either managed or unmanaged memory, it's imperative that the owner call `IMemoryOwner<T>.Dispose` when all work being performed on the `Memory<T>` instance is complete. Alternatively, the owner may *transfer ownership* of the `IMemoryOwner<T>` instance to a different component, at which point the acquiring component becomes responsible for calling `Dispose` at the appropriate time (more on this later).

Failure to call the `Dispose` method may lead to unmanaged memory leaks or other performance degradation.

This rule also applies to code which calls factory methods like `MemoryPool<T>.Rent` . The caller becomes the *owner* of the returned `IMemoryOwner<T>` and is responsible for disposing of the instance when finished.

Rule #8: If you have an `IMemoryOwner<T>` parameter in your API surface, you are *accepting ownership* of that instance.

Accepting an instance of this type signals that your component intends to *take ownership* of this instance. Your component is now responsible for proper disposal per Rule #7.

Any component handing over ownership of the `IMemoryOwner<T>` instance to a different component should *no longer use that instance* after the method call completes.

Reminder: If your constructor accepts `IMemoryOwner<T>` as a parameter, your type should also implement `IDisposable`, and your `Dispose` method should call `IMemoryOwner<T>.Dispose`.

Rule #9: If you're wrapping a synchronous p/invoke method, your API should accept `Span<T>` as a parameter.

Per Rule #1, `Span<T>` is generally the correct type to take for synchronous APIs. It is possible to pin `Span<T>` instances via the `fixed` keyword, as in the following example.

```
using System.Runtime.InteropServices;

[DllImport(...)]
private static extern unsafe int ExportedMethod(byte* pbData, int cbData);

public unsafe int ManagedWrapper(Span<byte> data)
{
    fixed (byte* pbData = &MemoryMarshal.GetReference(data))
    {
        int retVal = ExportedMethod(pbData, data.Length);

        /* error checking retVal goes here */

        return retVal;
    }

    // In the above example, 'pbData' can be null; e.g., if
    // the input span is empty. If the exported method absolutely
    // requires that 'pbData' be non-null, even if 'cbData' is 0,
    // consider the following implementation.

    fixed (byte* pbData = &MemoryMarshal.GetReference(data))
    {
        byte dummy = 0;
        int retVal = ExportedMethod((pbData != null) ? pbData : &dummy,
data.Length);
```

```
        /* error checking retVal goes here */

        return retVal;
    }
}
```

Rule #10: If you're wrapping an asynchronous p/invoke method, your API should accept `Memory<T>` as a parameter.

Since you cannot use the `fixed` keyword across asynchronous operations, the method `Memory<T>.Pin` is provided to pin `Memory<T>` instances, regardless of what kind of contiguous memory the instance represents.

The following example shows how to do use this API to perform an asynchronous p/invoke call.

```
using System.Runtime.InteropServices;

[UnmanagedFunctionPointer(...)]
private delegate void OnCompletedCallback(IntPtr state, int result);

[DllImport(...)]
private static extern unsafe int ExportedAsyncMethod(byte* pbData, int cbData,
IntPtr pState, IntPtr lpfnOnCompletedCallback);

private static readonly IntPtr _callbackPtr = GetCompletionCallbackPointer();

public unsafe Task<int> ManagedWrapperAsync(Memory<byte> data)
{
    // setup
    var tcs = new TaskCompletionSource<int>();
    var state = new MyCompletedCallbackState {
        Tcs = tcs
    };
    var pState = (IntPtr)GCHandle.Alloc();

    var memoryHandle = data.Pin();
    state.MemoryHandle = memoryHandle;

    // make the call
    int result;
    try {
        result = ExportedAsyncMethod((byte*)memoryHandle.Pointer, data.Length,
pState, _callbackPtr);
    } catch {
        ((GCHandle)pState).Free(); // cleanup since callback won't be invoked
```

```
            memoryHandle.Dispose();
            throw;
        }

        if (result != PENDING)
        {
            // Operation completed synchronously; invoke callback manually
            // for result processing and cleanup.
            MyCompletedCallbackImplementation(pState, result);
        }

        return tcs.Task;
    }

    private static void MyCompletedCallbackImplementation(IntPtr state, int result)
    {
        GCHandle handle = (GCHandle)state;
        var actualState = (MyCompletedCallbackState)state;
        handle.Free();
        actualState.MemoryHandle.Dispose();

        /* error checking result goes here */

        if (error) { actualState.Tcs.SetException(...); }
        else { actualState.Tcs.SetResult(result); }
    }

    private static IntPtr GetCompletionCallbackPointer()
    {
        OnCompletedCallback callback = MyCompletedCallbackImplementation;
        GCHandle.Alloc(callback); // keep alive for lifetime of application
        return Marshal.GetFunctionPointerForDelegate(callback);
    }

    private class MyCompletedCallbackState
    {
        public TaskCompletionSource<int> Tcs;
        public MemoryHandle MemoryHandle;
    }
```

jherby2k commented on Apr 12, 2018

Super useful guidance, thanks a bunch!

ghost commented on Aug 16, 2018

Thanks, really useful!

I've been experimenting with Span, I'm doing some machine learning research and wondering if the new constructs could allow C# to compete with Python + Numpy. I used your example above to create a p/invoke interface to the Intel MKL (Math Kernal Library), and built a simple Matrix class which wraps a buffer of native memory (allocated using AllocHGlobal). Works fine, but in languages like C++ / python etc the int datatype maps to the platform word size, i.e. Int32 or Int64. In C# it always seems to be an Int32, and the indexer for Span is int. Very large double precision matrix can very easily go over 2GB

So it looks like you can only use a Span to access 2GB of memory even if it's allocated using native code (or even using the AllocHGlobal overload which takes and IntPtr)?

Seems to me if AllocHGlobal has a IntPtr overload, so should Span? Do you know of any work arounds?

Regards
Dave

antonfirsov commented on Apr 23, 2021

This stuff is now on docs.microsoft.com, dropping a link here in case someone land on this gist from google:
https://docs.microsoft.com/en-us/dotnet/standard/memory-and-spans/memory-t-usage-guidelines