Managing Mutable State in Multithreaded Application Development:
Advantages of Using Large Mutable Structs in C# 8.0+
by Christopher P. Susie

This article explains the advantages of using large mutable structs in C# 8.0+, particularly with the DotNetVault analyzer and synchronization library, for objects that will be subject to synchronization across multiple threads. The past wisdom has been to eschew large-sized structs as well as mutable structs in C#: in the general case, this wisdom is still applicable. Yet, in the specific case of the design of objects to be shared across multiple threads and subject to a synchronization mechanism such as a Mutex or Lock, mutable structs have distinct advantages over reference types. Recent developments in the C# language, however, have mitigated concerns about using large structs and the DotNetVault library takes full advantage of this mitigation. Full code showing how a large mutable struct can be used conveniently with a BasicReadWriteVault is found at the end of this article.

1. Shared Mutable State is the Enemy of Multithreaded Applications

    a. Mitigating problem through immutability

When dealing with multithreaded applications, *shared mutable state* is the enemy. One effective strategy for dealing with this is to embrace immutability. An object of reference type is said to be immutable when there is no way (without, perhaps, reflection or unsafe code) to use any reference to the object *to change the state of the object it refers to.* The paradigmatic example of such a type in C# is *string:* a variable of string type cannot be used to change the contents of any string object. One can produce new string objects from it with their contents derived from the original object, but there is no way to change the contents of any string. Some confusion arises, particularly among beginners, as to the difference between variables of reference type (which are not immutable unless marked const or readonly) and the reference type *objects* to which the variables refer. With *string* and other reference types that are immutable, the immutability described is the immutability of the referred-to object: the variables that refer to the objects are mutable: they can be made to refer to different string objects or be made null. The crux of what they *cannot do* is cause a change to the value of the actual string object on the managed heap.

The use of immutable string objects through references in the presence of garbage collection also prevents some of the difficulties encountered in programming in languages without automatic tracing garbage collection. C++, as a counter-example, can provide understanding of how automatic garbage collection, used in conjunction with immutability, can be beneficial in multithreaded contexts.

In C++, a std::string is *mutable*. Suppose, for example, that multiple threads of execution are accessing the string through a std::string* (a raw pointer to standard string, similar to a reference in C#, but *not garbage collected*). If two threads of execution access the string through a pointer at the same time and either performs a mutation on the std::string through the pointer, they cause a data race, which is undefined behavior. The further difficulty is that if the std::string in question is destroyed on one thread, other threads have no way to know that it has been destroyed (unless such a mechanism is carefully and correctly introduced by the programmer). Access to an object after it has been destroyed is another case of undefined behavior.

Making string immutable as in C# *prevents the possibility of one thread from making modifications to a string at the same time another thread is examining the same object.* That is, use of string variables of reference type cannot be used to cause a data race *as to the* ***string object itself*** because no reference to the string object can change the string object. Automatic tracing garbage collection further helps this issue by making it impossible to access an object after it has been destroyed: if there is any way (short of unsafe code or reflection) for a program to reach an object (e.g. by using a reference to it, or indexing to it via array, etc.), the object **has not been destroyed**. Only when the Garbage Collector can establish that further access to an object is impossible will that object become subject to collection. With immutable, automatically garbage collected reference types, therefore, one may safely access the immutable object[1] from any number of threads concurrently without worry of a data race or potential access to a destroyed object: the immutable object cannot be changed and the fact that you can access it guarantees it has not been destroyed.

Immutability is not, however, a panacea. Access to the non-readonly, non-const fields that refer to the immutable objects should still be synchronized if the same field can be accessed from multiple threads of execution. Immutability also carries a performance penalty with it. Instead of using a reference to the object to make a small change to its value, such small changes with immutable objects require copying the entire value (as updated with the small change) into a new object and then assigning a reference to the new object to a variable of reference types (or re-assigning the reference returned to the original). Obviously, the entire object must be copied regardless of the smallness of the "change". This also may increase memory-usage if references to the old immutable object still exist elsewhere. Even if there was only one reference to the old immutable object and it is immediately reassigned to refer to the new immutable object, a penalty still exists: increased GC pressure. The old objects will now have to be garbage collected and frequent "changes" of this sort can result in significant GC overhead.

[1] Of course, accessing a mutable **variable** of type string and mutating it (via null assignment or assignment to a different string object) on multiple threads is a different matter entirely. Although this will not result in a *data race* causing undefined behavior as it would in C++, the outcome and ordering of the concurrent write operations is not predictable without using *Interlocked* mechanisms to synchronize the access to the variable itself. The immutable object **itself** can be safely accessed by individual variables of type *string* that all refer to the same string object.

b. Mitigating the problem by eliminating sharing (isolation)

Isolating mutable state so that it cannot be accessed by more than one thread at the same time is another way to approach multithreaded applications. This is the approach taken by the *Rust* programming language.

In Rust, values exist within a certain lifetime (defined often though not exclusively by lexical scope on the stack). Multiple non-mutable references (which cannot be used to change the value of the object) to the object can exist so long as their lifetime is shorter than or equal to the lifetime of the object and no mutable reference (capable of making a change to the underlying object) exists. If any mutable reference to the object exists, it must be the *only* reference to the object (mutable or immutable). Rust accomplishes this without garbage collection and without runtime cost through the use of its "borrow-checker", a static analyzer that is part of the Rust compiler and will refuse to compile a program unless it can statically prove that those conditions hold.

A similar approach to synchronization is more difficult in C#: the compiler does not have a built-in borrow checker and reference types, most of which are mutable, are freely shared throughout the program. This makes it easier to manage *single threaded programs*: the user does not have to concern themselves with object lifetime because if they can access the object through a reference type or array, it still exists. Its lifetime ends only sometime after such access becomes *impossible*. With mutable state, this solves the problem of use-after-free but does *not* solve the problem of concurrent access causing the state of the objects to become incoherent – to violate the invariants guarded by those classes.

2. DotNetVault Attempts to provide Rust-Like compile time guarantees by use of static analysis

DotNetVault attempts to bridge this gap by providing static analysis rules that cause compilation errors when its synchronization mechanisms are used in a way contrary to their design. It deems certain types *VaultSafe*: that is, easy to protect and isolate. A type is easy to protect and isolate if:

1. It is a provably immutable sealed reference type
2. It is an *unmanaged* value type (a value type that contains no reference types anywhere in its object graph)
3. It is a value type that contains within its graph only objects that comply with #1, #2 or (recursively, #3),
4. The analyzer is configured to recognize the type as being VaultSafe despite its inability to statically prove the same

DotNetVault provides very easy-to-use mechanisms to isolate and protect VaultSafe types. It also provides mechanisms to isolate and protect types that are not VaultSafe: these mechanisms however are more restrictive and require extra effort. It is beyond the scope of this article to address all DotNetVault's mechanisms. This article will suggest, however, that large mutable structs are uniquely suited in C# 8.0+ in general and with DotNetVault in particular, to serve as a store for mutable state shared by multiple threads of execution.

3. Advantages of Large Mutable Structs

In the old days of C#, passing large structs around could be expensive because they were always copied by value. Today, however, when used carefully, this is no longer the case.

a. Ref Returns

A value type stored as a field in a reference type, within an array or in static storage may be returned *by reference*. If the field is readonly or const, it may only be returned by *readonly reference*. If the field is mutable, it may be returned either by *mutable* or *readonly reference.* This return facility can apply to a property, a method-return value or to an indexer into an array or other collection that exposes an indexer supporting return-by-reference.

b. Passing by readonly reference with the "in" keyword (including to extension methods)

A further mitigation is that large structs can now be passed by readonly reference to constructors, methods, and extension methods by using the *in* keyword. Normally, only such values types as are defined as *readonly* structs should be passed by the *in* keyword because access to a member may make an unexpected defensive copy and if this happens multiple times within a method, will be far more expensive than merely copying it by value to the method one time. With C# 8.0 and the expectation that we will be designing these structs carefully to store our shared state, this concern can be overcome.

4. Designing Structs

    a. In General

When designing a value type in C#, the best practice is (for types intended for broad or external use):

1. to make all its fields readonly
2. declare the struct as a readonly struct in its type declaration, e.g. “public readonly struct UInt128”
3. Implement IEquatable<UInt128> and if it makes sense or is helpful IComparable<UInt128>
4. implement the ==, !=, GetHashCode(), IEquatable<T>, and override bool Equals(object other).
5. If the struct is large (say bigger than 128 bytes or so), make == and != take params with “in”
6. Define “bool Equals(UInt128 other)”, override bool Equals(object other) in terms of the equality operators (note when you define ==, != for a reference type, the opposite advice holds)
7. If you want to do IComparable, make a static method int Compare(in UInt128 lhs, in UInt128 rhs) and implement int CompareTo(UInt128 other) in terms of that function. This makes it easy to also define >, <, >= and <= in terms of Compare(in lhs, in rhs) > 0, etc.

The following provides an example of how this is done with a UInt128 value type, which is on the threshold of what should define operators and static methods as receiving parameters by readonly reference. For the example, it is so done (for a UInt256 it would almost certainly be appropriate). Also, arithmetic operators are excluded from the example, but should also accept parameters by readonly reference with “in”.

```
public readonly struct UInt128 : IEquatable<UInt128>, IComparable<UInt128>
{
    public static ref readonly UInt128 Zero => ref TheZeroVal;
    public static ref readonly UInt128 MinValue => ref Zero;
    public static ref readonly UInt128 MaxValue => ref TheMaxValue;

    public static int Compare(in UInt128 lhs, in UInt128 rhs)
    {
        int highCompare = lhs._high.CompareTo(rhs._high);
        return highCompare == 0 ? lhs._low.CompareTo(rhs._low) : highCompare;
    }

    public UInt128(ulong high, ulong low)
    {
        _low = low;
        _high = high;
    }

    public static bool operator ==(in UInt128 lhs, in UInt128 rhs) => lhs._high == rhs._high &&
             lhs._low == rhs._low;
    public static bool operator !=(in UInt128 lhs, in UInt128 rhs) => !(lhs==rhs);
    public static bool operator >(in UInt128 lhs, in UInt128 rhs) => Compare(in lhs, in rhs) > 0;
    public static bool operator <(in UInt128 lhs, in UInt128 rhs) => Compare(in lhs, in rhs) < 0;
    public static bool operator >=(in UInt128 lhs, in UInt128 rhs) => !(lhs < rhs);
    public static bool operator <=(in UInt128 lhs, in UInt128 rhs) => !(lhs > rhs);
    public bool Equals(UInt128 other) => other == this;
    public int CompareTo(UInt128 other) => Compare(in this, in other);
    public override bool Equals(object other) => other is UInt128 u128 && u128 == this;
    public override string ToString() => "0x" + _high.ToString("X8") + _low.ToString("X8");

    public override int GetHashCode()
    {
        int hash = _high.GetHashCode();
        unchecked
        {
            return hash * ((hash * 397) ^ _low.GetHashCode());
        }
    }

    private readonly ulong _high;
    private readonly ulong _low;
    private static readonly UInt128 TheZeroVal = default;
    private static readonly UInt128 TheMaxValue = new UInt128(ulong.MaxValue, ulong.MaxValue);
}
```

Figure 1: Best Practice for Typical Largish Value Type

1. *readonly* struct declaration and all readonly fields

The readonly struct declaration, together with only readonly fields, avoids the need for defensive copying of the type. When passed with "in", which is desirable because this is a large value type, you can be confident that the compiler will not emit unnecessary defensive copies of the struct. This mitigates the need advice for the old advice to avoid large value-types (preferring instead sealed classes with value-semantics): value types do just fine if they are defined readonly and define their most important operations as accepting parameters by readonly reference with "in".

2. static ref readonly special values

The struct defines several "known values" likely to be important to users of this value type at the top.

```
public static ref readonly UInt128 Zero => ref TheZeroVal;
public static ref readonly UInt128 MinValue => ref Zero;
public static ref readonly UInt128 MaxValue => ref TheMaxValue;
```

Although value types may not return *non-static fields* by reference, they may return *static* fields by reference. These properties are declared to return by readonly reference so that they cannot be used to change the value of Zero or the maximum value of the type. Exposing them by reference rather than value allows for efficient access thereto. Note that a public static readonly field would also allow for efficient access. It would be inefficient, however, to have a readonly *property* that returns by value… each access, absent optimizations, implies a value copy.

3. Implementation of IComparable<UInt128>

For many types it might not make senses to implement IComparable<T> and implement comparison operators (other than == and !=) and methods. All are undoubtedly required for a value type implementing an integer. Unfortunately, IComparable<T> does not accept the comparand by reference, but only by value. Fortunately, IComparable<T> is not likely to be preferred comparison mechanism for programmers using an integer type – they will likely rely on the operators or the handy static Compare method we provide, which *does* accept values by readonly reference.

Implementing a static comparison method that:

1. returns int > 0 for lhs > rhs, int < 0 for lhs < rhs and int ==0 for lhs == rhs and
2. accepts parameters by readonly reference

provides several benefits. First, implementing >, <, >=, and <= (also accepting parameters by readonly reference) in terms of that static comparison function is simplicity itself. For >, Compare(in lhs, in rhs) > 0; for <, Compare(in lhs, in rhs) < 0 can be applied mechanically:

```
public static bool operator >(in UInt128 lhs, in UInt128 rhs)
       => Compare(in lhs, in rhs) > 0;

public static bool operator <(in UInt128 lhs, in UInt128 rhs)
       => Compare(in lhs, in rhs) < 0;
```

For >=, and <= the solution is likewise mechanical **always implement them in terms of *!* and > or <:**

```
public static bool operator >=(in UInt128 lhs, in UInt128 rhs) => !(lhs < rhs);
public static bool operator <=(in UInt128 lhs, in UInt128 rhs) => !(lhs > rhs);
```

Then, IComparable<T>'s int CompareTo(T other) method can also be implemented mechanically

```
public int CompareTo(UInt128 other) => Compare(in this, in other);
```

While the by-value copy is a sad artifact of IEquatable<T> and IComparable<T> being developed before the possibility of passing largish value types by readonly reference, this methodology, by taking advantage of your static Compare method, does not incur any *additional* overhead: the unnecessary copy happens only once.

You may at some point want to refine your > and < to only check if > or < rather than doing the full comparison. That is certainly a reasonable thing to do if the implementation is imposing a measurable performance penalty. In many cases, any such improvement will be negligible, and the methodology provided here provides a simple way to get correct, consistent comparison operations defined in a way convenient for users of your value type. Get the type working and correct before worrying about small efficiencies.

4. Implementing IEquatable<T> and Equality Members

Unlike reference types, it is best for value types to implement all equitable comparisons in terms of the == operator. There are several reasons for this distinction. Reference types can be null, participate in polymorphic type hierarchies and offer an efficient default for equality testing (reference equality test via == or !=). Value types, however, cannot be null, do not participate in a polymorphic type hierarchy (with the exception of System.Object and System.ValueType), and provide only a slow clunky field-based equality check that requires both boxing and reflection. Implementing in terms of == and != allows for passing by readonly-reference, checking two values for equality using the intuitive operators rather than bizarre-looking (when applied to values) and slow method syntax.

Also, unlike the inequality operators >, <, >=, and <=, which do not make sense for all value types, == and != should be implemented for any value type in wide enough use that it is conceivable that a user might wish to check if two of them represent the same value. Thus, except for special situations (e.g., struct enumerators used for efficiency and throwaway structs intended only for narrow internal use), one should almost always, for value types:

1. Implement == and !=
2. Implement override bool Equals(object other)
3. Implement IEquatable<T>
4. Implement GetHashCode().

Implement both !=, override bool Equals(object other) and bool Equals(T other) in terms of == as shown:

```
public static bool operator ==(in UInt128 lhs, in UInt128 rhs)
    => lhs._high == rhs._high && lhs._low == rhs._low;
public static bool operator !=(in UInt128 lhs, in UInt128 rhs)
       => !(lhs==rhs);
public override bool Equals(object other)
       => other is UInt128 u128 && u128 == this;
public bool Equals(UInt128 other) => other == this;
```

This makes it easy to have all your equality operations consistent and using the most efficient call available.

GetHashCode() should also be implemented in a way not inconsistent with == and therefore with all the other equality operations. Failure to implement *GetHashCode()* consistently with the equality operations can cause surprises if your type is ever used as the key to a dictionary or the value stored in a set that uses hashing.

Because what should happen in *GetHashCode*() is the least intuitive of the equality members and not everyone may be entirely familiar with hashing, some background is provided. GetHashCode computes a "hash" value based on the value of an object. In C#, all hash codes are 32-bit signed integers. Hash codes are used by dictionaries and sets to determine whether a key is present and if so where it is in the set/dictionary. The absolute minimum requirement for a hash function that works is that *all objects that have the **same value** produce the same **hash code***. If this condition is not met, dictionaries and sets will not work properly. To be useful, the hash code should be quickly computable and *likely to produce different hash codes for different values*. Obviously, there is not a unique 32-bit integer for every value of every type that could be imagined (and there is not a unique 32-bit integer for every possible value of our unsigned 128-bit integer). That is two different values *may* produce the same hash code; two equal values *must* produce the same hash code.

How GetHashCode is dealt with for our unsigned 128-bit integer is straight forward:

```
public override int GetHashCode()
{
    int hash = _high.GetHashCode();
    unchecked
    {
        return ((hash * 397) ^ _low.GetHashCode());
    }
}
```

The high field's GetHashCode method is called. This value is then multiplied in an *unchecked block* (because we expect and desire overflow) by a prime number and that result is then bitwise XORed with the hash code of the low field. This is the routine that ReSharper provides when autogenerating the GetHashCode method. This is a fast routine, will always return the same hash for equal values and, hopefully, will provide distinct hash codes for values that are not equal in most data sets.

Note that you do not always have to include every field in your hash codes. If for example, the struct contained a GUID or high precision timestamp, you would probably just return the hash of the GUID or timestamp as the hash code: it is very unlikely that two objects will have the same GUID or timestamp if they are not equal.

b. As a store of mutable state subject to thread synchronization using DotNetVault

Although an entirely immutable struct as detailed above could certainly be synchronized easily using DotNetVault, requiring the entire value to be copied every time a change is made is inefficient. Moreover, the struct may not be a coherent value like a UInt128 but rather a repository of values related only in that they are subject to the same synchronization mechanism. The primary design considerations here are:

1. Avoiding defensive copies of the struct during "readonly" operations by
    a. Marking all operations that do not need to change the state of the struct as readonly
    b. Overriding all the methods from System.Object and System.ValueType and including the *readonly* specifier for them
    c. If implementing equality and comparison members, doing so by readonly reference as shown above for UInt128

2. Adhering to Vault Safety
    a. If not an unmanaged, including the VaultSafe attribute
    b. Not having any fields that are mutable reference types or otherwise not vault safe (note that the field for a reference type need not be readonly; the *type itself*, however, must be immutable)
    c. Providing any mutator methods or properties needed to manage the struct

The following is an example of a struct designed for use with a vault:

```
[VaultSafe]
public struct BigStruct : IEquatable<BigStruct>, IComparable<BigStruct>
{
    public static ref readonly BigStruct DefaultValue => ref TheDefaultValue;
    public readonly bool IsDefault => this == DefaultValue;

    public Guid First
    {
        readonly get => _first;
        set => _first = value;
    }
    public Guid Second {  get; set; }
    public Guid Third { get; set; }
    public Guid Fourth { get; set; }

    [NotNull]
    public string Name
    {
        readonly get => _name ?? string.Empty;
        set => _name = value ?? throw new ArgumentNullException(nameof(value));
    }

    public BigStruct(in Guid first, in Guid second, in Guid third, in Guid fourth,
          [NotNull] string name)
    {
        _first = first;
        Second = second;
        Third = third;
        Fourth = fourth;
        _name = name ?? throw new ArgumentNullException(nameof(name));
    }

    public static bool operator ==(in BigStruct lhs, in BigStruct rhs)
        => lhs._first == rhs._first && lhs.Second == rhs.Second &&
           lhs.Third == rhs.Third && lhs.Fourth == rhs.Fourth &&
            StringComparer.Ordinal.Equals(lhs.Name, rhs.Name);
    public static bool operator !=(in BigStruct lhs, in BigStruct rhs) => !(lhs == rhs);
    public static bool operator >(in BigStruct lhs, in BigStruct rhs) => Compare(in lhs, in rhs) > 0;
    public static bool operator <(in BigStruct lhs, in BigStruct rhs) => Compare(in lhs, in rhs) > 0;
    public static bool operator >=(in BigStruct lhs, in BigStruct rhs) => !(lhs < rhs);
    public static bool operator <=(in BigStruct lhs, in BigStruct rhs) => !(lhs > rhs);

    [SuppressMessage("ReSharper", "NonReadonlyMemberInGetHashCode")]
    public override readonly int GetHashCode()
    {
        int hash = _first.GetHashCode();
        unchecked
        {
            hash = (hash * 397) ^ Second.GetHashCode();
            hash = (hash * 397) ^ Third.GetHashCode();
            hash = (hash * 397) ^ Fourth.GetHashCode();
            hash = (hash * 397) ^ StringComparer.Ordinal.GetHashCode(Name);
        }
        return hash;
    }

    public static int Compare(in BigStruct lhs, in BigStruct rhs)
    {
        int compareRes = lhs._first.CompareTo(rhs._first);
        if (compareRes != 0) return compareRes;

        compareRes = lhs.Second.CompareTo(rhs.Second);
        if (compareRes != 0) return compareRes;

        compareRes = lhs.Third.CompareTo(rhs.Third);
        if (compareRes != 0) return compareRes;

        compareRes = lhs.Fourth.CompareTo(rhs.Fourth);
        if (compareRes != 0) return compareRes;

        return StringComparer.Ordinal.Compare(lhs.Name, rhs.Name);
    }
    public override readonly bool Equals(object other) => other is BigStruct bs && bs == this;
    public readonly bool Equals(BigStruct other) => other == this;
    public override readonly string ToString() => "First: [" + _first + "]; Second: [" + Second + "]; Third: [" +
                                                  Third + "]; Fourth: [" + Fourth + "].";
    public readonly int CompareTo(BigStruct other) => Compare(in this, in other);

    private string _name;
    private Guid _first;
    private static readonly BigStruct TheDefaultValue = default;
}
```

Figure 2 – Large Mutable Struct Designed for Use in Vault

The key here is to take extreme care that defensive copies are never made of the struct. *Readonly* structs avoid the possibility of defensive copies, but our present use-case requires mutability. This requires that all *properties and methods that do not actually change the value of the struct be explicitly marked as readonly*. Otherwise, calling *ToString()* or accessing a property's getter will cause a defensive copy if the struct is accessed in a readonly context.

Examine the properties carefully:

```
public Guid First
{
    readonly get => _first;
    set => _first = value;
}
public Guid Second {  get; set; }
public Guid Third { get; set; }
public Guid Fourth { get; set; }

[NotNull]
public string Name
{
    readonly get => _name ?? string.Empty;
    set => _name = value ?? throw new ArgumentNullException(nameof(value));
}
```

For properties that are NOT auto-implemented, you must mark the *getter* with *readonly* as shown to prevent defensive copying when accessing the get accessor. If the property were **get-only**, it would be sufficient to mark the property itself as readonly as in "`public readonly Guid Fifth => _fifth;`". If you specify a setter, the getter needs to be marked readonly. For the auto-implemented properties, the get accessor is implicitly readonly.

Care must also be taken with non-static methods: if the method does not change the value, it **must** be marked readonly to avoid defensive copying. You should therefore override Equals(object other), GetHashCode() and ToString() and include the keyword readonly in your override declaration as shown:

```
public override readonly bool Equals(object other)
        => other is BigStruct bs && bs == this;
public readonly bool Equals(BigStruct other) => other == this;
public override readonly string ToString() => "First: [" + _first + "]; Second: [" +
        Second + "]; Third: [" + Third + "]; Fourth: [" + Fourth + "].";
public readonly int CompareTo(BigStruct other) => Compare(in this, in other);

[SuppressMessage("ReSharper", "NonReadonlyMemberInGetHashCode")]
public override readonly int GetHashCode()
{
    int hash = _first.GetHashCode();
    unchecked
    {
        hash = (hash * 397) ^ Second.GetHashCode();
        hash = (hash * 397) ^ Third.GetHashCode();
        hash = (hash * 397) ^ Fourth.GetHashCode();
        hash = (hash * 397) ^ StringComparer.Ordinal.GetHashCode(Name);
    }
    return hash;
}
```

Note that the warning about non-readonly members in GetHashCode is suppressed. If this were a reference type, being able to change an object's value such that it has a different Hash would be disastrous if these objects were keys in a dictionary or set. As a value type, the problem would only materialize if the set exposed its keys by non-readonly reference. All standard .NET dictionaries return keys by value and thus there is no danger: any change you make will be a change to the copy, not a change to the value stored in the dictionary. Also, because hash sets enumerate their contents by value, this should not be concerning.

Also notice that the static methods and operators accept parameters by readonly-reference. If the member properties and methods accessed by those methods and operators were not marked readonly, each access would cause a defensive copy.

5. Using potentially large mutable structs in a BasicReadWriteVault<T>

Large mutable structs work particularly well when protected by a BasicReadWriteVault<T>, a vault that allows shared readonly locks (many threads can hold this lock simultaneously), exclusive write locks (no lock of any type may be held while any thread holds it) and upgradable readonly locks (exactly one thread may hold an upgradable readonly lock at a time: other threads may simultaneously hold a standard readonly lock). The semantics of readonly specifiers shown above and the nature of the locks (readonly lock exposes protected struct by readonly reference, writable lock exposes it by read-write reference) match wonderfully well. The example code shows how easy and convenient it is to use.

```
static class BigStructVaultExample
{
    public static void DemonstrateReadOnlyLock()
    {
        using var lck = BigStructVault.RoLock();
        Console.WriteLine($"Name: {lck.Value.Name}; First Guid: {lck.Value.First}.");

        //following line will not compile: cannot perform write operation with readonly lock
        //lck.Value.Fourth = Guid.NewGuid();

        //Note that THIS is ok: it is a totally independent deep copy
        //(probably not something you want to do frequently)
        BigStruct deepCopy = lck.Value;
        if (deepCopy != lck.Value) throw new Exception("Should be equal!");

        deepCopy.Name = "Steve";
        if (deepCopy == lck.Value) throw new Exception("Should not be equal!");

        Console.WriteLine($"Lock resource name: {lck.Value.Name}; Deep copy name: {deepCopy.Name}");
        Console.WriteLine($"Lock resource object stringified without defensive copy: {lck.Value}");
    }
    public static void DemonstrateWriteLock()
    {
        {
            using var lck = BigStructVault.Lock();
            lck.Value.Name = "Tamara";
        }
        using var roLck = BigStructVault.RoLock();
        Console.WriteLine($"Changed name to: {roLck.Value.Name}");
    }
    public static void DemonstrateUpgradableReadOnlyLock()
    {
        using var upgrRoLck = BigStructVault.UpgradableRoLock();
        if (upgrRoLck.Value.Name == "Tamara") //upgrade to writable lock if and only if name exactly
          //equals "Tamara"
        {
            Console.WriteLine("Name exactly equal to Tamara .... adding smith");
            using var rwLock = upgrRoLck.Lock(); //upgrades the lock for the scope of if block
            rwLock.Value.Name += " Smith";
        } //writable lock released here, still have ro lock
        else
        {
            Console.WriteLine("Name not exactly equal to Tamara ... not upgrading ro lock.");
        }
        Console.WriteLine($"Print name now: {upgrRoLck.Value.Name}");
    }
    public static void RunDemo()
    {
        Console.WriteLine($"Calling {nameof(DemonstrateReadOnlyLock)}: ");
        DemonstrateReadOnlyLock();
        Console.WriteLine($"Done {nameof(DemonstrateReadOnlyLock)}");
        Console.WriteLine();

        Console.WriteLine($"Calling {nameof(DemonstrateWriteLock)}: ");
        DemonstrateWriteLock();
        Console.WriteLine($"Done {nameof(DemonstrateWriteLock)}");
        Console.WriteLine();

        Console.WriteLine($"Calling {nameof(DemonstrateUpgradableReadOnlyLock)} first time: ");
        DemonstrateUpgradableReadOnlyLock();
        Console.WriteLine($"Done {nameof(DemonstrateUpgradableReadOnlyLock)} first time");
        Console.WriteLine();

        Console.WriteLine($"Calling {nameof(DemonstrateUpgradableReadOnlyLock)} second time: ");
        DemonstrateUpgradableReadOnlyLock();
        Console.WriteLine($"Done {nameof(DemonstrateUpgradableReadOnlyLock)} second time");
        Console.WriteLine();

        Console.WriteLine("Demo complete.");
    }

    private static readonly BasicReadWriteVault<BigStruct> BigStructVault =
        new BasicReadWriteVault<BigStruct>(new BigStruct(Guid.NewGuid(), Guid.NewGuid(), Guid.NewGuid(),
            Guid.NewGuid(), "Fred"));
}
```

Figure 3 Example Code: BasicReadWriteVault<BigStruct>

```
Calling DemonstrateReadOnlyLock:
Name: Fred; First Guid: d6393eef-8c16-4de7-a6cf-852ee2b509ee.
Lock resource name: Fred; Deep copy name: Steve
Lock resource object stringified without defensive copy: First: [d6393eef-
8c16-4de7-a6cf-852ee2b509ee]; Second: [9b32787d-797f-4761-8ae4-
54470e65be63]; Third: [f10b90ec-ae70-4a9c-bd0f-d712721bdf92]; Fourth:
[feab5b8a-11d3-4df8-a850-78e46d85b1a0].
Done DemonstrateReadOnlyLock

Calling DemonstrateWriteLock:
Changed name to: Tamara
Done DemonstrateWriteLock

Calling DemonstrateUpgradableReadOnlyLock first time:
Name exactly equal to Tamara .... adding smith
Print name now: Tamara Smith
Done DemonstrateUpgradableReadOnlyLock first time

Calling DemonstrateUpgradableReadOnlyLock second time:
Name not exactly equal to Tamara ... not upgrading ro lock.
Print name now: Tamara Smith
Done DemonstrateUpgradableReadOnlyLock second time

Demo complete.

G:\CjmScrews\Source\VaultAnalyzer\MasterBranch_SecondReleasePrepBranch(Story
86)_Tasks90_91\ExampleCodePlayground\bin\Release\netcoreapp3.0\ExampleCodePl
ayground.exe (process 7148) exited with code 0.
```

Figure 4 – Output from Call to `BigStructVaultExample.RunDemo()`